logiCells

Programmation conceptuelle déclarative avec la plateforme logiCells

Guide du développeur

Modèles, FunctionalP, H-Logic
et conception d'applications

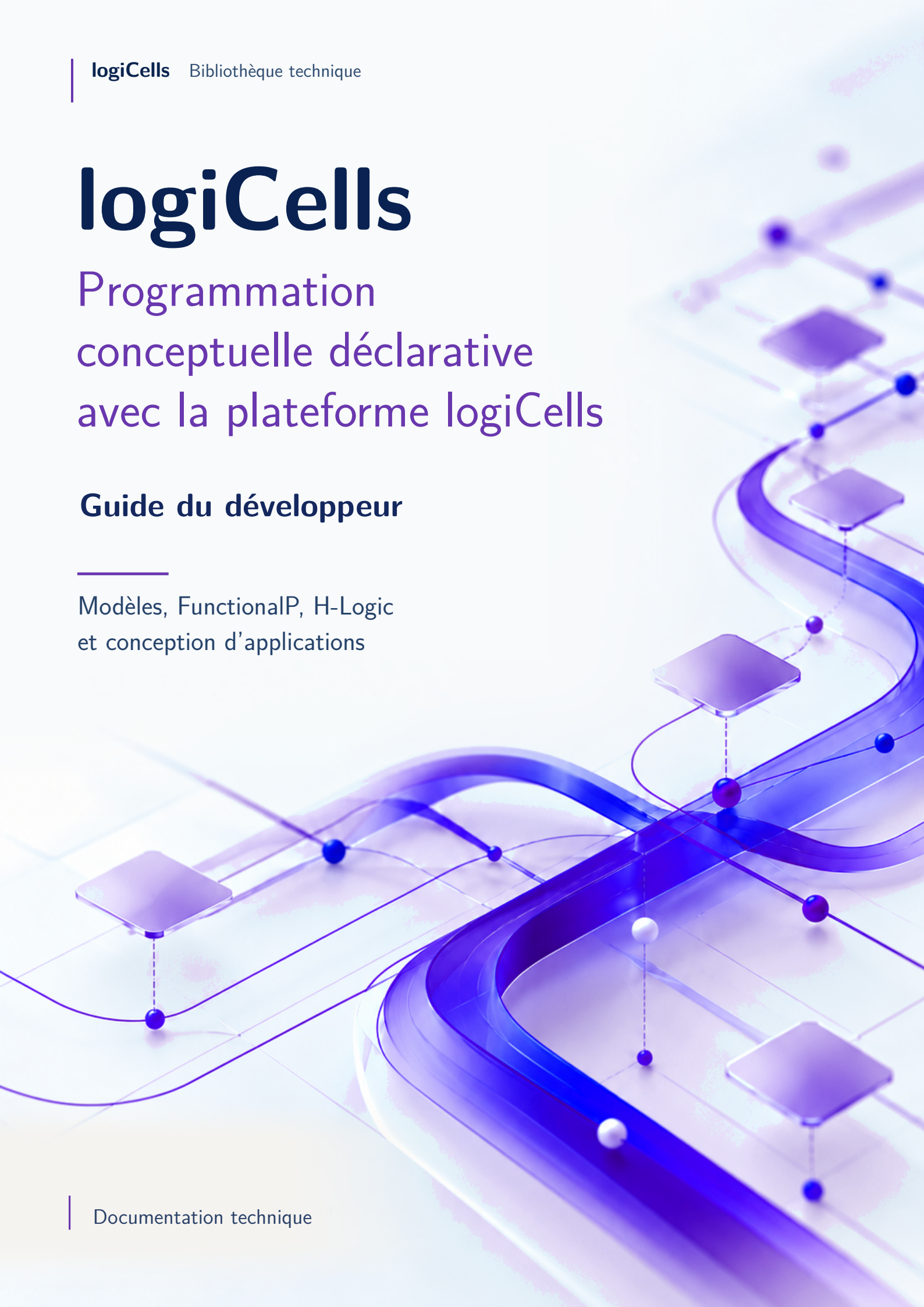


Table des matières

À propos de ce guide du développeur **20**

I Premiers pas : le modèle conceptuel exécutable **22**

1 Le modèle mental de logiCells **23**
1.1 Les applications comme systèmes conceptuels 23
1.2 Concepts 23
1.3 Facettes 24
1.4 Relations et chemins de rôles 24
1.5 Trois catégories de valeurs : scalaire, référence d'objet et relation 25
1.6 Actions 25
1.7 Événements 26
1.8 Collections et vues 26
1.9 Le runtime 26

2 Premiers pas : installer, déclarer, valider **27**
2.1 Vérifier l'outillage 27
2.2 Créer une source MODEL complète 27
2.3 Valider et inspecter 28

3 Interfaces applicatives et choix d'architecture **29**
3.1 CLI 29
3.2 API et services 29
3.3 MVP ou autre interface personnalisée 30
3.4 Vues logiCells 30
3.5 Intégration directe depuis un langage hôte externe 30
3.6 Règle de décision 30

4 Documentation, compatibilité et périmètre **31**
4.1 Choisir le contrat de référence 31
4.2 Interpréter les capacités présentées 31
4.3 Grammaire déclarative et H-Logic 31
4.4 Compatibilité des plateformes et des intégrations 32
4.5 Avant de reprendre un exemple en production 32

II Le changement de paradigme conceptuel **33**

5 Pourquoi la programmation conceptuelle déclarative ? **34**
5.1 Le problème de la fragmentation 34
5.2 Pourquoi les systèmes centrés sur le code perdent le sens métier 35

5.3 Du modèle de données au modèle conceptuel . . . 36
5.4 Déclaratif ne signifie pas passif . . . 36
5.5 logiCells, plateforme conceptuelle exécutable . . . 36
5.6 Quand utiliser la programmation conceptuelle déclarative ? . . . 36

6 Parcours de lecture et cohérence des modèles 38
6.1 La chaîne fondamentale . . . 38
6.2 Modèles de plateforme et modèles applicatifs . . . 38
6.3 Composants fonctionnels et apparence . . . 39
6.4 Ce qui assure la cohérence de l'architecture . . . 39

7 Parcours complet de programmation conceptuelle 40
7.1 Pourquoi suivre un scénario de bout en bout ? . . . 40
7.2 Le scénario métier . . . 40
7.3 Déclarer l'élément planifié . . . 41
7.4 Ajouter la projection de persistance . . . 41
7.5 Déclarer l'action au niveau de l'objet . . . 41
7.6 Déclarer le gestionnaire de planification . . . 42
7.7 Ce que démontre ce scénario . . . 42
7.8 Pour aller plus loin . . . 43

8 Des systèmes centrés sur le code aux systèmes centrés sur les concepts 44
8.1 Le réflexe de commencer par le code . . . 44
8.2 Où se trouve le sens métier ? . . . 44
8.3 Des objets aux concepts . . . 45
8.4 Des tables à la mémoire conceptuelle . . . 45
8.5 Des routes aux capacités . . . 45
8.6 Des écrans aux vues . . . 45
8.7 Conséquences architecturales . . . 45

9 Les applications comme modèles conceptuels exécutables 47
9.1 De l'application-programme à l'application-modèle . . . 47
9.2 Qu'est-ce qui rend un modèle exécutable ? . . . 47
9.3 La frontière de l'application . . . 48
9.4 Les actions font partie du modèle . . . 48
9.5 Les vues sont des projections facultatives . . . 48
9.6 Pourquoi l'exécutabilité est-elle importante ? . . . 48

10 Les concepts avant les classes, les tables et les routes 49
10.1 Le réflexe technique . . . 49
10.2 L'ordre conceptuel . . . 49
10.3 Les concepts avant les classes . . . 49
10.4 Les concepts avant les tables . . . 50
10.5 Les concepts avant les routes . . . 50
10.6 Les concepts avant les écrans . . . 50
10.7 Pourquoi cette discipline est-elle essentielle ? . . . 50

11 Métadonnées, modèles et sémantique du runtime 51
11.1 Les métadonnées ne sont pas décoratives . . . 51
11.2 Structure déclarative . . . 51
11.3 Trois autorités pour trois questions . . . 52
11.4 Des métadonnées descriptives aux métadonnées exécutables . . . 52

11.5 Sens du modèle et sens à l'exécution . . . 52
11.6 Métadonnées conceptuelles et métadonnées techniques . . . 53
11.7 Source MODEL YAML et contrat déclaratif . . . 53
11.8 Un modèle n'est pas un diagramme . . . 53

12 La grammaire déclarative exécutable 54
12.1 Familles de fichiers déclaratifs prises en charge . . . 54
12.2 Encodage des éléments YAML nommés . . . 54
12.3 Contrat de la racine MODEL . . . 55
12.4 Actions canoniques . . . 55
12.5 Les événements ne sont pas des déclarations d'action . . . 55
12.6 Collections et chemins de rôle . . . 56
12.7 Compatibilité et rédaction canonique . . . 56
12.8 Noms publics et liaisons d'implémentation . . . 56

13 Migrer les métadonnées XML existantes vers YAML 57
13.1 Chaîne de migration . . . 57
13.2 Comparer le sens, non la présentation . . . 57
13.3 Risques connus lors de la migration . . . 58
13.3.1 Valeurs préfixées par un dièse . . . 58
13.3.2 Éléments compacts ou développés . . . 58
13.3.3 Séquences polymorphes . . . 58
13.3.4 Conteneurs historiques . . . 58
13.3.5 Données intégrées aux collections . . . 58
13.4 Migrer les vues historiques vers les contrats d'interaction . . . 58
13.5 Trois niveaux de validation . . . 59
13.6 Ce qu'il ne faut pas migrer automatiquement . . . 59
13.7 Tests de non-régression par aller-retour . . . 59

III Fondements de la modélisation conceptuelle 60

14 Concepts, instances et relations 61
14.1 Concepts . . . 61
14.2 Instances . . . 62
14.3 Identifiants conceptuels . . . 62
14.4 Facettes . . . 62
14.5 Relations . . . 62
14.6 Contraintes . . . 63
14.7 Concepts, instances et relations à l'exécution . . . 63

15 Types, rôles et relations polyadiques 64
15.1 Des champs aux rôles conceptuels . . . 64
15.2 Les noms de rôles appartiennent au modèle . . . 65
15.3 Relations binaires . . . 65
15.4 Relations polyadiques . . . 66
15.5 Identité d'une relation . . . 67
15.6 Correspondances polyadiques . . . 67
15.6.1 Correspondances explicites, dérivées et contraintes non applicables . . . 68
15.7 Règles de conception . . . 68

16 Identité, contexte et portée conceptuelle 69
16.1 Identité conceptuelle . . . 69
16.2 Identité des instances . . . 69
16.3 Identité d'une relation et `this` . . . 70
16.4 Identité d'une entité et `me` . . . 70
16.5 Contexte . . . 71
16.6 Portée conceptuelle . . . 71
16.7 Redéfinition et extension . . . 71

17 Contraintes, valeurs par défaut et validité 72
17.1 Contraintes sur les concepts . . . 72
17.2 Typage des participants et chemins de rôle . . . 72
17.3 Valeurs par défaut . . . 73
17.4 Caractère nullable . . . 74
17.5 Contrats de paramètres d'action . . . 74
17.6 Invariants métier fixes . . . 74
17.7 Une validité à plusieurs niveaux . . . 75

18 Graphes conceptuels et mémoire hypergraphique 76
18.1 Pourquoi une mémoire hypergraphique ? . . . 76
18.2 Concepts, relations et hyperarêtes . . . 77
18.3 Exemple YAML : facettes hypergraphe et base de données . . . 77
18.4 Mémoire hypergraphique et exécution du runtime . . . 78
18.5 Lien avec l'IA et la mémoire sémantique . . . 78
18.6 Évaluation persistante des motifs structurels . . . 78

19 Couches conceptuelles, projections et vues 79
19.1 Le concept n'est pas sa projection . . . 79
19.2 Les facettes comme frontières de projection . . . 80
19.3 Les chemins de rôle relient les couches . . . 80
19.4 Les contrats d'interaction comme projections du modèle . . . 81
19.5 Évolution des modèles en couches . . . 81

IV H-Logic et expressions déclaratives 82

20 Fondamentaux de H-Logic 83
20.1 Pourquoi H-Logic est important . . . 83
20.2 Articulation avec YAML . . . 84
20.3 Termes, types et utilisation structurelle . . . 84
20.4 Rôles et attributs . . . 84
20.5 Application d'un type connu avec des noms de rôles locaux . . . 85
20.6 Rattachement d'interface et encapsulation des valeurs . . . 85
20.7 Termes nommés . . . 85
20.8 Quantificateurs . . . 86
20.9 Relations logiques et propriétés complémentaires . . . 86
20.10Commandes et requêtes . . . 86
20.11Définitions composites et projetées . . . 87
20.12Domaines d'utilisation privilégiés de H-Logic . . . 87
20.13H-Logic comme forme canonique d'échange . . . 87

21 Déclarer les concepts et les instances 89
21.1 Déclaration de classe . . . 89
21.2 Concepts déclarés dans une classe . . . 89
21.3 Type de classe Entity et concept Entity . . . 90
21.4 Identité implicite `me` des entités . . . 90
21.5 L'implication comme correspondance sémantique . . . 91
21.6 Champs conceptuels . . . 91
21.7 Instances comme occurrences du runtime . . . 91
21.8 Facettes de base de données et instances . . . 92
21.9 Lecture en H-Logic . . . 92

22 Déclarer des relations et des faits réifiés 93
22.1 Références par rôle . . . 93
22.2 Relations réifiées . . . 94
22.3 Pourquoi réifier une relation ? . . . 94
22.4 Persistance des relations réifiées . . . 94
22.5 Lecture en H-Logic . . . 95
22.6 Les chemins de rôles comme ponts conceptuels . . . 95

23 Contextes, interfaces et termes nommés 97
23.1 Les contextes sont des rôles ordinaires . . . 97
23.2 Ordre des rôles . . . 97
23.3 Les interfaces préservent l'identité du support . . . 97
23.4 Instance directe et rattachement d'interface . . . 98
23.5 Termes nommés explicites . . . 98
23.6 Valeurs fixées et valeurs par défaut . . . 98
23.7 Domaines de rôles complexes . . . 98
23.8 Pourquoi ces distinctions comptent . . . 99

24 Requêtes sur les structures conceptuelles 100
24.1 Formes `ask` et point d'interrogation . . . 100
24.2 Quantification typée . . . 100
24.3 Liaison à un terme nommé . . . 101
24.4 Composition logique . . . 101
24.5 La requête comme passerelle exécutable . . . 101
24.6 Intérêt des requêtes H-Logic . . . 101

25 Modes de requête H-Logic, portée et sémantique des résultats 102
25.1 Les trois modes de requête . . . 102
25.2 Recherche de motifs homomorphes . . . 102
25.3 Mode d'unification . . . 103
25.4 Mode de résolution . . . 103
25.5 Options de portée . . . 104
25.6 Sens du parcours dans la hiérarchie des types . . . 104
25.7 Instances héritées et vues adaptées . . . 104
25.8 Les mappages polyadiques préservent les informations d'égalité . . . 105
25.9 Contrat de résultat . . . 105
25.10 Itération persistante et performances . . . 106
25.11 Choisir le mode adapté . . . 106
25.12 H-Logic, autorité unique pour la recherche de connaissances . . . 106

26 Règles, contraintes et modèles éprouvés 107
26.1 Les contraintes comme engagements du modèle . . . 107
26.2 Les règles comme interprétation exécutable . . . 107
26.3 Contraintes d'expression et compatibilité des rôles . . . 107
26.4 Modèles éprouvés . . . 108
26.5 Emplacement des contraintes . . . 108
26.6 Règles de conception . . . 108
26.7 Synthèse . . . 108

27 Erreurs d'interprétation courantes de H-Logic 109
27.1 Le nom d'un rôle n'est pas une référence à un terme nommé . . . 109
27.2 La restriction de type et la liaison par égalité sont distinctes . . . 109
27.3 Les contextes sont des rôles, non des appels de fonction . . . 109
27.4 Les opérateurs `:` et `::` ont des sens différents . . . 110
27.5 Un usage structurel exige un type défini . . . 110
27.6 Les variables de requête ne portent pas de point d'interrogation . . . 110
27.7 L'ordre des rôles appartient aux métadonnées de déclaration . . . 110
27.8 Les définitions composites doivent contraindre leurs rôles publics . . . 110
27.9 Un joker n'est pas une contrainte imposant `nil` . . . 111
27.10 Une recherche sur un supertype ne retourne pas les descendants bruts . . . 111
27.11 Un mappage polyadique n'est pas une simple permutation . . . 111
27.12 Les noms locaux de rôles ne redéfinissent pas un type connu . . . 111
27.13 Un littéral n'est pas un rôle public . . . 111
27.14 L'absence de rôle commun peut interdire tout mappage valide . . . 111
27.15 Les groupes logiques imbriqués doivent rester groupés . . . 112
27.16 Un résultat unique n'est pas forcément un conteneur à un élément . . . 112
27.17 H-Logic ne remplace ni YAML, ni le XML des vues, ni functionalP . . . 112

V Des spécifications métier aux applications 113

28 Deux parcours vers les applications logiCells 114
28.1 Deux points d'entrée, deux boucles de rétroaction . . . 115
28.2 Parcours A : les modèles d'abord, H-Logic comme synthèse sémantique . . . 115
28.2.1 Contextes adaptés au parcours « modèles d'abord » . . . 116
28.3 Parcours B : H-Logic d'abord, les modèles comme opérationnalisation . . . 116
28.3.1 Verbes métier et opérationnalisation . . . 116
28.3.2 Contextes adaptés au parcours « H-Logic d'abord » . . . 117
28.4 Convergence des deux parcours . . . 117
28.5 Choisir un parcours . . . 118
28.6 Passer d'une boucle à l'autre est normal . . . 118
28.7 Adapter le niveau d'expertise au projet . . . 119
28.8 Utiliser les Skills pour entrer dans l'une ou l'autre boucle . . . 119
28.9 Pour aller plus loin . . . 120

29 Des spécifications métier à une ontologie H-Logic validée 121
29.1 Pourquoi introduire une représentation logique intermédiaire ? . . . 121
29.2 Trois représentations complémentaires . . . 122
29.2.1 Spécification en langage naturel . . . 122
29.2.2 Spécification sémantique H-Logic . . . 122
29.2.3 Artefacts déclaratifs logiCells . . . 122
29.3 Les trois boucles de validation . . . 122

29.3.1 Revue sémantique humaine . . . 122
29.3.2 Inférence assistée par LLM sous contraintes symboliques . . . 123
29.3.3 Vérification syntaxique et sémantique par le moteur LGC . . . 123
29.4 Statut et provenance des énoncés de l'analyse . . . 123
29.5 Une méthode de construction reproductible . . . 124
29.5.1 Étape 1 : rédiger une spécification concise . . . 124
29.5.2 Étape 2 : extraire et normaliser le vocabulaire . . . 124
29.5.3 Étape 3 : déterminer les besoins d'identité . . . 124
29.5.4 Étape 4 : déclarer les types fondamentaux avant leur utilisation structurelle 125
29.5.5 Étape 5 : identifier les vocabulaires métier exhaustifs . . . 125
29.5.6 Étape 6 : propager le temps et le contexte dans toute la règle . . . 125
29.5.7 Étape 7 : exprimer les calculs par des relations auxiliaires typées . . . 126
29.5.8 Étape 8 : dériver des faits de diagnostic positifs . . . 126
29.5.9 Étape 9 : concevoir les accès aux requêtes à partir des usages réels . . . 126
29.5.10 Étape 10 : définir les déductions attendues avant la génération . . . 127
29.6 Le rôle des questions dans une analyse assistée par LLM . . . 127
29.7 Points de contrôle avant génération . . . 127

30 Étude de cas : analyse temporelle des stocks ERP en H-Logic 128
30.1 Spécification métier . . . 128
30.2 Examiner le vocabulaire avant de rédiger les règles . . . 128
30.3 Types fondamentaux et vocabulaire exhaustif . . . 129
30.4 Entités métier temporelles identifiées . . . 129
30.5 Faits dérivés et relations auxiliaires . . . 130
30.6 Classification des mouvements . . . 130
30.7 Cohérence des stocks et propagation des besoins de réapprovisionnement . . . 131
30.8 Représenter les états invalides par des faits explicables . . . 131
30.9 Déduire les index des motifs de requêtes . . . 132
30.10 Scénario manuel de propagation . . . 133
30.11 Liste de contrôle pour la revue métier . . . 133
30.12 Tâches d'analyse assistée par LLM . . . 134
30.13 Plan de tests du moteur . . . 134

31 De H-Logic validé aux modèles déclaratifs 135
31.1 Opérationnaliser plutôt que traduire textuellement . . . 135
31.2 Générer des concepts à partir des types identifiés . . . 135
31.3 Générer un modèle d'entité temporelle . . . 136
31.4 Générer des relations à partir des faits dérivés . . . 137
31.5 Les décisions que H-Logic ne prend pas seul . . . 137
31.6 Deux modes de génération . . . 138
31.6.1 Génération pilotée par logiCells . . . 138
31.6.2 Génération assistée par LLM et Skills . . . 138
31.7 Vérifier les artefacts générés . . . 139
31.8 Traçabilité de bout en bout . . . 139
31.9 Une stratégie de test à quatre niveaux . . . 139
31.10 Artefacts de projet recommandés . . . 140
31.11 Pourquoi ce parcours est important . . . 140

VI Construire des applications 141

32 Formats de sources et conventions de projet 142
32.1 Règle de choix du format source . . . 142
32.2 Convention YAML pour les champs . . . 143
32.3 VIEW est volontairement différent . . . 143
32.4 Règle de migration des formulaires et cadres historiques . . . 143
32.5 Ressources d'extension de pages HTTP . . . 144
32.6 Ne pas confondre les autorités . . . 144

33 Manifestes de classes : chargement, modes et conventions 145
33.1 Ce que garantit un manifeste . . . 145
33.2 Attributs du manifeste . . . 145
33.3 Chargement d'une classe depuis un conteneur `.models` multiclasse . . . 146
33.4 Choisir le préfixe d'un identifiant de classe . . . 147
33.5 Doublons et références entre dossiers . . . 147
33.6 Unicité globale des identifiants de classe . . . 147
33.7 Événements dans les manifestes . . . 148
33.8 Validation recommandée . . . 148

34 Modèles d'application 149
34.1 Rôle d'un modèle d'application . . . 149
34.2 Espaces de noms . . . 149
34.3 Règles de conception . . . 149
34.4 Synthèse . . . 149

35 Collections et vues conceptuelles 150
35.1 Les collections comme parcours porteurs de sens . . . 150
35.2 Les vues comme projections . . . 150
35.3 Collections, curseurs et contexte . . . 150
35.4 Règles de conception . . . 150
35.5 Synthèse . . . 150

36 Les actions comme intentions métier 151
36.1 Pourquoi les actions appartiennent aux concepts . . . 151
36.2 Actions d'instance et actions de classe . . . 151
36.3 Implémentations scriptées et natives . . . 152
36.4 Les paramètres comme frontières conceptuelles . . . 152
36.5 Actions publiées . . . 152
36.6 Actions au sein des processus . . . 152
36.7 Actions et événements . . . 153

37 Premiers pas avec functionalP dans logiCells 154
37.1 Où rencontre-t-on functionalP ? . . . 154
37.2 Script minimal dans les métadonnées . . . 154
37.3 Variables et affectation . . . 155
37.4 Self, Sender et Result . . . 155
37.5 Accéder à PageControler . . . 155
37.6 Lire un dataset . . . 155
37.7 Initialiser l'état local d'une page . . . 156
37.8 Filtrer avant l'ouverture d'un dataset . . . 156
37.9 Enregistrer l'objet courant . . . 156

37.10Créer un objet . . . 157
37.11Dispatcher un événement visuel . . . 157
37.12Appeler l'objet sous-jacent d'un composant visuel . . . 157
37.13Quand déplacer la logique ailleurs ? . . . 157
37.14Erreurs fréquentes chez les débutants . . . 158
37.15Parcours de lecture pratique . . . 158

38 Guide de décision : que créer et à quel moment ? 159
38.1 La chaîne de décision centrale . . . 159
38.2 Choisir le parcours de construction avant les artefacts . . . 159
38.3 Tableau de décision . . . 160
38.4 Questions rapides avant toute création . . . 161
38.5 Décisions fréquentes chez les débutants . . . 161
38.5.1 Concept ou champ ? . . . 161
38.5.2 Relation ou clé étrangère ? . . . 162
38.5.3 Action ou événement ? . . . 162
38.5.4 Action ou workflow ? . . . 162
38.5.5 Vue ou cadre réactif ? . . . 162
38.6 Mauvaises pratiques et corrections . . . 162
38.7 Pour aller plus loin . . . 163

39 Les événements comme changements conceptuels observables 164
39.1 Événements et actions . . . 164
39.2 Événements locaux et distants . . . 164
39.3 Les événements comme frontières de processus . . . 165
39.4 Les contrôleurs comme observateurs . . . 165
39.5 Événements visuels . . . 166

40 Flux de travail et processus 167
40.1 Le processus comme objet conceptuel . . . 167
40.2 Coordination synchrone et asynchrone . . . 167
40.3 Le rôle de `lgc#process` . . . 168
40.4 Le modèle de processus comme frontière de complexité . . . 168
40.5 Exemple de processus synchrone . . . 168
40.6 Exemple de processus asynchrone . . . 169
40.7 Règles de conception . . . 170

41 La messagerie asynchrone comme modèle de plateforme 171
41.1 Concepts de messagerie fournis par la plateforme . . . 171
41.2 Le message en tant qu'objet conceptuel . . . 171
41.3 États des messages . . . 172
41.4 Distribution d'un événement à partir d'un message . . . 172
41.5 La conversation comme contexte de coordination . . . 173
41.6 Messagerie locale et distante . . . 173
41.7 Organisation en packages . . . 174

42 Rôles, autorisations et visibilité contextuelle 175
42.1 Les rôles comme positions conceptuelles . . . 175
42.2 Frontières d'autorisation . . . 175
42.3 Visibilité contextuelle . . . 175
42.4 Règles de conception . . . 175
42.5 Synthèse . . . 175

43 Règles de conception des modèles applicatifs 176
43.1 Séparer les responsabilités . . . 176
43.2 Modéliser avant de coder . . . 176
43.3 Publier de manière délibérée . . . 176
43.4 Présenter les exemples par paires . . . 176
43.5 Synthèse . . . 176

44 Objets du runtime, conteneurs et contextes d'exécution 177
44.1 De la déclaration à l'objet du runtime . . . 177
44.2 Éléments de classe et éléments d'objet . . . 178
44.3 Le conteneur d'objets . . . 178
44.4 Self, contexte et liaison à l'exécution . . . 178
44.5 Persistance des changements conceptuels . . . 179
44.6 Règle de conception . . . 179

45 Collections, jointures et modèles interrogeables 180
45.1 Les collections comme contrats de navigation du runtime . . . 180
45.2 Les relations comme structures interrogeables . . . 180
45.3 Collections issues de jointures . . . 181
45.4 Chemins de rôles . . . 182
45.5 Modèles interrogeables et complexité . . . 182

46 Configuration de l'application et assemblage des packages 183
46.1 Configuration de lancement du runtime : YAML `.lgc` . . . 183
46.2 Identité applicative et bureau . . . 184
46.3 Chargement du graphe et des métadonnées . . . 184
46.4 Éléments de package, chemins de schémas, chemins de vues et bureaux . . . 184
46.5 Alias de bases de données . . . 185
46.6 Packages binaires et packages de modèles . . . 185
46.7 Espaces de noms de la plateforme et de l'application . . . 186
46.8 Services et fichiers . . . 186
46.9 Capacités réutilisables de la plateforme . . . 186
46.10 Ressources locales et distantes du runtime . . . 187
46.11 Du package au modèle chargé . . . 187
46.12 Compiler et démarrer une application en ligne de commande . . . 188
46.12.1 Reconnaître un démarrage réussi . . . 189
46.13 Règle de conception . . . 189
46.14 Configurations initiales en mémoire et persistantes . . . 190
46.15 Métadonnées compilées et initialisation du graphe sémantique . . . 190

47 Compilation incrémentale et impact 191
47.1 Identifier l'autorité avant de choisir une commande . . . 191
47.2 Procédure de travail pour le développeur . . . 191
47.3 Distinguer source, aspect et dépendance . . . 192
47.4 NativeStore ne remplace pas le graphe sémantique . . . 192
47.5 Démarrage à froid, démarrage à chaud et cache YAML physique . . . 192
47.6 Analyser une défaillance sans affaiblir les invariants . . . 193
47.7 Analyse d'impact dans VS Code : respecter les responsabilités . . . 193
47.8 Liste de vérification du développeur . . . 193

48 Services déclaratifs du runtime et modèles d'exécution 194
48.1 De la déclaration au comportement du runtime . . . 194
48.2 Trois niveaux de service . . . 194
48.3 Attributs de définition courants . . . 194
48.4 Sections complémentaires . . . 195
48.5 Modèles architecturaux intégrés . . . 195
48.6 Exemple : définition d'un service HTTP . . . 196
48.7 Pages HTTP virtuelles et fichiers d'extension de page . . . 196
48.8 Ajouter un nouveau modèle de service du runtime . . . 196
48.9 Stratégie de référence documentaire . . . 197

49 Déploiement, conditionnement et distribution du runtime 198
49.1 Plateformes d'exécution prises en charge . . . 198
49.2 Disponibilité des plateformes et certification des sous-systèmes . . . 198
49.3 Arborescence de déploiement . . . 199
49.4 Assembler les ressources de plateforme et d'application . . . 200
49.5 Configuration selon l'environnement . . . 200
49.6 Mises à jour et évolution du runtime . . . 200
49.7 Déploiement en conteneur . . . 201
49.8 Règle de conception . . . 201

50 Plan de référence pour un nouveau module ERP 202
50.1 1. Partir d'une capacité métier délimitée . . . 202
50.2 2. Arborescence recommandée . . . 202
50.3 3. Définir d'abord l'identité et les invariants du modèle . . . 202
50.4 4. Exprimer les intentions métier dans des actions réutilisables . . . 203
50.5 5. Employer les collections comme modèles de lecture réutilisables . . . 203
50.6 6. Confier la coordination aux processus . . . 203
50.7 7. Ajouter le package VIEW après le contrat métier . . . 204
50.8 8. Tester le contrat par couches . . . 204
50.9 9. Définition de fin de réalisation . . . 204

VII Sémantique du runtime 205

51 Des déclarations à l'exécution dans le runtime 206
51.1 Le parcours d'exécution dans le runtime . . . 206
51.2 ClassItem et ObjectItem . . . 206
51.3 Les conteneurs du runtime . . . 207
51.4 Les facettes à l'exécution . . . 207
51.5 Les actions comme points d'entrée du runtime . . . 207
51.6 Événements et exécution réactive . . . 208

52 Mémoire conceptuelle et propagation des changements 209
52.1 La mémoire comme graphe conceptuel . . . 209
52.2 Du changement local à sa propagation . . . 209
52.3 Changements observables . . . 210
52.4 Règles de conception . . . 210
52.5 Synthèse . . . 210

53 Contextes d'évaluation **211**
53.1 Pourquoi le contexte importe . . . 211
53.2 Contextes courants . . . 211
53.3 Propagation du contexte . . . 211
53.4 Règles de conception . . . 211
53.5 Synthèse . . . 212

54 Transactions, cohérence et identité **213**
54.1 Identité conceptuelle . . . 213
54.2 Cohérence entre facettes . . . 213
54.3 Frontières transactionnelles . . . 213
54.4 Cohérence au niveau du processus . . . 213
54.5 Règles de conception . . . 214
54.6 Lorsque les métadonnées, le graphe sémantique et SQL évoluent ensemble . . . 214
54.7 Synthèse . . . 214

55 Exécution réactive et routage des événements **215**
55.1 Les événements comme liens déclaratifs . . . 215
55.2 Les événements métier . . . 216
55.3 Les événements visuels . . . 216
55.4 Les événements de processus . . . 216
55.5 Les événements de messagerie . . . 216
55.6 Sources d'événements et héritage . . . 217
55.7 Distribution et propagation . . . 217

56 Services, actions et publication **218**
56.1 L'action comme frontière de service . . . 218
56.2 La signature déclarative . . . 219
56.3 Accès par service Web . . . 219
56.4 Appels de service synchrones . . . 219
56.5 Passerelles de service asynchrones . . . 219
56.6 Sécurité et périmètre de publication . . . 220
56.7 Règles de conception . . . 220

57 Frontières du runtime et points d'extension **221**
57.1 Frontière du modèle . . . 221
57.2 Frontière d'implémentation . . . 221
57.3 Frontière de publication . . . 221
57.4 Frontière de messagerie . . . 221
57.5 Règles de conception . . . 221
57.6 Les liaisons d'implémentation ne sont pas des concepts métier . . . 222
57.7 Synthèse . . . 222

58 Cycle de vie du runtime, propriété des ressources et arrêt **223**
58.1 Durée de vie mémoire et durée de vie opérationnelle . . . 223
58.2 Catégories de propriété des ressources . . . 223
58.3 Ordre d'arrêt des services et des workers . . . 223
58.4 Propriété au niveau de l'ABI . . . 224
58.5 Cycle de vie des ressources d'IA résidentes . . . 224
58.6 Cycle de vie du client riche . . . 224
58.7 Liste de contrôle du développeur . . . 224

VIII CLI, API et intégration externe 226

59 Utiliser l'outil en ligne de commande `cell` 227
59.1 Architecture . . . 227
59.2 Options globales . . . 227
59.3 Commencer par `doctor` . . . 227
59.4 Explorer l'espace de travail, les applications et les packages . . . 227
59.5 Écrire et inspecter des modèles MODEL . . . 228
59.6 Explorer la sémantique . . . 228
59.7 Scripts, services, runtime et débogueur . . . 228
59.8 Publication d'applications : MCP et OpenAPI . . . 229
59.9 Exécution des agents et sessions persistantes . . . 229
59.10 Règle de conformité au contrat . . . 230

60 Publication des API et des services 231
60.1 Objectif . . . 231
60.2 De la signature d'action à l'opération API . . . 231
60.3 Mappage des requêtes et des réponses . . . 231
60.4 Groupes de services . . . 232
60.5 Publication d'actions par des services web . . . 232
60.6 Pages HTTP virtuelles et sélection des routes . . . 232
60.7 API orientées processus . . . 232
60.8 Contrats lisibles par machine : OpenAPI et MCP . . . 232
60.9 Sémantique des erreurs et de la validation . . . 233
60.10 Règles de conception . . . 233

61 Publier des applications avec OpenAPI et MCP 234
61.1 La publication part du contrat applicatif . . . 234
61.2 Compiler une application pour MCP . . . 234
61.2.1 Héberger la surface MCP générée . . . 235
61.3 Compiler une application pour OpenAPI . . . 235
61.4 Publication au niveau de l'application ou du package . . . 235
61.5 Les deux directions MCP . . . 236
61.6 Discipline des types et de la validation . . . 236
61.7 Parcours de publication recommandé . . . 236
61.8 Projeter un protocole n'autorise pas l'exécution . . . 237
61.9 Appels MCP entrants et autorité du runtime . . . 237
61.10 Configurer et tester le serveur publié . . . 237

62 Adaptateurs déclaratifs d'API externes 238
62.1 Les API externes comme frontières conceptuelles . . . 238
62.2 Concepts de requête, de réponse et de service . . . 238
62.3 Modèle générique d'API paginée . . . 239
62.4 De l'adaptateur d'API au fournisseur de collection . . . 240
62.5 Erreurs, statuts et traçabilité . . . 240
62.6 Règles de conception . . . 240

63 Intégration avec les systèmes externes 241
63.1 L'intégration comme gestion des frontières . . . 241
63.2 Intégration synchrone . . . 241
63.3 Intégration asynchrone . . . 242
63.4 Intégration aux bases de données . . . 242

63.5 Intégration des messages et des événements . . . 242
63.6 Agents et exécution externe . . . 242
63.7 Règles de conception . . . 243

64 Modèles déclaratifs et ABI du runtime 244
64.1 Disponibilité actuelle . . . 244
64.2 Identité stable des invocations . . . 244
64.3 Modèle de valeurs . . . 244
64.4 La propriété fait partie du contrat ABI . . . 245
64.5 Vocabulaire complet de la propriété . . . 245
64.6 Politiques de libération . . . 245
64.7 Binding Python . . . 246
64.8 Binding C# . . . 246
64.9 Paramètres et réécriture des valeurs . . . 246
64.10Placement natif et RPC . . . 246
64.11Manifestes ABI et informations de cible . . . 247
64.12Portée des plateformes pour le chargement ABI natif . . . 247
64.13Comportements ABI validés . . . 247
64.14Règle pour les développeurs . . . 247

65 Règles de conception de la publication 248
65.1 Publier des capacités, non des détails internes . . . 248
65.2 Utiliser des signatures typées . . . 248
65.3 Séparer les canaux de publication . . . 248
65.4 Versionner les contrats . . . 248
65.5 Synthèse . . . 248

IX logiCells et intelligence artificielle 249

66 Mémoire conceptuelle pour les applications d'IA 250
66.1 Objectif . . . 250
66.2 Au-delà de la recherche documentaire . . . 250
66.3 Un contexte structuré pour l'IA . . . 250
66.4 Les actions comme points d'exécution contrôlés . . . 251
66.5 Mémoire conceptuelle et explications . . . 251
66.6 Règles de conception . . . 251

67 Positionnement de logiCells dans les architectures neuro-symboliques 252
67.1 La taxonomie de Kautz comme carte architecturale . . . 253
67.1.1 Pipelines symbolique–neuronal–symbolique . . . 253
67.1.2 Perception neuronale suivie d'un raisonnement symbolique . . . 253
67.1.3 Systèmes symboliques produisant des données d'apprentissage neuronal . 253
67.1.4 Symbolic[Neuro] . . . 254
67.2 Première position de logiCells : la composition gouvernée . . . 254
67.3 Deuxième position : déduction Symbolic[Neuro] . . . 255
67.4 Pourquoi l'hypergraphe transforme-t-il le problème d'intégration ? . . . 256
67.5 Des hypergraphes, pas seulement des graphes binaires . . . 256
67.6 Engagements de conception . . . 257
67.6.1 Ne pas confondre performance neuronale et correction formelle . . . 257
67.6.2 Toute intuition est versionnée . . . 257
67.6.3 Les propositions ne sont pas des conclusions . . . 258

67.6.4 Un runtime natif aux responsabilités ciblées . . . 258
67.7 Périmètre architectural et objectifs explicitement exclus . . . 258
67.8 Terminologie et références principales . . . 258
67.9 État actuel et évolution envisagée . . . 259
67.10 Synthèse du positionnement . . . 259

68 Agents opérant sur les modèles conceptuels 261
68.1 Objet du chapitre . . . 261
68.2 Principes fondamentaux . . . 261
68.3 Frontière de publication et MCP . . . 262
68.4 Utiliser les contrats d'interaction VIEW sans afficher de formulaire . . . 262
68.5 Invocation des actions publiées par un agent . . . 263
68.6 Coordination entre agents et processus . . . 263
68.7 Agents et messagerie . . . 263
68.8 Automatisation avec intervention humaine . . . 263
68.9 Agents d'IA et barrière symbolique . . . 263
68.10 Règles de conception . . . 263

69 Runtime Agent, objectifs et sessions persistantes 265
69.1 Exécution ponctuelle d'un agent . . . 265
69.2 Sources de connaissances et requêtes H-Logic . . . 265
69.3 Termes canoniques du runtime Agent . . . 266
69.4 Sessions persistantes des agents . . . 266
69.5 Goals : les objectifs . . . 267
69.6 Confirmations en attente . . . 267
69.7 Instantanés et évaluation . . . 268
69.8 MCP dans l'exécution des agents . . . 268
69.9 Fournisseurs locaux . . . 268
69.10 Tester un scénario d'agent . . . 268
69.11 Effets externes typés et preuves associées . . . 269

70 Contrat d'exécution gouvernée des agents 270
70.1 Modèle d'autorité . . . 270
70.2 Contexte d'exécution indépendant du protocole . . . 270
70.3 Métadonnées d'action gouvernée . . . 271
70.4 Sémantique de PendingConfirmation . . . 271
70.4.1 Rejet . . . 271
70.4.2 Confirmation . . . 271
70.5 Exécution d'outils MCP sortants . . . 272
70.6 Exécution entrante d'actions publiées par MCP . . . 272
70.7 OpenAPI et autres adaptateurs de publication . . . 272
70.8 Scénarios de régression requis . . . 273
70.9 Critères d'acceptation du projet de démarrage . . . 273
70.10 Vérifier les actions publiées avant déploiement . . . 273
70.11 Séparer assistance au développement et connaissances partagées . . . 274

71 Processus conceptuels avec intervention humaine 275
71.1 Les tâches humaines comme étapes de processus . . . 275
71.2 Les vues comme interfaces de tâche . . . 275
71.3 Assistance par l'IA . . . 275
71.4 Règles de conception . . . 275
71.5 Synthèse . . . 275

72 Explicabilité, traçabilité et gouvernance 276
72.1 Une explicabilité fondée sur la structure . . . 276
72.2 La traçabilité par les événements et les journaux . . . 276
72.3 Gouvernance des capacités publiées . . . 276
72.4 Gouvernance de l'IA . . . 276
72.5 Règles de conception . . . 276
72.6 Synthèse . . . 277

X Pratiques d'ingénierie conceptuelle 278

73 Tester les modèles conceptuels 279
73.1 Objectif . . . 279
73.2 Que faut-il tester ? . . . 279
73.3 Structurer un package de tests déclaratifs . . . 279
73.4 Déclarer une suite de tests sous forme de modèle . . . 280
73.5 Un cas de test par action déclarée . . . 281
73.6 Assertions en functionalP . . . 281
73.7 Organiser les suites par contrat . . . 282
73.8 Exécuter les tests déclaratifs avec sclgc . . . 282
73.9 Tester les actions publiées . . . 282
73.10Tester les processus . . . 283
73.11Tester les comportements asynchrones . . . 283
73.12Tester les vues . . . 283
73.13Régression et évolution du modèle . . . 283
73.14Synthèse . . . 283

74 Déboguer les applications déclaratives 284
74.1 Parcours de diagnostic . . . 284
74.2 Défaillances fréquentes . . . 284
74.3 Journalisation . . . 284
74.4 Déboguer le modèle avant le script . . . 284
74.5 Ordre de vérification . . . 284
74.6 Utiliser des noms explicites dans les traces . . . 285
74.7 Diagnostic du chargement et de la sémantique . . . 285
74.8 Déboguer une action FunctionalP intégrée à YAML . . . 285
74.9 Synthèse . . . 286

75 Débogage source de FunctionalP dans YAML 287
75.1 Fonctionnalités disponibles et cible de développement . . . 287
75.2 Séparer conception et exécution . . . 287
75.3 La provenance source est un produit de compilation à part entière . . . 287
75.4 Identité et état des points d'arrêt . . . 288
75.5 Projection des sources dans DAP et commandes du runtime . . . 288
75.6 La correspondance source et la compilation incrémentale se rejoignent sur l'identité 289
75.7 Scénarios d'acceptation . . . 289

76 Journalisation, diagnostic et traçabilité 290
76.1 Unités de trace . . . 290
76.2 Diagnostic . . . 290
76.3 Traçabilité . . . 290
76.4 Que faut-il journaliser ? . . . 290

76.5 Diagnostic des événements . . . 290
76.6 Traçabilité et gouvernance . . . 291
76.7 Synthèse . . . 291

77 Performances, profilage et complexité des modèles 292
77.1 Aucun résultat de benchmark dans cette révision . . . 292
77.2 Mesurer le parcours complet . . . 292
77.3 Complexité des modèles . . . 293
77.4 Collections et jointures . . . 293
77.5 Requêtes H-Logic et hypergraphiques . . . 293
77.6 Frontières locales et distantes . . . 293
77.7 Clients riches . . . 293
77.8 Mesures du runtime IA . . . 293
77.9 Contrat de publication des benchmarks . . . 294

78 Modèles d'extension et erreurs fréquentes 295
78.1 Modèles d'extension recommandés . . . 295
78.2 Points d'extension des composants et du runtime . . . 295
78.3 Erreurs fréquentes . . . 295
78.4 Règle de sûreté pour les extensions . . . 296
78.5 Synthèse . . . 296

79 Gouvernance de l'évolution des modèles 297
79.1 Catégories de changements . . . 297
79.2 Compatibilité . . . 297
79.3 Documentation . . . 297
79.4 Synthèse . . . 297

XI Conclusion et références 298

80 Fonctionnalités du runtime et liste de vérification 299
80.1 Parcours essentiels du développeur . . . 299
80.2 Conditions préalables à l'exploitation . . . 299
80.3 Fonctionnalités facultatives ou en évolution . . . 300
80.4 Cohérence de la documentation . . . 300

81 Ce que change la programmation conceptuelle 301
81.1 Du code au sens exécutable . . . 301
81.2 Des écrans aux projections . . . 301
81.3 Des API aux capacités . . . 301
81.4 Des prompts d'IA aux propositions gouvernées . . . 301
81.5 D'une capacité « prise en charge » à un statut étayé par des preuves . . . 302
81.6 Ce qui relève toujours de l'ingénierie classique . . . 302
81.7 Synthèse . . . 302

82 Organisation d'un projet logiCells 303
82.1 Périmètre . . . 303
82.2 Arborescence publique recommandée . . . 303
82.3 Ce qui appartient à la couche MODEL . . . 303
82.4 Ce qui appartient à la couche VIEW . . . 304

83 Aide-mémoire de modélisation conceptuelle 305
83.1 Notions abordées . . . 305
83.2 Liste de contrôle du modèle conceptuel . . . 305
83.3 Liste de contrôle d'un nouveau module déclaratif . . . 305
83.4 Conseils concernant les espaces de noms . . . 306
83.5 Signaux de mauvaise conception . . . 306

84 Aide-mémoire H-Logic 307
84.1 Définitions et instances . . . 307
84.2 Rôles, attributs et ordre . . . 307
84.3 Interfaces et termes nommés . . . 307
84.4 Contraintes . . . 307
84.5 Relations logiques . . . 307
84.6 Quantificateurs et requêtes . . . 308
84.7 Modes de requête et portée . . . 308
84.8 Définitions composites . . . 308
84.9 Formes à éviter . . . 309

85 Motifs H-Logic vérifiés par les tests 310
85.1 Valeurs primitives et collections . . . 310
85.2 Définitions, rôles et attributs . . . 310
85.3 Instances nommées et anonymes . . . 310
85.4 Rattachement d'interface . . . 310
85.5 Quantificateurs . . . 311
85.6 Relations logiques et compléments . . . 311
85.7 Définitions composites et projetées . . . 311
85.8 Commandes et requêtes . . . 311
85.9 Catégories de tests négatifs . . . 311
85.10Règle documentaire . . . 312

86 Modèles d'actions déclaratives 313
86.1 Périmètre . . . 313
86.2 Actions d'objet . . . 313
86.3 Actions de classe . . . 313
86.4 Actions publiées . . . 313

87 Modèles d'événements conceptuels 314
87.1 Périmètre . . . 314
87.2 Un événement signale un changement observable . . . 314
87.3 Événements locaux, visuels et distants . . . 314
87.4 Événements de processus . . . 314

88 Modèles de processus métier 315
88.1 Périmètre . . . 315
88.2 Quand créer un modèle de processus ? . . . 315
88.3 Processus synchrone . . . 315
88.4 Processus asynchrone . . . 315
88.5 Anti-modèle . . . 315

89 Modèles d'assistants d'intelligence artificielle 316
89.1 Périmètre . . . 316
89.2 Un assistant fondé sur les concepts . . . 316
89.3 Lire, expliquer, agir . . . 316
89.4 Garde-fous . . . 316

90 Recettes de programmation du runtime 317
90.1 Périmètre . . . 317
90.2 Créer un objet . . . 317
90.3 Rechercher un concept . . . 317
90.4 Exécuter un processus . . . 317

91 Éléments essentiels du langage functionalP 318
91.1 Objectif . . . 318
91.2 Structure d'un programme . . . 318
91.3 Séquences . . . 319
91.4 Valeurs primitives . . . 319
91.5 Affectations et variables . . . 319
91.6 Arithmétique, comparaisons et logique booléenne . . . 319
91.7 Listes et intervalles . . . 320
91.8 Structures de contrôle . . . 320
91.9 Fonctions . . . 320
91.10 Application partielle et lambdas . . . 321
91.11 Expressions case . . . 321
91.12 Expressions pointées et objets du runtime . . . 321
91.13 Littéraux d'hypergraphe et constructions de processus . . . 322
91.14 Erreurs d'interprétation fréquentes . . . 322
91.15 Ordre d'apprentissage recommandé . . . 322

92 Glossaire de la programmation conceptuelle 323
92.1 Périmètre . . . 323
92.2 Notions essentielles . . . 323

93 Guide de diagnostic 324
93.1 Points à vérifier . . . 324
93.2 Une vue n'affiche pas les données attendues . . . 324
93.3 Une action n'est pas accessible depuis l'extérieur . . . 324
93.4 Un processus est difficile à déboguer . . . 324
93.5 Un composant dépend trop d'une plateforme . . . 324

À propos de ce guide du développeur

Destiné aux développeurs et intégrateurs tiers, ce guide montre comment logiCells transforme les concepts métier en modèles exécutables et en contrats applicatifs réutilisables. Il présente les fondements de la programmation conceptuelle déclarative à travers des situations de développement concrètes. Son objet est le modèle conceptuel exécutable et le runtime qui l'interprète : concepts, relations, contraintes, actions, processus, persistance, publication, tests, diagnostics et automatisation gouvernée.

L'architecture MVC/View de logiCells n'est *pas* un prérequis à l'utilisation de la plateforme. Une application conceptuelle peut être exploitée depuis une interface en ligne de commande, des API et des services, une interface utilisateur organisée selon le modèle MVP ou toute autre frontière d'intégration. L'architecture View constitue une option pour les clients natifs ou Web avancés qui nécessitent un état local, un cache, des interactions asynchrones et les mécanismes de coordination correspondants entre le client et le runtime.

C'est pourquoi les développements détaillés consacrés à View/MVC figurent désormais dans l'ouvrage compagnon, *Guide des vues pour client riche logiCells*. Le présent guide se limite aux choix d'architecture et aux frontières de publication indispensables pour comprendre la place du client.

Plateformes du runtime et liaisons externes

Le runtime logiCells actuel est disponible sous Windows, Linux et macOS et peut être déployé dans des conteneurs. L'intégration directe par ABI est actuellement proposée au moyen de liaisons Python et C#. L'empaquetage propre à chaque plateforme et les liaisons vers les langages hôtes constituent des projections opérationnelles d'un même modèle conceptuel : ils ne modifient pas ce modèle.

Comment utiliser ce guide

Les nouveaux mécanismes sont introduits selon l'ordre suivant :

```
besoin metier
-> concept ou intention
-> declaration minimale
-> resultat observable
-> contrat et mecanismes du runtime
```

Listing 1 – Progression pédagogique de ce guide

Le lecteur doit pouvoir expliquer la signification du modèle avant d'avoir à décrire la façon dont un framework d'interface, un chargeur ou une classe particulière du runtime l'implémente.

Dénomination des éléments dans les exemples

Les exemples emploient les noms publics ou documentaires des types du runtime, plutôt que les noms de classes propres au langage d'implémentation. Lorsqu'un projet historique contient des identifiants de classe préfixés selon les conventions d'implémentation, considérez-les comme des détails de compatibilité propres à ce projet, et non comme un vocabulaire que tout nouveau développeur doit apprendre.

Première partie

Premiers pas : le modèle conceptuel exécutable

Chapitre 1

Le modèle mental de logiCells

Pour comprendre logiCells, le plus simple est de ne pas commencer par les classes, les tables, les routes ou les écrans. Ces éléments existent toujours, mais ils ne constituent pas le point de départ. Celui-ci est le modèle conceptuel : ce que l'application connaît, la manière dont cette connaissance est structurée, ce qu'il est possible d'en faire, comment l'observer et comment la projeter vers des utilisateurs, des services ou des agents.

Une application logiCells peut être comprise comme un réseau de concepts, d'instances, de relations, de facettes, d'actions, d'événements, de collections et de vues. Le runtime donne un sens opérationnel à ces déclarations. Le développeur travaille ainsi simultanément à deux niveaux : le niveau conceptuel, où le sens métier est déclaré, et le niveau du runtime, où ces déclarations deviennent des structures exécutables.

1.1 Les applications comme systèmes conceptuels

Un fichier de modèle logiCells commence généralement par une déclaration `class`. Ici, le terme « classe » ne doit pas être compris dans son seul sens orienté objet. Il désigne le point d'entrée d'un élément conceptuel du runtime. La déclaration porte un nom, un identifiant de classe, un type, d'éventuelles informations relatives au générateur, des informations sur l'objet, puis les sections conceptuelles qui définissent son comportement.

```yaml
class:
  name: UserContact
  classId: lgc#user_contact
  className: CustomClassItem
  currentGeneratorName: Lgc.Common.UserContact
  type: Entity
  generators: []
  object: {}
  concepts: []
  collections: []
  actions: []
  events: []
  views: []
```

Listing 1.1 – Structure de premier niveau dans la syntaxe YAML de logiCells

Cette structure donne une première carte de la plateforme. Un modèle n'est pas seulement une structure de données : c'est une description organisée du sens que le runtime devra interpréter.

1.2 Concepts

Un concept définit la catégorie d'élément à laquelle le modèle s'intéresse. Il peut déclarer des contraintes et des facettes. Un modèle courant consiste à déclarer un concept `Entity` et à

préciser qu'il implique un concept plus général de la plateforme, tel que `lgc#entity`.

```yaml
concepts:
  - Entity:
      constraints:
        - implies: lgc#entity
```

Listing 1.2 – Concept et implication dans le YAML MODEL canonique

Cette implication n'est pas un simple attribut décoratif du modèle. Elle situe le concept dans une hiérarchie conceptuelle que le runtime et les outils peuvent interpréter.

1.3 Facettes

Les facettes permettent de distinguer plusieurs aspects complémentaires d'un même concept. La facette `hypergraph` décrit des champs conceptuels. La facette `database` décrit la correspondance avec le stockage persistant. D'autres facettes peuvent être introduites pour prendre en charge des projections spécialisées.

```yaml
facets:
  - main:
      type: hypergraph
      fields:
        - NextExecution: DateTime
        - Executed: Boolean
  - ScheduleActionItem:
      type: database
      map: ScheduleActionItem
      fields:
        - IdScheduleActionItem:
            type: String
            rolePath: Id
        - NextExecution: DateTime
        - Executed: Boolean
```

Listing 1.3 – Facettes dans la syntaxe YAML de logiCells

La règle de compréhension est simple : un concept ne se réduit pas à une représentation physique. Il peut posséder plusieurs facettes, chacune répondant à une préoccupation différente du runtime.

1.4 Relations et chemins de rôles

Les relations sont souvent représentées par des chemins de rôles. Dans le modèle `UserContact`, des champs tels que `User.Id`, `User.ClassId`, `Contact.Id` et `Contact.ClassId` montrent que le modèle ne se contente pas de stocker des colonnes indépendantes. Il associe des champs à des rôles conceptuels.

```yaml
fields:
  - IdUser:
      type: String
      rolePath: User.Id
      nullable: true
  - ClassIdUser:
      type: String
      rolePath: User.ClassId
      nullable: true
  - IdContact:
      type: String
      rolePath: Contact.Id
      nullable: true
  - ClassIdContact:
      type: String
      rolePath: Contact.ClassId
```

```
17        nullable: true
```

Listing 1.4 – Chemins de rôles dans la syntaxe YAML de logiCells

Les chemins de rôles occupent une place essentielle dans le modèle mental de logiCells : ils rendent explicite la correspondance entre les champs de stockage et les positions conceptuelles.

1.5 Trois catégories de valeurs : scalaire, référence d'objet et relation

Une source fréquente d'erreurs consiste à traiter toutes les valeurs JSON, YAML ou issues d'un formulaire comme des scalaires. Dans logiCells, trois contrats doivent être distingués dès le début.

Catégorie	Forme habituelle	Signification
Scalaire	`Caption: "Ready"`	Valeur telle que `String`, `Integer`, `Boolean`, `Float` ou `DateTime`.
Référence d'objet typée	`Id + ClassId + FormalName`	Référence à un objet conceptuel. Elle ne doit pas être ramenée à une chaîne arbitraire si le runtime attend une identité d'objet.
Relation conceptuelle	Rôles nommés / chemins de rôles	Fait reliant plusieurs participants, au moins deux. Les champs de stockage tels que `Owner.Id` et `Owner.ClassId` sont des projections de ces rôles, non la relation elle-même.

Pour une référence typée, la représentation externe transporte généralement de manière explicite le triplet d'identité :

```
{
  "Id": "42",
  "ClassId": "lgc#authorized_role",
  "FormalName": "quality-manager"
}
```

Listing 1.5 – Référence typée plutôt que chaîne non typée

Lorsqu'un contrat publié attend cette référence, une charge utile telle que `"AuthorizedBy": "quality-manager"` n'est pas un raccourci équivalent, à moins que l'adaptateur ne le documente explicitement. Supposer cette équivalence peut conduire à un faux succès : la requête peut être acceptée alors que la relation n'a pas réellement été résolue.

La même rigueur s'applique à `DateTime`. La couche de publication doit définir une représentation acceptée et rejeter les valeurs mal formées à la frontière de la requête. Ne comptez pas sur des erreurs de base de données ou de persistance pour révéler une date invalide plus tard dans la transaction.

Pour `ClassId`, utilisez exactement l'identifiant canonique publié. Ne présumez ni une conversion automatique de la casse, ni l'insertion de traits de soulignement, ni un repli vers le nom, sauf si le contrat définit explicitement cette normalisation.

1.6 Actions

Les actions expriment ce qu'il est possible de faire avec un élément conceptuel. Dans la mesure du possible, leur nom doit décrire une intention métier. Une action peut porter sur une

capacité au niveau de la classe ou d'un objet ; elle peut également être publiée pour être appelée par d'autres parties du runtime.

Ainsi, un gestionnaire de planification peut proposer `ScheduleAction` au niveau de la classe. Un élément planifié peut, lui, proposer `Execute` au niveau de l'objet. Cette paire forme une chaîne opérationnelle : le gestionnaire recherche un élément en attente, puis cet élément exécute son propre comportement.

1.7 Événements

Les événements représentent des changements observables ou des signaux du runtime. Ils rendent le modèle réactif. Un événement n'a pas vocation à dupliquer toute la logique de l'action : il rend visible un changement significatif afin que d'autres composants puissent réagir.

Dans un runtime conceptuel, l'intérêt des événements ne se limite pas à leur distribution technique. Ils préservent la traçabilité : ils peuvent préciser le concept modifié, l'action à l'origine du changement, l'objet concerné et le comportement déclenché en aval.

1.8 Collections et vues

Les collections définissent des projections interrogeables de structures conceptuelles. Les vues définissent la manière de présenter ces projections. Une collection peut être simple, issue d'une jointure, agrégée, adossée à une liste d'objets ou reliée à un fournisseur. Une vue peut viser un élément de classe, un élément d'objet, une liste, un formulaire, un tableau de bord ou toute autre surface de publication.

La séparation essentielle est la suivante : les collections organisent l'accès aux données conceptuelles ; les vues organisent leur présentation. Les unes comme les autres sont des projections du modèle.

1.9 Le runtime

Le runtime interprète et exécute le modèle. Il charge les déclarations, résout la structure, construit les éléments d'exécution, expose les collections, route les actions, propage les événements et publie les vues ou les services. Il doit donc être considéré comme l'environnement d'exécution des déclarations conceptuelles.

Les technologies d'implémentation restent internes au runtime. Le modèle public adopté par cet ouvrage est celui de logiCells lui-même : concepts, facettes, rôles, collections, actions, événements, vues et services.

Chapitre 2

Premiers pas : installer, déclarer, valider

Cette première mise en pratique est volontairement indépendante de toute architecture d'interface utilisateur. Installez l'édition Community actuelle de logiCells depuis `logiCells.ai` ; pour les modalités d'installation, reportez-vous à la distribution en vigueur. La commande publique utilisée par le code exécutable fourni avec ce guide est `cell`.

2.1 Vérifier l'outillage

```
cell doctor
```

Listing 2.1 – Vérifier les outils installés

2.2 Créer une source MODEL complète

Créez le fichier `Task.model.yaml`. La structure ci-dessous respecte la grammaire déclarative exécutable : les générateurs nommés, les concepts, les facettes, les champs et les paramètres utilisent leur nom d'affichage comme clé ; les éléments enfants polymorphes, tels que les actions, conservent leur type d'élément explicite.

```yaml
class:
  classId: lgc#task
  name: Task
  className: CustomClassItem
  type: Entity
  currentGeneratorName: TaskManagement.Task

  generators:
    - TaskManagement.Task:
        nameFieldRecord: Task
        nameField: Title
        class: GUIDStringGeneratorInfo
        params: ''

  object:
    className: CustomObjectItem
    objectName: Task

  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
        facets:
          - main:
              type: hypergraph
```

```
            fields:
              - Title: String
              - Status: String
        - Task:
            type: database
            map: lgcTask
            fields:
              - IdTask:
                  type: String
                  size: 32
                  rolePath: Id
                  generatorName: TaskManagement.Task
              - ClassIdTask:
                  type: String
                  size: 32
                  rolePath: ClassId
                  defaultValue: lgc#task
              - Title:
                  type: String
                  size: 200
              - Status:
                  type: String
                  size: 50
                  defaultValue: Open
```

Listing 2.2 – Première source MODEL complète : Task.model.yaml

Les noms indiqués dans `className` et dans le champ `class` du générateur sont des identifiants de liaison au runtime imposés par la grammaire actuelle. La documentation emploie les noms publics des liaisons et n'expose pas les noms de classes préfixés selon les conventions du langage d'implémentation.

2.3 Valider et inspecter

```
cell declarative validate Task.model.yaml
```

Listing 2.3 – Valider la source MODEL

Pour automatiser l'opération, ajoutez `-json` : la sortie standard contient alors un résultat de commande exploitable par un programme.

```
cell --json declarative validate Task.model.yaml
```

Listing 2.4 – Validation avec résultat lisible par machine

Inspectez ensuite soit le fichier déclaratif, soit les métadonnées MODEL reconnues par l'outillage :

```
cell declarative inspect Task.model.yaml
cell model metadata inspect Task.model.yaml
```

Listing 2.5 – Inspecter la source

Cette séquence établit la boucle fondamentale utilisée tout au long du guide : déclarer, valider, inspecter, puis assembler une application ou lui ajouter une interface uniquement lorsque cela est nécessaire.

Chapitre 3

Interfaces applicatives et choix d'architecture

Le modèle conceptuel exécutable constitue le cœur d'une application logiCells. L'architecture client détermine la manière d'interagir avec ce modèle ; elle ne définit pas le modèle lui-même.

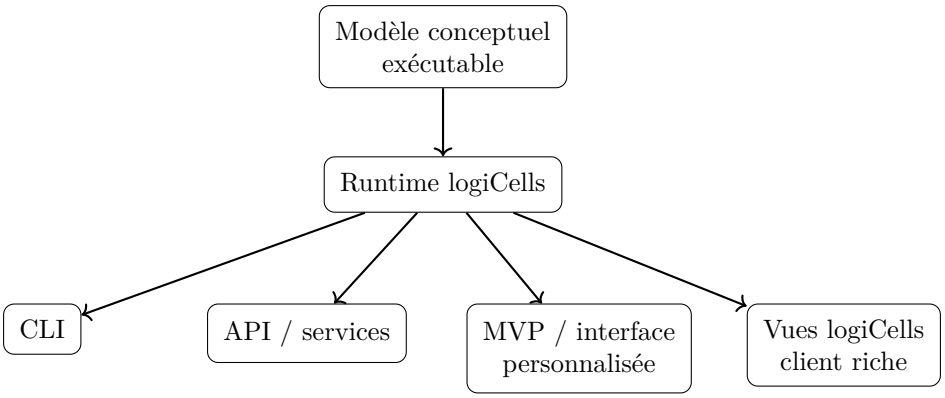


FIGURE 3.1 – Plusieurs interfaces applicatives peuvent agir sur un même modèle conceptuel.

3.1 CLI

Une application en ligne de commande peut charger le modèle, y accéder et invoquer ses capacités conceptuelles sans introduire de framework visuel. Cette approche convient à l'outillage, à l'administration, à l'automatisation, aux tâches de développement et à l'exécution sans interface graphique.

3.2 API et services

Les actions publiées peuvent être exposées par des adaptateurs de services. La route ou le protocole n'est qu'une surface de publication d'un contrat d'action existant ; il ne doit pas devenir une seconde définition du sens métier. Les publications OpenAPI et MCP en sont des exemples concrets. Toutes deux peuvent être générées à partir d'un contrat d'application ou de package, comme l'explique Chapitre 61, "Publier des applications avec OpenAPI et MCP".

3.3 MVP ou autre interface personnalisée

Une interface utilisateur peut adopter une architecture de type MVP ou une autre technologie de présentation, tout en continuant à appeler les capacités du modèle et à observer l'état du runtime. Le modèle conceptuel demeure indépendant de cette architecture de présentation.

3.4 Vues logiCells

L'architecture View est une option destinée aux clients riches. Retenez-la lorsque le client doit bénéficier de projections gérées par logiCells, de jeux de données (*DataSets*) et de curseurs locaux, d'un cache, d'un routage des événements, d'interactions asynchrones, d'un rendu natif ou Web, ou d'une coordination spécifique entre client et runtime.

Ses conventions détaillées ne font volontairement pas partie du parcours d'initiation de cet ouvrage. Consultez le guide compagnon *Guide des vues pour client riche logiCells* lorsque ces capacités sont nécessaires.

3.5 Intégration directe depuis un langage hôte externe

Lorsqu'un processus externe a besoin d'une intégration directe au runtime, plutôt que d'un service publié, l'ABI du runtime fournit une frontière typée. Les liaisons actuellement proposées sont **Python** et **C#**. Leurs règles de durée de vie des objets, notamment la distinction entre résultats empruntés et résultats dont l'appelant possède la responsabilité, figurent dans Chapitre 64, "Modèles déclaratifs et ABI du runtime".

3.6 Règle de décision

Choisissez l'architecture d'interface la plus simple qui satisfasse les besoins de l'application. N'introduisez pas de packages View, de DataSets locaux ou de routage visuel des événements au seul motif qu'ils existent dans la plateforme. Utilisez-les lorsque le client a effectivement besoin des capacités qu'ils apportent.

Chapitre 4

Documentation, compatibilité et périmètre

Une application logiCells fiable repose sur une compréhension précise de son modèle, de la version du runtime et de ses frontières d'intégration. Ce chapitre indique quelles références consulter lors de l'implémentation et du déploiement.

4.1 Choisir le contrat de référence

Pour la version déployée, consultez dans cet ordre :

1. La grammaire acceptée par le compilateur installé et les API publiques du runtime.
2. Les manifestes ABI, les bindings et les descripteurs OpenAPI/MCP générés pour cette version.
3. Les références du runtime et de la ligne de commande correspondant à cette version.
4. Les exemples conceptuels et les modèles d'intégration présentés dans cet ouvrage.

Lorsqu'un exemple diffère de l'interface réellement distribuée, privilégiez l'interface versionnée et signalez la divergence. N'adaptez pas un modèle métier pour contourner un tutoriel devenu obsolète.

4.2 Interpréter les capacités présentées

Ce guide distingue les *interfaces disponibles*, les *modèles d'intégration applicatifs* et les *extensions prévues*. La disponibilité d'une interface n'implique pas un comportement identique sur tous les systèmes, fournisseurs, transports ou modes de déploiement. Un modèle conceptuel montre comment composer les composants ; la configuration, les autorisations et les essais d'exploitation restent propres à l'application.

4.3 Grammaire déclarative et H-Logic

Les déclarations MODEL doivent respecter la grammaire reconnue par le compilateur cible. Les définitions et requêtes H-Logic suivent de même l'analyseur fourni avec le runtime ; les formes `define .:\#...` ainsi que `ask homomorphic`, `ask unifying` et `ask resolving` sont utilisées dans ce livre. Une syntaxe plus ancienne provenant d'un autre exemple reste illustrative tant que les outils installés ne l'acceptent pas.

4.4 Compatibilité des plateformes et des intégrations

Le runtime, les modules natifs, les bibliothèques d'interface et les bindings ont chacun leurs conditions de compatibilité. Avant le déploiement, vérifiez les distributions et versions d'hôtes prises en charge. Pour les actions publiées par API ou MCP et susceptibles de modifier l'état, configurez l'authentification, les autorisations et, si nécessaire, la confirmation humaine sur le chemin réel des appels entrants : publier une action n'autorise jamais, à elle seule, son exécution.

4.5 Avant de reprendre un exemple en production

1. Vérifiez les versions du compilateur, du runtime, du binding et du point d'accès Tooling.
2. Compilez le modèle avec les outils de ligne de commande installés.
3. Testez les demandes autorisées et refusées, ainsi que les frontières de rôle et de session.
4. Vérifiez la persistance, les reprises, la récupération après incident et l'arrêt sur l'hôte visé.
5. Consignez les décisions et limites propres à votre application.

Ces pratiques rendent le guide utilisable d'une version à l'autre, sans transformer la documentation développeur en rapport d'essais interne.

Deuxième partie

Le changement de paradigme conceptuel

Chapitre 5

Pourquoi la programmation conceptuelle déclarative ?

La plupart des applications métier reposent sur des concepts, mais ceux-ci sont rarement considérés comme des éléments exécutables à part entière. Un utilisateur, un contact, une action à exécuter, une tâche planifiée, une file d'attente, un import, un export ou un processus d'intégration peut être représenté simultanément à plusieurs endroits : métadonnées du modèle, correspondances de persistance, collections, vues, actions scriptables, points de terminaison de service, événements du runtime et écrans d'administration. Chaque représentation répond à un besoin, mais aucune ne devrait, à elle seule, dissimuler le sens métier.

logiCells part d'un autre principe : l'application est d'abord décrite comme un modèle conceptuel exécutable. Ce modèle déclare les concepts, facettes, champs, relations, collections, actions, événements, vues et structures de publication. Le runtime matérialise ensuite ces déclarations sous forme de structures exécutables. Il serait donc réducteur de considérer logiCells comme un simple format de configuration : ses déclarations font partie intégrante du modèle de programmation.

Cette édition utilise YAML pour les métadonnées MODEL et réserve XML aux sources canoniques `*.view.xml`. L'ancien format XML des modèles MODEL peut apparaître dans les explications de migration, mais il n'est pas proposé pour la création de nouveaux modèles.

5.1 Le problème de la fragmentation

Dans le développement traditionnel, le sens métier se disperse souvent entre plusieurs artefacts. Une action planifiée, par exemple, peut être représentée par une table de base de données, une classe serveur, un service d'interrogation périodique, un script, un message de journal, une entrée de file d'attente et un écran d'administration. L'application fonctionne, mais le concept même d'action planifiée ne se présente plus comme un ensemble cohérent.

logiCells limite cette fragmentation : le modèle d'une classe peut rassembler plusieurs dimensions d'un même concept. La déclaration YAML suivante montre comment un élément d'action planifiée peut réunir son identité, ses champs destinés à l'hypergraphe, sa projection en base de données et son action exécutable dans un seul modèle.

```yaml
actions:
  - action:
      name: Execute
      type: ObjectItem
      caption: "#Execute"
      published: true
      params:
        - Result:
            dataType: Boolean
            paramType: ptReturn
```

Listing 5.1 – Déclaration d'action dans le YAML MODEL canonique

Cet extrait d'action prend tout son sens dans la déclaration MODEL qui l'entoure. YAML est la notation canonique enseignée pour créer de nouveaux modèles MODEL ; l'ancien format XML n'est abordé qu'aux frontières de migration. Les formes YAML canoniques employées dans ce livre sont récapitulées à la Section 12.2, "Encodage des éléments YAML nommés".

```yaml
class:
  name: ScheduleActionItem
  classId: lgc#schedule_action_item
  className: CustomClassItem
  type: Entity
  generators:
    - Lgc.Common.ScheduleActionItem:
        class: GUIDStringGeneratorInfo
  object:
    className: CustomObjectItem
    objectName: ScheduleActionItem
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
        facets:
          - main:
              type: hypergraph
              fields:
                - NextExecution: DateTime
                - Executed: Boolean
                - Priority: Integer
          - ScheduleActionItem:
              type: database
              map: ScheduleActionItem
              fields:
                - IdScheduleActionItem:
                    type: String
                    generatorName: Lgc.Common.ScheduleActionItem
                    rolePath: Id
                - NextExecution: DateTime
                - Executed: Boolean
                - Priority: Integer
  actions:
    - action:
        name: Execute
        type: ObjectItem
        published: true
        params:
          - Result:
              dataType: Boolean
              paramType: ptReturn
```

Listing 5.2 – Projection équivalente du modèle logiCells en YAML

L'essentiel n'est pas la syntaxe en elle-même. Le modèle réunit ce qui est souvent séparé ailleurs : identité conceptuelle, persistance, champs du runtime et comportement exécutable.

5.2 Pourquoi les systèmes centrés sur le code perdent le sens métier

Le code est précis, mais il n'est pas toujours le meilleur support pour exprimer une intention métier. Une méthode peut exécuter un élément planifié ; prise isolément, elle ne précise ni comment cet élément est identifié, ni son type conceptuel, ni ses règles de persistance, ni son exposition dans une vue, ni le générateur de son identité, ni la possibilité de le publier comme capacité du runtime.

Dans une architecture centrée sur le code, le lecteur doit reconstruire le domaine à partir des détails d'implémentation. Dans un modèle logiCells, il peut partir des concepts déclarés, puis suivre leurs projections : champs, facettes, actions, collections, vues et événements. Le code reste présent, mais il se rattache à des intentions explicites, au lieu d'être le seul endroit où les découvrir.

5.3 Du modèle de données au modèle conceptuel

Un modèle de données définit ce qui peut être stocké. Un modèle conceptuel décrit ce que représente l'application et la manière dont ses concepts peuvent être utilisés. Une facette de base de données définit éventuellement des champs et des correspondances, mais le concept dépasse la table : il peut posséder une facette d'hypergraphe, des contraintes, des actions, des vues et des événements.

Cette différence apparaît dans les deux usages des facettes. Une facette `hypergraph` exprime la structure conceptuelle ; une facette `database` exprime la persistance. Elles sont liées sans remplir le même rôle. logiCells les conserve au sein du même modèle, afin que la persistance demeure une projection du sens conceptuel et non un schéma autonome.

5.4 Déclaratif ne signifie pas passif

Le terme *déclaratif* prête parfois à confusion : on peut l'associer à une configuration statique, à de la documentation ou à des diagrammes. Dans logiCells, une déclaration possède une portée exécutable : elle peut être chargée, interprétée, instanciée, interrogée, projetée et exécutée.

La déclaration d'une action l'illustre clairement. Le modèle peut déclarer une action nommée `Execute`, son niveau de ciblage, ses paramètres et son statut de publication. Le runtime peut ensuite l'exposer comme une capacité de l'objet ou de la classe concernée.

La déclaration n'exécute toutefois rien par elle-même, isolément. Elle devient exécutable parce que le runtime sait interpréter l'élément action, sa cible, ses paramètres ainsi que le script ou le gestionnaire qui lui est associé.

5.5 logiCells, plateforme conceptuelle exécutable

On peut voir logiCells comme une plateforme qui transforme des déclarations conceptuelles en structures de runtime. Elle fournit une mémoire conceptuelle, une structure de modèles, des mécanismes d'action et d'événement, des collections, des vues, des services et des modes de publication. Ces éléments ne sont pas des frameworks indépendants réunis après coup : ils s'organisent autour d'un même modèle déclaré.

La conséquence est architecturale. L'application n'est plus, avant tout, un ensemble de classes, de tables, d'écrans et de routes. Elle se définit d'abord par ses concepts et ses capacités déclarés. Les classes, tables, écrans et routes deviennent des projections ou des supports d'implémentation.

5.6 Quand utiliser la programmation conceptuelle déclarative ?

Cette approche est particulièrement adaptée aux applications dont la sémantique métier est riche et qui comportent de nombreuses relations, plusieurs projections, des permissions, des processus, des vues dynamiques, des services publiés et des besoins d'évolution à long terme. Elle présente moins d'intérêt pour des scripts ponctuels ou de très petites applications lorsque l'effort de modélisation dépasserait les bénéfices attendus.

Elle convient notamment aux systèmes de type ERP, aux applications de gestion opérationnelle, aux plateformes de processus, aux applications fondées sur les connaissances, aux plateformes d'intégration et aux systèmes assistés par l'IA, pour lesquels une mémoire conceptuelle structurée constitue un atout important.

Chapitre 6

Parcours de lecture et cohérence des modèles

Ce guide suit un fil conducteur unique : une application logiCells naît sous la forme d'un modèle conceptuel déclaratif, puis devient exécutable grâce au runtime. Les concepts définissent le vocabulaire ; les relations les relient ; les facettes les projettent en mémoire ou dans la persistance ; les actions expriment les capacités ; les événements rendent les changements observables ; les modèles de processus coordonnent le travail ; les vues projettent l'état ; les services publient des capacités ; enfin, les agents interviennent sur la surface conceptuelle ainsi constituée.

6.1 La chaîne fondamentale

```
MODEL
  YAML physique + grammaire déclarative exécutable
        |
        | compiler / assembler si nécessaire
        v
APPLICATION
  métadonnées applicatives résolues et packages actifs
        |
        | compilation / projection sémantique si nécessaire
        v
SEMANTIC
  sources H-Logic et hypergraphe conceptuel compilé
```

Listing 6.1 – Trois autorités, un même cycle de vie applicatif

Il s'agit de représentations liées d'un même projet, mais elles ne constituent pas des autorités interchangeables. Une question portant sur les fichiers relève de MODEL. Une question sur l'héritage, les packages actifs ou les redéfinitions relève d'APPLICATION. Une question sur les relations sémantiques, les correspondances, les requêtes ou H-Logic relève de SEMANTIC. La suite du livre explique ensuite comment les actions, événements, processus, vues, services et agents interviennent au bon niveau de représentation résolue.

Cette distinction simplifie l'apprentissage : modifier un modèle ne requiert pas de connaître le fonctionnement interne du moteur d'hypergraphe ; lorsque l'application exige un raisonnement conceptuel explicite, l'exploration sémantique demeure accessible.

6.2 Modèles de plateforme et modèles applicatifs

L'espace de noms `lgc#` désigne les concepts de plateforme et les mécanismes réutilisables du runtime. L'espace de noms `lbs#` peut servir aux exemples applicatifs. Cette séparation clarifie

la documentation publique : la plateforme fournit des mécanismes génériques, tandis qu'une application les emploie pour modéliser son propre domaine.

6.3 Composants fonctionnels et apparence

Un modèle visuel déclare des composants fonctionnels. Chacun décrit la fonction attendue dans une vue : champ, liste, bouton, grille, cadre ou zone de tableau de bord. Le composant effectif résulte de l'enregistrement d'une implémentation concrète. La vue conceptuelle ne définit donc pas l'apparence finale ; elle définit la projection fonctionnelle du concept.

Certains composants sont abstraits : ils décrivent une fonction qui ne peut pas être instanciée directement. Une implémentation concrète propre à la plateforme doit être enregistrée pour que le runtime puisse créer le composant effectif. Cette règle préserve la portabilité du modèle entre les projections pour ordinateur, mobile, Web et agents.

6.4 Ce qui assure la cohérence de l'architecture

Les mêmes principes conceptuels s'appliquent partout : un objet métier porte son comportement local ; un objet relation donne une identité aux liens importants ; un objet processus coordonne plusieurs participants ; un contrôleur réagit aux événements ; une action publiée définit une frontière de service ; enfin, la messagerie permet la coordination asynchrone. Les chapitres suivants détaillent ces modèles sans imposer de parcours propre à un client ni de conception d'interface particulière.

Chapitre 7

Parcours complet de programmation conceptuelle

Ce chapitre suit un scénario logiCells modeste, mais réaliste : la planification et l'exécution d'actions du runtime. L'exemple se rapproche volontairement des fichiers de modèle existants. Il fait intervenir un `ScheduleActionItem`, qui représente l'élément à exécuter, et un `ActionSchedulerManager`, qui représente le gestionnaire chargé de rechercher et d'exécuter le travail planifié.

Il ne s'agit pas de décrire chaque détail d'implémentation, mais de montrer le passage de la déclaration conceptuelle au comportement exécutable.

7.1 Pourquoi suivre un scénario de bout en bout ?

La programmation conceptuelle peut paraître abstraite si elle n'est présentée qu'au moyen de définitions. Un scénario suivi donne au lecteur un parcours concret : déclarer un concept, lui associer des champs, définir sa persistance, exposer une action, interroger une collection, puis déclencher un comportement à l'exécution.

Voici la chaîne que nous suivrons :

```
déclaration du concept
-> facettes et champs
-> projection de persistance
-> déclaration de l'action
-> accès à la collection
-> sélection de l'objet
-> exécution de l'action
-> mise à jour de l'état
-> traçabilité et publication
```

Listing 7.1 – Chaîne conceptuelle complète, de la déclaration à l'exécution

7.2 Le scénario métier

Le système doit exécuter périodiquement les actions en attente. Chaque élément d'action planifiée possède une date de prochaine exécution, un état d'exécution et une priorité. Le gestionnaire de planification consulte la file des actions en attente, choisit un élément, appelle son action `Execute`, puis marque cet élément comme exécuté.

Ce scénario restreint met néanmoins en jeu plusieurs concepts importants : modélisation d'entité, modélisation d'action, collections du runtime, comportement lié à l'objet et traçabilité.

7.3 Déclarer l'élément planifié

L'élément planifié est déclaré comme une entité. Ses champs conceptuels restent simples : une date, un booléen et une priorité.

```yaml
concepts:
  - Entity:
      facets:
        - main:
            type: hypergraph
            fields:
              - NextExecution: DateTime
              - Executed: Boolean
              - Priority: Integer
```

Listing 7.2 – Champs d'un élément planifié en YAML logiCells

Il ne s'agit déjà plus d'une simple définition de table. Les champs appartiennent à une facette conceptuelle et peuvent, de ce fait, être reliés à des vues, actions, filtres et projections de persistance.

7.4 Ajouter la projection de persistance

La facette de base de données associe l'élément conceptuel aux champs persistants. Elle relie également les champs d'identité à des chemins de rôles tels que `Id`, `FormalName` et `ClassId`.

```yaml
concepts:
  - Entity:
      facets:
        - ScheduleActionItem:
            type: database
            map: ScheduleActionItem
            fields:
              - IdScheduleActionItem:
                  type: String
                  generatorName: Lgc.Common.ScheduleActionItem
                  rolePath: Id
              - FormalNameScheduleActionItem:
                  type: String
                  rolePath: FormalName
              - ClassIdScheduleActionItem:
                  type: String
                  defaultValue: lgc#schedule_action_item
                  rolePath: ClassId
              - NextExecution: DateTime
              - Executed: Boolean
              - Priority: Integer
```

Listing 7.3 – Facette de base de données en YAML logiCells

Le runtime peut désormais interpréter les champs conceptuels et leur représentation persistante.

7.5 Déclarer l'action au niveau de l'objet

L'élément planifié expose une action `Execute`. Celle-ci renvoie un résultat booléen et peut être publiée comme capacité du runtime.

```yaml
action:
  name: Execute
  class: ObjectItemScriptableObjectElementAction
  type: ObjectItem
  caption: "#Execute"
  published: true
  params:
```

```
    - Result:
        dataType: Boolean
        paramType: ptReturn
  scripts:
    - script:
        clientScript: false
        serverScript: true
        language: functionalP
        classes: Desktop
        cdata: |
          Self.Executed := true;
          Self.PostData();
          Result := true;
```

Listing 7.4 – Action Execute en YAML logiCells

L'action est déclarée dans le modèle ; son exécution relève du mécanisme d'actions du runtime.

7.6 Déclarer le gestionnaire de planification

Le gestionnaire de planification est lui aussi déclaré comme entité. Il expose une action de niveau classe, nommée `ScheduleAction`. Cette action peut récupérer une file, ouvrir un tampon, sélectionner un élément, l'exécuter, puis enregistrer son nouvel état.

```
action:
  name: ScheduleAction
  class: ClassItemScriptableObjectElementAction
  type: ClassItem
  published: true
  params:
    - Result:
        dataType: Boolean
        paramType: ptReturn
  scripts:
    - script:
        clientScript: false
        serverScript: true
        language: functionalP
        classes: Desktop
        cdata: |
      ScheduleActionsCI := Self.ObjectsContainer.ClassItemByClassId('lgc#schedule_actions');
      ScheduleActionsProvider := ScheduleActionsCI.GenerateCollectionProvider('ActionQueue');
      Buffer.Open();
      if not Buffer.Eof() then begin
        lActionItemOI := lScheduleActionsOI.ScheduleActionItem;
        lSuccess := lActionItemOI.Execute();
        lActionItemOI.Executed := true;
        lActionItemOI.PostData();
      end;
```

Listing 7.5 – Ébauche de l'action du gestionnaire de planification en YAML

C'est à ce point que la modélisation déclarative rejoint le comportement exécutable : le modèle déclare l'action, et le runtime lui donne accès à son environnement conceptuel.

7.7 Ce que démontre ce scénario

Cet exemple montre que les déclarations logiCells peuvent couvrir une chaîne d'exécution complète. Le modèle décrit l'élément conceptuel, ses champs, son identité, sa persistance, son action exécutable et la capacité de gestion qui coordonne l'exécution.

Il montre aussi pourquoi le runtime doit rester centré sur les concepts : le planificateur ne traite pas des enregistrements anonymes, mais des éléments d'objet conceptuels dont les rôles, les actions et le comportement de persistance sont déclarés.

7.8 Pour aller plus loin

Les chapitres suivants généralisent les notions introduites ici. Nous distinguerons les systèmes centrés sur le code de ceux centrés sur les concepts, expliquerons pourquoi les applications peuvent être considérées comme des modèles conceptuels exécutables et montrerons pourquoi les concepts doivent être définis avant les classes, tables, routes et écrans.

Chapitre 8

Des systèmes centrés sur le code aux systèmes centrés sur les concepts

Un système centré sur le code part des artefacts d'implémentation : classes, méthodes, tables, contrôleurs, composants et services. Un système centré sur les concepts part des notions du domaine et cherche à déterminer leurs relations, leurs contraintes, leur exécution, leur observation et leurs projections. logiCells relève de cette seconde approche.

Cela ne signifie pas que le code disparaît, mais qu'il n'est plus le seul endroit où se trouve le sens de l'application. Le code se rattache à des structures conceptuelles déclarées.

8.1 Le réflexe de commencer par le code

Lorsqu'un développeur entend « action planifiée », son premier réflexe peut être de créer une table, une classe, un service de traitement en arrière-plan et une méthode. Cette démarche est compréhensible, mais elle peut masquer la structure conceptuelle plus générale. L'action planifiée est-elle une entité ? Possède-t-elle une identité conceptuelle ? Peut-on la présenter dans une vue ? L'exécuter au moyen d'une action publiée ? Tracer son état ? Permettre à un autre agent de raisonner à son sujet ?

logiCells conduit à poser ces questions avant de ramener le concept à des détails de réalisation.

8.2 Où se trouve le sens métier ?

Dans une réalisation centrée sur le code, le sens métier se répartit entre de nombreux fichiers. La table connaît les champs ; la classe, les propriétés ; le service de traitement, l'exécution ; l'interface, la présentation ; l'API, la structure des messages ; la documentation, l'intention. Aucune structure n'exprime à elle seule la signification complète.

Dans logiCells, le modèle rapproche ces dimensions. Une même déclaration peut réunir facettes conceptuelles, facettes de persistance, chemins de rôles, actions et vues.

```
Centré sur le code :
  table + classe + worker + route + formulaire + documentation

Centré sur les concepts :
  concept + facettes + champs + rôles + action + vue + publication
```

Listing 8.1 – Décomposition centrée sur le code et modèle centré sur les concepts

8.3 Des objets aux concepts

Les objets sont des structures utiles à l'exécution, mais un concept est plus large qu'un objet. Un objet représente généralement une instance dotée de champs et de méthodes. Un concept peut aussi porter des contraintes, des relations, des facettes, des correspondances de rôles, des événements et plusieurs projections.

Le modèle `UserContact` l'illustre : il ne s'agit pas seulement d'un objet reliant un utilisateur à un contact. Ses champs associent l'identité et les noms formels aux deux rôles, et le modèle peut être relié à des générateurs, à des facettes et à la persistance.

```yaml
fields:
  - IdUser:
      type: String
      rolePath: User.Id
      nullable: true
  - FormalNameUser:
      type: String
      rolePath: User.FormalName
      nullable: true
  - IdContact:
      type: String
      rolePath: Contact.Id
      nullable: true
  - FormalNameContact:
      type: String
      rolePath: Contact.FormalName
      nullable: true
```

Listing 8.2 – Association des rôles de UserContact dans le YAML logiCells

8.4 Des tables à la mémoire conceptuelle

Une table stocke des enregistrements ; la mémoire conceptuelle préserve leur signification. logiCells peut toujours utiliser des facettes de base de données, des correspondances et des champs, mais il convient de lire ces structures comme des projections des déclarations conceptuelles.

Cette différence compte lors des évolutions. Si la table fait autorité, toute modification est d'abord une modification du stockage. Si le concept fait autorité, le stockage s'adapte afin de préserver et d'exposer son sens.

8.5 Des routes aux capacités

Une route est un point d'entrée technique. Une capacité exprime une intention conceptuelle qui peut être publiée par un ou plusieurs canaux. Dans logiCells, une action telle que `Execute` ou `ScheduleAction` doit d'abord être comprise comme une capacité. Son invocation depuis une interface, un service, un script ou un agent relève ensuite de la publication.

8.6 Des écrans aux vues

Un écran est une interface utilisateur concrète. Une vue projette le modèle conceptuel. Cette distinction importe, car une même vue conceptuelle peut être publiée différemment selon le contexte : formulaire, liste, tableau de bord, composant Web ou représentation de service.

8.7 Conséquences architecturales

Une architecture centrée sur les concepts modifie l'ordre de conception. On commence par nommer le concept, son identité, ses rôles, ses contraintes, ses actions et ses projections. On

décide ensuite seulement comment le stocker, l'afficher et l'exposer.

Il ne s'agit pas d'une architecture moins technique, mais d'une discipline technique organisée autour d'un noyau conceptuel explicite.

Chapitre 9

Les applications comme modèles conceptuels exécutables

Une application logiCells n'est pas uniquement un programme qui manipule des données : c'est un modèle conceptuel exécutable. Le modèle contient des déclarations que le runtime sait interpréter : concepts, facettes, champs, contraintes, collections, actions, événements et vues.

Le terme *exécutable* est essentiel. Un modèle est exécutable lorsqu'il peut produire des objets du runtime, des champs, des collections, des appels d'action, des événements, des vues et des capacités de service. Il n'est pas seulement un artefact d'analyse : il participe au fonctionnement effectif du système.

9.1 De l'application-programme à l'application-modèle

Dans une architecture traditionnelle, l'application est souvent perçue comme un ensemble de code organisé en couches. Dans logiCells, elle peut être comprise comme un modèle conceptuel interprété par un runtime. Le code demeure présent, notamment dans les scripts et les gestionnaires, mais la structure qui l'entoure est déclarée.

Une application de planification d'actions, par exemple, ne se résume pas à une boucle de traitement : elle rassemble plusieurs concepts, dont l'élément planifié, la file d'attente, le gestionnaire, l'action, son état d'exécution et sa priorité.

9.2 Qu'est-ce qui rend un modèle exécutable ?

Un modèle devient exécutable lorsque le runtime peut en déduire les réponses à des questions opérationnelles :

— Quel est l'identifiant du concept ?
— Comment son identité est-elle générée ?
— Quels champs existent et quels sont leurs types ?
— Quels champs correspondent à des rôles conceptuels ?
— Quelle projection de persistance faut-il utiliser ?
— Quelles actions sont disponibles ?
— Quelles vues ou collections peuvent être publiées ?

Le modèle MODEL déclaratif suivant fournit déjà plusieurs de ces réponses.

```
class:
  name: ScheduleActionItem
  classId: lgc#schedule_action_item
  currentGeneratorName: Lgc.Common.ScheduleActionItem
  type: Entity
  generators:
    - class: GUIDStringGeneratorInfo
```

```yaml
      generatorName: Lgc.Common.ScheduleActionItem
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
```

Listing 9.1 – Informations du modèle exécutable en YAML logiCells

9.3 La frontière de l'application

La frontière applicative ne se limite pas aux sources du programme : elle est aussi conceptuelle. Elle détermine les concepts qui appartiennent à l'application, les capacités publiées, les vues disponibles et les systèmes externes avec lesquels elle interagit.

La frontière du planificateur peut inclure plusieurs éléments : `ScheduleActionItem` et `ScheduleActions`, puis `ActionSchedulerManager` et les collections qui sélectionnent les tâches en attente. À l'extérieur, elle peut n'exposer que quelques actions, tout en conservant les scripts propres à l'implémentation en interne.

9.4 Les actions font partie du modèle

Les actions ne sont pas des ajouts tardifs intervenant une fois les données structurées : elles appartiennent au modèle, car elles expriment ce que l'application est capable de faire. Un modèle conceptuel exécutable doit donc déclarer les actions au voisinage des concepts sur lesquels elles opèrent.

9.5 Les vues sont des projections facultatives

Une interface utilisateur peut projeter le même modèle conceptuel. L'architecture View de logiCells n'est cependant pas indispensable pour rendre ce modèle exécutable : une interface en ligne de commande, une API, un client MVP spécifique ou tout autre adaptateur peut s'appuyer sur les mêmes concepts et actions déclarés. Lorsqu'un client riche avec cache est nécessaire, le guide complémentaire *Guide des vues pour client riche logiCells* définit le contrat canonique `*.view.xml`, ainsi que le cycle de vie des DataSet et des événements.

La règle importante est qu'une vue n'est jamais le concept lui-même ; elle en constitue une projection d'interaction facultative.

9.6 Pourquoi l'exécutabilité est-elle importante ?

L'exécutabilité rend le modèle testable, inspectable et gouvernable. Un diagramme passif peut devenir obsolète sans effet immédiat. Un modèle exécutable est, lui, continuellement confronté au comportement du runtime. Une action, un champ, une vue ou une collection incorrects entraînent une incohérence qui peut être constatée à l'exécution.

Chapitre 10

Les concepts avant les classes, les tables et les routes

logiCells invite à respecter une discipline simple : nommer et structurer le concept avant de choisir la classe, la table, la route ou l'écran. Il ne s'agit pas de nier l'importance des artefacts techniques, mais d'éviter qu'ils deviennent la première et l'unique source de signification.

10.1 Le réflexe technique

À la réception d'un besoin, la conception technique démarre souvent immédiatement : créer une classe, une table, un point de terminaison, un formulaire. Ce réflexe peut être efficace pour une fonctionnalité limitée, mais il risque de compromettre la cohérence conceptuelle à long terme.

Par exemple, le besoin « associer un utilisateur à un contact » pourrait se traduire directement par une table de jointure. Dans logiCells, il s'agit d'abord d'un concept ou d'une relation : `UserContact`. Ce n'est qu'ensuite que l'on précise comment cet élément sera généré, persisté, présenté et utilisé.

10.2 L'ordre conceptuel

L'ordre de conception privilégié est le suivant :

```
concept
-> identity
-> roles and relations
-> constraints
-> facets
-> actions
-> collections and views
-> storage and publication
```

Listing 10.1 – Ordre de la conception conceptuelle

Cet ordre permet de garder la structure métier au premier plan.

10.3 Les concepts avant les classes

La déclaration YAML `class:` est nécessaire, mais la question conceptuelle vient d'abord : que cherche-t-on à modéliser ? Dans l'exemple suivant, `UserContact` représente une entité relationnelle entre un utilisateur et un contact. La classe qui en assure la prise en charge technique vient en second.

```yaml
class:
  name: UserContact
  classId: lgc#user_contact
  type: Entity
  implementation:
    className: CustomClassItem
    objectClassName: CustomObjectItem
    objectName: UserContact
```

Listing 10.2 – Le concept UserContact exprimé en YAML logiCells

La projection YAML distingue volontairement le nom conceptuel des éléments qui assurent son implémentation.

10.4 Les concepts avant les tables

Une table de base de données peut assurer la persistance de `UserContact`, mais elle n'en définit pas toute la signification. Les chemins de rôle montrent que les champs renvoient à deux participants conceptuels.

```yaml
fields:
  - IdUser:
      type: String
      rolePath: User.Id
  - ClassIdUser:
      type: String
      rolePath: User.ClassId
  - IdContact:
      type: String
      rolePath: Contact.Id
  - ClassIdContact:
      type: String
      rolePath: Contact.ClassId
```

Listing 10.3 – Projections de rôles dans la syntaxe YAML logiCells

La structure de persistance devient ainsi une projection de la relation conceptuelle.

10.5 Les concepts avant les routes

Une route expose une capacité au moyen d'un protocole réseau. Elle ne doit pas définir cette capacité à elle seule. Si le modèle déclare `Execute` ou `ScheduleAction`, ces noms expriment des intentions conceptuelles. Une route pourra ensuite les exposer, mais elle ne constituera qu'une forme de publication parmi d'autres.

10.6 Les concepts avant les écrans

Un écran présente un concept à l'utilisateur : il ne doit pas en devenir la définition. Dans logiCells, les déclarations de vues doivent être comprises comme des projections de structures conceptuelles. Le même concept peut ainsi être présenté différemment selon les contextes, sans duplication de sa signification.

10.7 Pourquoi cette discipline est-elle essentielle ?

Les systèmes métier vivent longtemps. Les tables sont restructurées, les écrans repensés, les services remplacés et les implémentations évoluent. Lorsque le concept conserve une identité explicite et stable, le système peut évoluer sans perdre sa signification.

Chapitre 11

Métadonnées, modèles et sémantique du runtime

Les métadonnées sont souvent considérées comme des informations secondaires : libellés, options de configuration, indications de schéma ou intitulés d'interface. Dans logiCells, leur rôle est plus fondamental. Elles décrivent des structures que le runtime peut interpréter et exécuter. C'est pourquoi on parle de *métadonnées exécutables*.

Le contrat des structures déclaratives rend ce principe explicite. Il définit les éléments, les attributs, les types de fichiers et les mécanismes de résolution des règles. La plateforme sait quels enfants sont autorisés sous chaque règle parente, quels attributs sont obligatoires et comment interpréter une déclaration.

11.1 Les métadonnées ne sont pas décoratives

Un attribut de modèle tel que `classId` n'est pas un simple libellé : il identifie une classe conceptuelle du runtime. Un `rolePath` établit le lien entre un champ et un rôle conceptuel. Le drapeau `published` d'une action intervient dans la visibilité de la capacité correspondante au sein de l'environnement d'exécution.

Les métadonnées logiCells doivent donc être lues comme des informations porteuses de sens pour le runtime.

11.2 Structure déclarative

Un modèle déclaratif obéit à des règles. À sa racine, un fichier de modèle peut déclarer une `class`. Celle-ci peut contenir des sections telles que `generators`, `object`, `concepts`, `collections`, `events`, `actions` et `views`. Un concept peut contenir des `constraints` et des `facets` ; une facette peut à son tour contenir des champs.

```yaml
class:
  generators: []
  object: {}
  concepts:
    - Entity:
        constraints: []
        facets: []
  collections: []
  events: []
  actions: []
  views: []
```

Listing 11.1 – Structure YAML logiCells équivalente

L'outillage déclaratif peut immédiatement exploiter cette structure au niveau MODEL pour valider la source, y naviguer et la modifier. La matérialisation de l'application intervient ultérieurement, lorsque les déclarations sont assemblées en vue de leur exécution.

11.3 Trois autorités pour trois questions

Avant de choisir un outil, le développeur gagne à déterminer la question à laquelle il souhaite répondre :

MODEL Que déclare physiquement ce fichier YAML ? Le fichier source et sa grammaire déclarative exécutable font autorité.

APPLICATION Quelles définitions sont actives une fois assemblés les packages, l'héritage, les clones et les redéfinitions ? Les métadonnées applicatives résolues font autorité.

SEMANTIC Quelle relation conceptuelle, quel motif H-Logic ou quel graphe sémantique est valide ? La source H-Logic et le graphe sémantique compilé font autorité selon l'opération considérée.

Une modification au niveau MODEL n'exige pas d'initialiser l'application ou l'hypergraphe. Inversement, on ne peut répondre à une question concernant APPLICATION en relisant simplement un fichier isolé, puisque d'autres packages ou redéfinitions peuvent changer le résultat effectif.

11.4 Des métadonnées descriptives aux métadonnées exécutables

Les métadonnées deviennent exécutables dès lors que les composants du runtime peuvent agir à partir d'elles. Un champ peut devenir une propriété, une facette une projection, une collection un fournisseur de données, une action une capacité appelable, et une vue une structure de présentation.

C'est pourquoi une même déclaration peut être interprétée à plusieurs niveaux. Un `field` est une ligne dans un fichier de modèle, mais aussi une déclaration typée, une éventuelle colonne de base de données, la projection d'un rôle conceptuel et potentiellement un champ d'interface ou de service.

11.5 Sens du modèle et sens à l'exécution

Le sens du modèle correspond à ce que la déclaration exprime pour les personnes qui la lisent. Son sens à l'exécution correspond à ce que la plateforme peut effectivement en faire. logiCells relie ces deux dimensions en rendant explicites les éléments et les attributs déclaratifs.

Par exemple :

```
ClassIdScheduleActionItem:
  type: String
  defaultValue: lgc#schedule_action_item
  caption: ClassId
  rolePath: ClassId
```

Listing 11.2 – Métadonnées d'un champ en syntaxe YAML logiCells

Pour le lecteur, il s'agit d'un champ associé à l'identité. Pour le runtime, c'est un champ typé, doté d'une valeur par défaut et d'un chemin de rôle.

11.6 Métadonnées conceptuelles et métadonnées techniques

Les métadonnées conceptuelles expriment le domaine et la sémantique d'exécution : noms de concepts, chemins de rôle, contraintes, facettes, actions et vues. Les métadonnées techniques décrivent les éléments nécessaires à l'implémentation : noms de classes, classes de générateurs, classes de scripts, correspondances physiques et chemins d'accès.

Les deux catégories sont nécessaires. Dans cet ouvrage, nous privilégions les métadonnées conceptuelles et n'expliquons les éléments techniques que lorsqu'ils éclairent le comportement du runtime.

11.7 Source MODEL YAML et contrat déclaratif

Dans ce guide, YAML est la syntaxe canonique pour écrire les sources MODEL. La grammaire déclarative exécutable en définit le contrat. Les anciens modèles MODEL XML peuvent être acceptés aux frontières de migration lorsque la couche de compatibilité du runtime le permet, mais les nouveaux modules ne doivent pas produire en parallèle une représentation MODEL XML. Les vues VIEW XML relèvent d'un contrat de source visuelle distinct, décrit dans l'ouvrage complémentaire *Guide des vues pour client riche logiCells*.

11.8 Un modèle n'est pas un diagramme

Un diagramme peut aider à comprendre un modèle sans pour autant définir une structure exploitable par le runtime. Un modèle logiCells doit contenir assez d'informations formelles pour être interprété : éléments, attributs, champs obligatoires, enfants autorisés et déclarations exécutables.

La structure déclarative est donc essentielle : elle fournit au runtime le contrat nécessaire pour lire les modèles.

Chapitre 12

La grammaire déclarative exécutable

Le langage de rédaction YAML ne repose pas sur de simples conventions informelles. Le framework actuel enregistre des structures déclaratives exécutables par type de fichier. Chaque structure précise les relations parent-enfant autorisées, le comportement des listes, la correspondance des noms d'affichage, les attributs obligatoires et les valeurs par défaut. L'outillage et le runtime interprètent les déclarations à partir de ces règles enregistrées ; leur représentation interne ne fait volontairement pas partie du contrat présenté dans ce guide.

12.1 Familles de fichiers déclaratifs prises en charge

La grammaire actuelle distingue les fichiers d'application, de package, de manifeste, de modèle, de conteneur multimodèle, de vue, de service, d'IA, de table de traduction et de déploiement. Un fichier `.models` n'est qu'un conteneur physique : les classes qu'il renferme utilisent exactement la même grammaire de classe MODEL qu'une source `.model` autonome.

12.2 Encodage des éléments YAML nommés

Une règle peut exposer le nom d'un élément à partir de sa balise ou de l'un de ses attributs. C'est pourquoi la syntaxe YAML canonique emploie fréquemment une entrée de correspondance nommée plutôt qu'un champ `name:` à l'intérieur de l'objet.

```yaml
generators:
  - TaskManagement.Task:
      nameField: Title

concepts:
  - Entity:
      facets:
        - main:
            type: hypergraph
            fields:
              - Title: String
```

Listing 12.1 – Formes canoniques des noms d'affichage

Ici, les clés `TaskManagement.Task`, `Entity`, `main` et `Title` portent les attributs retenus par la grammaire (par exemple le nom d'un générateur, d'un concept, d'une facette ou d'un champ). Ne les remplacez pas par une forme générique `- name: ...`, sauf si la règle parente concernée autorise explicitement cette structure.

12.3 Contrat de la racine MODEL

Une classe MODEL exige actuellement `classId`, `name`, `className` et `type`. Les attributs de racine facultatifs comprennent `currentGeneratorName`, `redefined` et `namespace`. La même règle de classe est réutilisée dans les fichiers MODEL autonomes, les anciens emplacements associés aux vues et les conteneurs `.models`.

12.4 Actions canoniques

Sous `actions`, `action` et `actionGroup` sont des enfants polymorphes : leur balise explicite doit donc rester visible. Une action directe exige `name` et `type`. La grammaire actuelle reconnaît également, à titre facultatif, `class`, `path`, `caption`, `published`, ainsi que les attributs de relais, de vue de paramètres, de paramètres dynamiques, de mode et d'héritage.

```
actions:
  - action:
      name: Execute
      type: ObjectItem
      published: true
      params:
        - Result:
            dataType: Boolean
            paramType: ptReturn
      scripts:
        - script:
            clientScript: false
            serverScript: true
            language: functionalP
            cdata: |
              Self.Status := 'Executed';
              Result := true;
```

Listing 12.2 – Action directe canonique avec résultat typé et script

Les paramètres d'action exigent `dataType` et `paramType` ; `defaultValue` est facultatif. Les scripts d'action exigent `clientScript`, `serverScript` et `language` ; `classes` et `cdata` sont facultatifs dans la grammaire.

12.5 Les événements ne sont pas des déclarations d'action

Une entrée d'action liée à un événement constitue un relais ou une liaison vers une action. Sa grammaire diffère de celle d'une déclaration d'action. Les paramètres de redirection obéissent à une grammaire de liaison réduite :

```
name        target parameter name
eventParam  value forwarded from the source event
value       literal constant assigned by the redirect
path        value resolved through a path
```

Listing 12.3 – Sémantique des paramètres de redirection événement-action

`value` représente une valeur littérale, et non du code exécutable. Des attributs de script tels que `scripted` ou `language` n'appartiennent pas à cette grammaire de paramètres de redirection. Utilisez `eventParam` lorsque la valeur provient de l'événement et `value` lorsque la redirection fournit une constante. Ne recopiez pas la grammaire des paramètres d'une action directe (`dataType`, `paramType`, `defaultValue`) dans un relais d'événement.

12.6 Collections et chemins de rôle

Les familles de collections comprennent `collection`, `objectItemListCollection`, `storedProcedure`, `agregate`, `union` et `join`. Les règles des champs exposent à la fois des métadonnées physiques (nom, type et taille du champ) et des correspondances conceptuelles, notamment `rolePath`. Il importe de préserver cette distinction : une projection de collection ou de base de données peut changer alors que le chemin de rôle conserve le sens conceptuel.

12.7 Compatibilité et rédaction canonique

Le runtime conserve des chemins historiques de compatibilité, notamment pour les anciennes formes XML. Les nouvelles documentations et les fichiers YAML générés doivent suivre la grammaire déclarative enregistrée plutôt que reproduire des exemples anciens plus permissifs. Si un exemple entre en conflit avec les règles enregistrées, celles-ci font autorité pour la syntaxe de rédaction ; la compatibilité d'exécution est documentée séparément.

12.8 Noms publics et liaisons d'implémentation

Certains attributs de grammaire sélectionnent des liaisons vers le runtime, notamment `className` ou `class`. Ils font partie du contrat de métadonnées actuel, mais ce guide emploie des noms publics de registre ou de liaison et omet volontairement les préfixes propres au langage d'implémentation. Un identifiant d'implémentation du runtime n'est pas du vocabulaire métier.

Chapitre 13

Migrer les métadonnées XML existantes vers YAML

Une migration ne se résume pas à convertir un format texte en un autre. Son objectif est de préserver le sens à l'exécution des métadonnées XML historiques tout en adoptant la syntaxe YAML actuelle de logiCells et sa structure déclarative canonique.

13.1 Chaîne de migration

Une migration fiable repose sur un aller-retour contrôlé et sur un essai d'exécution :

```
historical XML
    -> migration tooling
    -> canonical YAML
    -> semantic comparison with historical source
    -> runtime load
    -> model/application tests
```

Listing 13.1 – Déroulement recommandé d'une migration

La validation doit privilégier les propriétés sémantiques qui influent sur le runtime, plutôt que les étapes internes de sérialisation. Le YAML canonique constitue le format cible de rédaction ; la vérification compare le comportement du modèle obtenu et la signification de ses métadonnées avec ceux de la source historique.

13.2 Comparer le sens, non la présentation

Une comparaison octet par octet des fichiers XML serait trop stricte : la grammaire canonique peut introduire des conteneurs ou normaliser la représentation. Il faut plutôt comparer les informations qui influent sur le runtime :

— les identifiants de classe et de package, en particulier `classId` ;
— les concepts, les facettes et les contraintes ;
— les noms, types et chemins de rôle des champs, ainsi que leurs propriétés de persistance ;
— les états et leurs index ;
— les collections, les en-têtes et les données d'objets intégrées ;
— les actions, les groupes d'actions, les paramètres et les scripts ;
— les événements et les vues ;
— les définitions de services et les métadonnées de publication.

13.3 Risques connus lors de la migration

13.3.1 Valeurs préfixées par un dièse

Encadrez les valeurs telles que `'#top'` par des apostrophes afin que YAML ne les interprète pas comme des commentaires.

13.3.2 Éléments compacts ou développés

Un élément compact tel que `- Count: Integer` encode à la fois un nom d'affichage et une valeur d'affichage. Lorsque XML2Yaml développe cet élément pour lui ajouter des propriétés, il doit conserver sa valeur d'affichage (pour un champ, son `type`).

13.3.3 Séquences polymorphes

Ne déduisez pas le type des éléments voisins lorsque la liste peut contenir plusieurs sortes d'éléments. Employez des clés explicites, telles que `action:` et `actionGroup:`.

13.3.4 Conteneurs historiques

La forme déclarative canonique peut comporter des conteneurs comme `classes`, `items`, `params` ou `actions`, même lorsqu'une source XML historique propose une écriture abrégée sans ces conteneurs. Les métadonnées nouvellement générées doivent privilégier la forme canonique.

13.3.5 Données intégrées aux collections

Les en-têtes et les lignes d'objets sont des métadonnées, non du YAML arbitraire. Préservez explicitement les noms de champs, les valeurs `dataType`, les tailles, les chemins de rôle et les valeurs des lignes.

13.4 Migrer les vues historiques vers les contrats d'interaction

Les anciennes sources `.form`, `.frame` et `.view` doivent évoluer vers une représentation contractuelle mixte, plutôt que vers une description d'écran entièrement en YAML :

```
meta/views/
  contracts/
    Common/Folder/
      Fields.view.xml
      List.view.xml
      List.models.yaml
      _.manifest.yaml
  models/
    Common/Folder/
      BusinessClass.model.yaml
      _.manifest.yaml
```

Listing 13.2 – Arborescence VIEW canonique après migration

L'arborescence `contracts` contient des contrats d'interaction indépendants du canal. Un contrat peut encore décrire la disposition visuelle des éléments, mais il peut aussi être exploité par un runtime agentique qui présente les informations et demande les valeurs manquantes sans afficher de formulaire. Les déclarations MODEL, de modèle de page ou de modèle de composant intégrées sont externalisées dans un fichier `*.models.yaml` placé à proximité, uniquement lorsqu'elles existent. Ne créez pas de fichier modèle compagnon vide pour un contrat dépourvu de contenu MODEL.

Chaque classe externalisée est référencée depuis le contrat XML au moyen de `source`, `class` et `classId`. Le fichier local `_.manifest.yaml` utilise `fileName` pour désigner le conteneur physique `*.models.yaml`. Les métadonnées de vue d'une classe métier, lorsqu'elles doivent être séparées du package métier, sont représentées par une redéfinition minimale dans `meta/views/models` ; il ne faut pas dupliquer toute la classe métier.

L'outil de migration doit aussi éviter de reproduire directement l'ancienne arborescence source sous `meta/views`. Ses enfants immédiats canoniques sont `contracts` et `models`.

13.5 Trois niveaux de validation

Une migration doit franchir trois contrôles de force croissante :

1. **Syntaxe** : le YAML peut être analysé et le XML est bien formé.
2. **Grammaire** : les clés et les éléments sont autorisés par la règle parente et le type de fichier concernés.
3. **Runtime** : les manifestes sont chargés, les modèles compilés, la complétion des métadonnées aboutit et l'application atteint son état d'initialisation attendu.

Un fichier YAML syntaxiquement valide peut rester sémantiquement incomplet. Le chargement par le runtime fait donc partie de la validation de migration ; ce n'est pas une simple finition facultative.

13.6 Ce qu'il ne faut pas migrer automatiquement

Toute anomalie historique ne justifie pas une nouvelle règle de grammaire. Lorsqu'un ancien fichier de test utilise une structure dont aucun modèle pris en charge n'a besoin, il vaut mieux corriger ou retirer cette source que d'introduire une ambiguïté permanente. La compatibilité doit être voulue, circonscrite et justifiée par des métadonnées réellement déployées.

13.7 Tests de non-régression par aller-retour

Pour les familles de modèles importantes en production, conservez des jeux de référence XML/YAML représentatifs et comparez leurs structures déclaratives ayant un effet sur le runtime. Ils doivent au minimum couvrir les champs compacts, les champs développés, les correspondances nommées, les groupes d'actions imbriqués, les états, les contraintes avec valeurs préfixées par un dièse, les données de collections et les définitions de services. Ces jeux de référence sécurisent les futures optimisations du codec, car les gains de performance peuvent être confrontés à une sémantique stable.

Troisième partie

Fondements de la modélisation conceptuelle

Chapitre 14

Concepts, instances et relations

Une application logiCells commence par les concepts. Un concept décrit une catégorie d'éléments existant dans le domaine de l'application ou dans la plateforme elle-même. Une instance est une occurrence concrète de ce concept. Une relation relie des concepts ou des instances et structure le modèle. Ensemble, concepts, instances et relations forment le vocabulaire fondamental de la programmation conceptuelle déclarative.

La distinction est simple, mais essentielle. Le modèle peut ensuite produire une classe, une table, un formulaire ou un point de terminaison. Ces éléments ne constituent pas le premier niveau de signification. Celui-ci est conceptuel : quelles catégories d'éléments existent, comment sont-elles identifiées et reliées, quelles contraintes leur sont applicables et quelles actions peut-on leur associer ?

14.1 Concepts

Un concept définit un type de signification. Dans les fichiers MODEL YAML actuels, une classe conceptuelle se déclare sous la racine `class:`. L'attribut `name` fournit le nom du modèle, `classId` son identifiant conceptuel stable, et `type` indique la catégorie générale de l'élément modélisé.

```yaml
facets:
  - main:
      type: hypergraph
      fields:
        - NextExecution: DateTime
        - Executed: Boolean
        - Priority: Integer
  - ScheduleActionItem:
      type: database
      map: ScheduleActionItem
      fields:
        - IdScheduleActionItem:
            type: String
            rolePath: Id
        - ClassIdScheduleActionItem:
            type: String
            defaultValue: lgc#schedule_action_item
            rolePath: ClassId
```

Listing 14.1 – Facettes dans la syntaxe YAML logiCells

Dans cet exemple, `ScheduleActionItem` est déclaré comme un concept de type entité. La contrainte indique qu'il implique le concept plus général `lgc#entity` de la plateforme. Le modèle n'introduit donc pas simplement un objet technique : il situe l'objet dans une hiérarchie conceptuelle.

```yaml
class:
```

```yaml
  name: ScheduleActionItem
  classId: lgc#schedule_action_item
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
```

Listing 14.2 – Déclaration d'un concept en syntaxe YAML logiCells

14.2 Instances

Une instance est un objet concret conforme à un concept. Le modèle peut déclarer comment ces objets sont représentés et générés. En YAML, la déclaration `object:` indique la représentation d'objet du runtime associée à la classe.

```yaml
object:
  className: CustomObjectItem
  objectName: ScheduleActionItem
```

Listing 14.3 – Représentation d'objet en syntaxe YAML logiCells

Sur le plan conceptuel, cette déclaration signifie que le modèle dispose d'une représentation au niveau des instances. Elle ne doit pas être présentée comme l'abstraction principale : elle relève de l'implémentation. L'idée publique est qu'un concept peut posséder des instances à l'exécution et que celles-ci peuvent contenir des champs, participer à des relations, être persistées, présentées et faire l'objet d'actions.

14.3 Identifiants conceptuels

Un concept a besoin d'une identité stable. Dans les exemples, `lbs#schedule_action_item`, `lbs#user_contact` et `lgc#user` sont des identifiants conceptuels. Ce ne sont pas des libellés d'affichage : ces noms stables permettent au runtime de reconnaître le type conceptuel.

Ils diffèrent du nom d'une classe ou d'une table. Le nom d'une classe peut changer lors d'une refactorisation, celui d'une table à la suite d'une évolution du schéma de persistance. L'identifiant conceptuel devrait, lui, rester stable, car il contribue à l'identité sémantique du modèle.

14.4 Facettes

Un concept peut comporter plusieurs facettes. Chaque facette en constitue une projection destinée à un usage particulier du runtime. Les modèles YAML actuels comportent fréquemment une facette `hypergraph` et une facette `database`. La première exprime des champs conceptuels ou propres au runtime ; la seconde établit une correspondance entre les informations conceptuelles et les enregistrements persistants.

L'essentiel n'est pas que logiCells propose plusieurs formats de stockage, mais qu'un même concept puisse présenter plusieurs projections sans perdre son identité. Les facettes hypergraphe et base de données offrent deux vues distinctes d'un même objet conceptuel.

14.5 Relations

Une relation relie des concepts ou leurs instances. Elle peut prendre la forme d'un objet conceptuel explicite ou apparaître au moyen de chemins de rôle dans une facette. Le modèle

`UserContact` illustre bien ce second cas : il relie un utilisateur et un contact au moyen de champs dont les `rolePath` désignent `User.Id`, `User.ClassId`, `Contact.Id` et `Contact.ClassId`.

```yaml
fields:
  - IdUser:
      type: String
      rolePath: User.Id
      nullable: true
  - ClassIdUser:
      type: String
      defaultValue: lgc#user
      rolePath: User.ClassId
      nullable: true
  - IdContact:
      type: String
      rolePath: Contact.Id
      nullable: true
  - ClassIdContact:
      type: String
      defaultValue: lgc#contact
      rolePath: Contact.ClassId
      nullable: true
```

Listing 14.4 – Une même relation en syntaxe YAML logiCells

Une lecture orientée base de données y verrait des champs analogues à des clés étrangères. Une lecture conceptuelle y distingue des rôles : l'objet porte un rôle `User` et un rôle `Contact`. La facette de base de données ne représente qu'une possibilité de persister cette relation.

14.6 Contraintes

Les contraintes définissent ce qui doit être vrai d'un concept, d'un rôle ou d'un champ. Elles ne se limitent pas à des règles de validation à la périphérie du système : elles participent au sens du modèle. Dans les exemples simples qui précèdent, la contrainte d'implication place un concept sous un concept plus général de la plateforme. D'autres contraintes peuvent exprimer des valeurs par défaut, des valeurs fixes, la non-nullité, des domaines de rôle et la validité dans un contexte donné.

Une contrainte doit être lue comme un engagement conceptuel : elle indique au runtime, aux outils et au lecteur le comportement attendu du modèle.

14.7 Concepts, instances et relations à l'exécution

Les concepts ne sont pas de simples éléments documentaires. Une fois interprétés par le runtime, ils permettent de créer des instances, de résoudre des collections, d'invoquer des actions, de publier des vues et d'exposer des capacités. L'exemple du planificateur illustre ce principe : `ScheduleActionItem` est un concept ; un élément planifié précis en est une instance. Le gestionnaire du planificateur utilise une collection pour retrouver les instances, invoque leur action `Execute` et met à jour leur état `Executed`.

C'est à ce stade que la programmation conceptuelle déclarative devient opérationnelle. Le modèle n'est pas un diagramme placé à côté de l'application : il participe à son fonctionnement.

Chapitre 15

Types, rôles et relations polyadiques

logiCells ne réduit pas les relations à des liens anonymes entre deux nœuds. Une relation peut avoir un type, des participants nommés, des contraintes propres à chaque participant et une identité distincte. C'est un principe essentiel de la programmation conceptuelle déclarative : de nombreux faits métier ne se résument pas à des propriétés simples. Ils associent plusieurs participants, dont chacun joue un rôle précis dans le fait exprimé.

Une action planifiée, par exemple, n'est pas simplement un objet muni d'une date. Elle peut associer la définition d'une action, une règle de planification, une file d'attente, un contexte d'exécution, un objet cible et un résultat. Décomposer cette structure en champs techniques indépendants fait perdre une partie de sa signification. La représenter comme une structure conceptuelle typée préserve l'intention du modèle.

15.1 Des champs aux rôles conceptuels

Dans les fichiers de modèles logiCells présentés dans ce livre, les participants d'une relation sont généralement représentés par des champs d'une facette conceptuelle, puis par des projections `rolePath` dans une facette de persistance. La notation déclarative MODEL n'introduit normalement pas d'élément de rôle distinct pour les participants des relations métier. Le rôle est plutôt porté par le nom du champ, son type et le chemin de rôle utilisé lors de la projection.

Un champ nommé `IdUser` indique seulement qu'un identifiant est stocké quelque part. Un champ conceptuel `User`, projeté par `User.Id` et `User.ClassId`, indique que la structure contient un participant jouant le rôle d'utilisateur. La différence est fondamentale : le rôle conceptuel est défini par le modèle, tandis que les champs de base de données n'en constituent qu'une projection.

```
class:
  name: ScheduleActionItem
  classId: lgc#schedule_action_item
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
        facets:
          - main:
              type: hypergraph
              fields:
                - Action: lgc#action
                - Queue: lgc#queue
                - NextExecution: DateTime
                - Executed: Boolean
          - ScheduleActionItem:
              type: database
              map: lgcScheduleActionItem
              fields:
```

```
                - IdAction:
                    type: String
                    rolePath: Action.Id
                - ClassIdAction:
                    type: String
                    rolePath: Action.ClassId
                    defaultValue: lgc#action
                - IdQueue:
                    type: String
                    rolePath: Queue.Id
                - ClassIdQueue:
                    type: String
                    rolePath: Queue.ClassId
                    defaultValue: lgc#queue
                - NextExecution: DateTime
                - Executed: Boolean
```

Listing 15.1 – Participants typés en notation MODEL YAML

YAML est la syntaxe déclarative native employée pour décrire les modèles métier logiCells. Sa structure respecte les contrats de métadonnées déclaratives implémentés par le runtime. XML offre une autre représentation des mêmes intentions de modélisation.

15.2 Les noms de rôles appartiennent au modèle

Le nom d'un participant n'est pas cosmétique : il contribue à la sémantique du modèle. Deux participants de même type mais de significations différentes ne doivent pas être fusionnés en une référence anonyme. Dans un modèle de contacts, un utilisateur peut être propriétaire d'un contact, son créateur, la dernière personne à l'avoir modifié ou la personne représentée par le contact. Toutes ces références peuvent viser des concepts d'utilisateur, sans être pour autant interchangeables.

Les noms de rôles renforcent également la stabilité des vues générées, des services et des explications produites par l'IA. Un modèle qui déclare un participant `Queue` peut expliquer pourquoi une action attend. Le seul stockage d'un identifiant numérique de file d'attente n'apporte pas la même interprétation conceptuelle sans documentation externe.

15.3 Relations binaires

Une relation binaire relie deux participants. Elle convient à des cas courants : un utilisateur possède un contact, une action appartient à une file d'attente, une catégorie classe un élément ou un élément planifié déclenche une action.

Dans les modèles logiCells, une telle relation est normalement décrite comme une classe conceptuelle ou un modèle de relation. Les participants sont déclarés dans les champs d'une facette conceptuelle, puis projetés dans la facette de base de données au moyen de `rolePath`.

```
class:
  name: BelongsToQueue
  classId: lgc#belongs_to_queue
  className: CustomClassItem
  type: Relation
  concepts:
    - Relation:
        constraints:
          - implies: lgc#relation
        facets:
          - main:
              type: hypergraph
              fields:
                - Item: lgc#schedule_action_item
                - Queue: lgc#queue
          - BelongsToQueue:
```

```
            type: database
            map: lgcBelongsToQueue
            fields:
              - IdBelongsToQueue:
                  type: String
                  rolePath: Id
              - ClassIdBelongsToQueue:
                  type: String
                  rolePath: ClassId
                  defaultValue: lgc#belongs_to_queue
              - IdItem:
                  type: String
                  rolePath: Item.Id
              - ClassIdItem:
                  type: String
                  rolePath: Item.ClassId
                  defaultValue: lgc#schedule_action_item
              - IdQueue:
                  type: String
                  rolePath: Queue.Id
              - ClassIdQueue:
                  type: String
                  rolePath: Queue.ClassId
                  defaultValue: lgc#queue
```

Listing 15.2 – Relation binaire en notation MODEL YAML

Une relation binaire reste un objet conceptuel. Des contraintes, métadonnées, règles de visibilité, projections de persistance et événements peuvent lui être associés.

15.4 Relations polyadiques

Une relation polyadique associe plus de deux participants. logiCells se distingue ainsi des modèles graphiques qui contraignent tous les faits à des arêtes binaires. Les faits métier comportent souvent trois, quatre participants ou davantage. Leur décomposition en chaînes de liens binaires peut être techniquement réalisable, mais masquer l'unité conceptuelle réelle.

Considérons l'exécution d'une action planifiée. Le fait pertinent peut associer l'élément planifié, la définition de l'action, la file d'attente, le contexte, l'instant d'exécution et le résultat. Il s'agit d'un seul événement métier, non d'une série de liens sans rapport.

```
class:
  name: ScheduledExecution
  classId: lgc#scheduled_execution
  className: CustomClassItem
  type: Relation
  concepts:
    - Relation:
        constraints:
          - implies: lgc#relation
        facets:
          - main:
              type: hypergraph
              fields:
                - Item: lgc#schedule_action_item
                - Action: lgc#action
                - Queue: lgc#queue
                - Context: lgc#execution_context
                - PlannedTime: DateTime
                - Result:
                    type: lgc#action_result
                    nullable: true
          - ScheduledExecution:
              type: database
              map: lgcScheduledExecution
              fields:
                - IdScheduledExecution:
                    type: String
                    rolePath: Id
```

```
                - ClassIdScheduledExecution:
                    type: String
                    rolePath: ClassId
                    defaultValue: lgc#scheduled_execution
                - IdItem:
                    type: String
                    rolePath: Item.Id
                - IdAction:
                    type: String
                    rolePath: Action.Id
                - IdQueue:
                    type: String
                    rolePath: Queue.Id
                - IdContext:
                    type: String
                    rolePath: Context.Id
                - PlannedTime: DateTime
                - IdResult:
                    type: String
                    rolePath: Result.Id
                    nullable: true
```

Listing 15.3 – Relation polyadique en notation MODEL YAML

Cette modélisation permet de nommer le fait d'exécution, de l'inspecter, de lui appliquer des contraintes, de l'interroger et de l'expliquer dans son ensemble.

15.5 Identité d'une relation

Une relation peut nécessiter sa propre identité, en particulier lorsqu'elle n'est pas seulement structurelle, mais aussi historique, auditable ou susceptible de déclencher des actions. L'affectation d'un contact, une exécution planifiée, une décision de validation ou une opération d'import-export peuvent devenir des faits assimilables à des entités, dotés d'horodatages, d'états, d'une provenance et d'événements.

C'est ici que la distinction entre relation et fait réifié devient importante. La relation décrit la structure ; le fait réifié donne une identité stable, dans le runtime, à une occurrence de cette structure.

```
class:
  name: ScheduledExecutionFact
  classId: lgc#scheduled_execution_fact
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: lgc#scheduled_execution
        facets:
          - main:
              type: hypergraph
              fields:
                - Item: lgc#schedule_action_item
                - Action: lgc#action
                - Queue: lgc#queue
                - PlannedTime: DateTime
                - ExecutedAt: DateTime
                - Status: String
```

Listing 15.4 – Occurrence de relation réifiée en notation MODEL YAML

15.6 Correspondances polyadiques

Une correspondance polyadique décrit comment les participants d'une structure conceptuelle sont alignés sur ceux d'une autre structure, ou sur ses paramètres. Elle intervient lors de la

transformation, de la spécialisation, de la projection ou de l'exécution d'un modèle. Un élément planifié peut ainsi être projeté en appel de service, ou une déclaration d'action en commande exécutable du runtime.

L'objectif n'est pas d'ajouter une couche de mapping technique, mais de rendre cet alignement explicite. Si le participant source `PlannedTime` devient le paramètre cible `StartAt`, le modèle doit pouvoir exprimer cette correspondance.

```
polyadicMap:
  name: scheduleItemToActionCall
  source: lgc#scheduled_execution
  target: lgc#action_call
  mapping:
    Item: TargetObject
    Action: Operation
    PlannedTime: StartAt
    Context: ExecutionContext
```

Listing 15.5 – Alignement des participants en notation MODEL YAML

15.6.1 Correspondances explicites, dérivées et contraintes non applicables

Une correspondance établit que des rôles appartenant à deux structures conceptuelles concordent. Si le modèle déclare explicitement un mapping, celui-ci est prioritaire. Si logiCells en déduit un, il ne peut utiliser que des correspondances de rôles effectivement compatibles entre les structures source et cible.

Par exemple, si une expression de contrainte expose les rôles `Amount` et `Limit`, elle peut être liée à un concept source qui expose des rôles compatibles de ces noms. Si la source ne présente que `Country`, aucune correspondance dérivée n'est pertinente. La contrainte doit alors être considérée comme non applicable, et non être associée par une supposition sur les positions.

```
explicit mapping exists
  -> use it

otherwise
  -> align compatible source and target roles

no compatible roles
  -> no valid polyadic map
  -> constraint is not applicable to that source
```

Listing 15.6 – Règle conceptuelle de dérivation des correspondances

Ne pas inventer de rôles artificiels pour faire correspondre une relation. Cette règle préserve la nature sémantique de la correspondance polyadique : elle décrit un alignement réel du modèle au lieu de masquer une incompatibilité.

15.7 Règles de conception

Modéliser une relation comme une structure conceptuelle lorsque le lien porte une signification métier. Nommer les participants à l'aide des champs de la facette hypergraphe. Dans les facettes de persistance, utiliser `rolePath` pour projeter les identités et identifiants de classe des participants. Définir un modèle propre pour une relation qui doit être interrogée, contrainte, auditée, affichée, publiée ou exploitée par une action. Ne pas créer d'élément de rôle XML artificiel pour les participants d'une relation si la grammaire réelle du modèle n'en prévoit pas. Dans les exemples de ce guide, les participants métier sont représentés par les champs, facettes, jointures, collections, contraintes et projections `rolePath`.

Chapitre 16

Identité, contexte et portée conceptuelle

Un modèle conceptuel a besoin d'identités stables. Sans identité, le runtime ne peut distinguer de manière fiable un concept, une instance, une occurrence de relation, une projection ou une vue contextuelle du même objet sous-jacent. logiCells place donc l'identité et le contexte au cœur de la programmation conceptuelle déclarative.

L'identité répond à la question « de quoi s'agit-il ? » ; le contexte, à « sous quelles conditions conceptuelles cet élément est-il valide, visible ou interprété ? » ; et la portée, à « dans quel périmètre une déclaration s'applique-t-elle ? ».

16.1 Identité conceptuelle

Un concept possède un identifiant conceptuel stable. Dans les fichiers MODEL YAML, celui-ci apparaît sous des propriétés telles que `classId`, `relationId` ou d'autres identifiants de modèle. Il ne faut pas le confondre avec une clé primaire de base de données : la clé désigne une ligne persistée ; l'identifiant conceptuel désigne une signification dans le modèle.

```yaml
class:
  name: UserContact
  classId: lgc#user_contact
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
```

Listing 16.1 – Identité conceptuelle en syntaxe YAML logiCells

L'identifiant `lbs#user_contact` donne à l'élément de modèle un nom conceptuel stable. Le runtime peut ensuite créer des instances, des vues, des actions et des projections persistantes à partir de la déclaration, mais l'identité conceptuelle reste le point d'ancrage.

16.2 Identité des instances

L'identité d'instance appartient aux occurrences concrètes. Deux contacts peuvent porter le même nom, partager un domaine de messagerie ou appartenir à la même catégorie : ils demeurent deux instances distinctes. Inversement, une même instance peut apparaître dans plusieurs vues, être exportée par des services, enrichie de facettes ou placée dans différents contextes.

Un modèle logiCells doit donc distinguer les libellés d'affichage de l'identité. Un libellé peut évoluer ; l'identité conceptuelle d'une instance doit être suffisamment stable pour la persistance, la traçabilité, la synchronisation et l'acheminement des événements.

```yaml
instance:
  id: contact-00042
  type: lgc#user_contact
  values:
    displayName: Marie Durand
    email: marie.durand@example.org
  context: production
```

Listing 16.2 – Identité d'instance en syntaxe YAML logiCells

16.3 Identité d'une relation et `this`

Certaines relations sont de simples liens structurels ; d'autres sont des faits dont il faut suivre l'évolution. Une occurrence de relation qui possède un historique, un état, une provenance ou une piste d'audit doit disposer de sa propre identité. Dans les explications fondées sur H-Logic, c'est souvent le rôle de `this`, qui désigne l'identité du fait ou de l'occurrence de relation elle-même.

Ainsi, l'exécution d'un élément planifié ne se réduit pas à un lien entre cet élément et une action. Elle peut porter un horodatage, un résultat, un message d'erreur, un opérateur et un nombre de nouvelles tentatives. L'occurrence de relation devient un fait du runtime.

```yaml
class:
  name: ScheduledExecutionFact
  classId: lgc#scheduled_execution_fact
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: lgc#scheduled_execution
        facets:
          - main:
              type: hypergraph
              fields:
                - Item: lgc#schedule_action_item
                - Action: lgc#action
                - Queue: lgc#queue
                - ExecutedAt: DateTime
                - Status: String
          - ScheduledExecutionFact:
              type: database
              fields:
                - IdItem:
                    type: String
                    rolePath: Item.Id
                - IdAction:
                    type: String
                    rolePath: Action.Id
                - IdQueue:
                    type: String
                    rolePath: Queue.Id
                - ExecutedAt: DateTime
                - Status: String
```

Listing 16.3 – Fait d'exécution réifié en notation MODEL YAML

16.4 Identité d'une entité et `me`

La notion de `me` intervient lorsqu'une déclaration est lue du point de vue de l'entité portée par la structure, ou de son support principal. Alors que `this` identifie le fait courant ou l'occurrence de relation, `me` peut désigner l'entité décrite ou projetée.

Cette distinction évite les ambiguïtés dans les déclarations qui combinent des propriétés d'entité et de relation. Par exemple, une relation d'affectation de contact peut avoir son propre

statut, tandis que le contact possède un nom et une adresse électronique. L'affectation et le contact n'ont pas la même identité.

16.5 Contexte

Un contexte constitue une dimension de validité et d'interprétation. Une déclaration peut être valide en production, en test, pour un client, pour un projet ou dans un contexte de sécurité. Le contexte n'est pas une simple étiquette : il peut modifier la visibilité, les contraintes, les actions disponibles et l'interprétation.

La grammaire MODEL actuelle ne définit pas d'objet générique `context:` à la racine. Les sémantiques contextuelles sont représentées explicitement dans les modèles et les règles. Dans H-Logic, un contexte est un rôle typé ordinaire :

```
1 define .:#Context(me)[label::#string];
2 define .:#ScheduledIn(item:#ScheduleActionItem, context:#Context);
3 #production:#Context;
```

Listing 16.4 – Contexte représenté par un rôle explicite

Un contexte comme `projectBeta` doit donc être modélisé comme une instance de concept ou une interface contextuelle, puis transmis explicitement dans les relations. Il ne doit pas être encapsulé dans un conteneur YAML non documenté ni dans un appel ambigu tel que `context(projectBeta)`.

16.6 Portée conceptuelle

La portée détermine le périmètre d'application d'une déclaration. Une contrainte peut s'appliquer à tous les éléments d'actions planifiées, uniquement à ceux de production ou à un package donné. Une vue peut être globale, propre à un rôle ou à un contexte. Une action peut être disponible sur toutes les instances d'un concept, ou seulement sur celles qui satisfont une condition.

Une portée explicite évite les généralisations involontaires. Une règle correcte pour des opérations d'export structuré ne l'est pas nécessairement pour tous les exports. De même, un workflow d'accueil peut être inadapté à des tâches d'arrière-plan périodiques.

16.7 Redéfinition et extension

Les modèles logiCells évoluent fréquemment : de nouvelles facettes, contraintes et actions peuvent s'y ajouter. Pour faire évoluer un modèle en sécurité, il faut distinguer l'extension de la redéfinition. L'extension apporte une signification supplémentaire tout en conservant l'identité du concept existant ; la redéfinition en modifie la signification et requiert donc une attention particulière.

Chapitre 17

Contraintes, valeurs par défaut et validité

Un modèle conceptuel déclaratif n'est pas seulement un vocabulaire. Il consigne également des contraintes et des politiques de valeurs que le runtime peut interpréter. Ce chapitre se limite délibérément aux formes présentes dans la grammaire MODEL exécutable actuelle. Les invariants métier sans attribut MODEL correspondant doivent être exprimés dans la couche des règles ou des actions, sans inventer de nouvelles clés YAML.

17.1 Contraintes sur les concepts

La grammaire MODEL actuelle accepte une collection `constraints` sous un concept. Les formes de contraintes enregistrées comprennent `implies`, `equivalent`, `and`, `or`, `xor`, `not` ainsi que la forme générique `constraint`. Les formes logiques nommées peuvent faire référence à un autre concept au moyen de `relatedTo`, ou porter une expression. La forme YAML compacte rend directement visible la cible associée.

```yaml
class:
  name: ScheduleActionItem
  classId: lgc#schedule_action_item
  className: CustomClassItem
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
```

Listing 17.1 – Contrainte d'implication canonique

Quand une contrainte repose sur une expression, la grammaire transporte également le langage de cette expression et, éventuellement, une correspondance polyadique. Sa sémantique exacte dépend du contrat du langage concerné : il ne faut pas déduire des exemples l'existence d'un nouveau mini-langage YAML.

17.2 Typage des participants et chemins de rôle

Pour un concept de type relation, les attentes envers les participants peuvent être exprimées par des champs typés dans la facette conceptuelle, puis reliées aux champs persistés à l'aide de `rolePath`. Le caractère nullable relève du contrat de champ et ne se confond pas avec la signification logique de la relation.

```yaml
class:
  name: ScheduledExecution
```

```
  classId: lgc#scheduled_execution
  className: CustomClassItem
  type: Relation
  concepts:
    - Relation:
        constraints:
          - implies: lgc#relation
        facets:
          - main:
              type: hypergraph
              fields:
                - Item: lgc#schedule_action_item
                - Action: lgc#action
                - Queue: lgc#queue
                - Context: lgc#execution_context
                - Result:
                    type: lgc#action_result
                    nullable: true
          - ScheduledExecution:
              type: database
              fields:
                - IdItem:
                    type: String
                    rolePath: Item.Id
                    nullable: false
                - IdAction:
                    type: String
                    rolePath: Action.Id
                    nullable: false
                - IdQueue:
                    type: String
                    rolePath: Queue.Id
                    nullable: false
                - IdContext:
                    type: String
                    rolePath: Context.Id
                    nullable: false
                - IdResult:
                    type: String
                    rolePath: Result.Id
                    nullable: true
```

Listing 17.2 – Participants typés et chemins de rôle persistants

Cette déclaration rend le contrat de représentation inspectable. Elle ne démontre pas, à elle seule, tous les invariants métier relatifs à une exécution planifiée. Les invariants temporels ou mettant en jeu plusieurs objets doivent être validés par le contrat H-Logic, d'action ou de processus approprié.

17.3 Valeurs par défaut

La grammaire actuelle des champs utilise `defaultValue`. Il s'agit d'une politique de valeur en cas d'absence d'un champ, et non d'une preuve que cette valeur est admissible dans tous les contextes métier.

```
concepts:
  - Entity:
      facets:
        - main:
            type: hypergraph
            fields:
              - Priority:
                  type: Integer
                  defaultValue: 0
              - Executed:
                  type: Boolean
                  defaultValue: false
```

Listing 17.3 – Valeurs par défaut en YAML MODEL canonique

Les paramètres d'action prennent également en charge `defaultValue`, mais leur grammaire diffère de celle des champs.

17.4 Caractère nullable

La grammaire des champs MODEL expose `nullable`. Utiliser cet attribut lorsque l'absence de valeur fait partie du contrat du champ.

```
fields:
  - Action:
      type: lgc#action
      nullable: false
  - Result:
      type: lgc#action_result
      nullable: true
```

Listing 17.4 – Champs nullables et non nullables

Ne pas remplacer les attributs documentés par des clés non prises en charge, telles que `required` ou `validWhen`, dans le YAML MODEL. Lorsqu'une condition de validité dépend de l'état d'exécution, du contexte, des autorisations ou d'autres objets, elle doit être exprimée dans la couche de règles ou d'actions responsable de l'invariant.

17.5 Contrats de paramètres d'action

Un paramètre d'action direct est déclaré sous `params`. Dans la grammaire actuelle, chaque paramètre requiert `dataType` et `paramType`, tandis que `defaultValue` reste facultatif. La direction et les modalités de retour sont donc définies explicitement par le contrat du paramètre, sans être déduites d'un indicateur générique `required`.

```
actions:
  - action:
      name: ExecuteScheduledAction
      type: ObjectItem
      params:
        - Context:
            dataType: String
            paramType: ptInput
        - Retry:
            dataType: Boolean
            paramType: ptInput
            defaultValue: false
        - Result:
            dataType: Boolean
            paramType: ptReturn
```

Listing 17.5 – Contrat canonique des paramètres d'action

Les valeurs admises pour `paramType` relèvent du contrat du runtime. La grammaire déclarative définit la structure ; l'implémentation du runtime ou de l'action définit la signification opérationnelle de chacune des valeurs.

17.6 Invariants métier fixes

Ce guide ne présente pas de construction MODEL générique `fixedValues` à la racine, car elle ne fait pas partie de la grammaire MODEL actuellement enregistrée. Une règle métier telle

que « cette famille d'actions sert toujours à exporter » doit donc s'appuyer sur un mécanisme réellement pris en charge : concept spécialisé, champ ou valeur par défaut déclarés lorsque cela convient, expression de contrainte, H-Logic, ou validation par une action ou un processus.

Cette distinction est intentionnelle : les exemples de documentation ne doivent pas faire passer pour canonique une clé YAML non prise en charge sous prétexte que le principe serait utile.

17.7 Une validité à plusieurs niveaux

La validation intervient à plusieurs niveaux, qui ne doivent pas être confondus :

Validité déclarative La source YAML respecte la grammaire enregistrée : sous-éléments autorisés, attributs obligatoires et formes structurelles valides.

Validité du modèle Les identifiants, chemins de rôle, références et contraintes propres au modèle sont suffisamment cohérents pour permettre sa résolution.

Validité sémantique Les propositions et requêtes H-Logic respectent le contrat sémantique applicable à l'opération.

Validité opérationnelle Une action ou un processus est autorisé dans l'état, le contexte et le périmètre de permissions courants.

Une validation syntaxique réussie n'établit que le premier niveau, sauf si l'outil invoqué exécute explicitement des contrôles supplémentaires. C'est pourquoi le guide distingue partout les autorités MODEL, APPLICATION et SEMANTIC, notamment dans les chapitres dédiés à l'outillage.

Chapitre 18

Graphes conceptuels et mémoire hypergraphique

logiCells repose sur un principe : la signification métier ne doit pas être éparpillée entre le code, les tables, les routes, les écrans et la documentation. Le développeur la décrit d'abord dans les sources MODEL. Après compilation sémantique de ces déclarations ou des sources H-Logic, le runtime peut les matérialiser dans une mémoire hypergraphique conceptuelle, où les concepts, relations, rôles et contraintes deviennent directement interrogeables et composables.

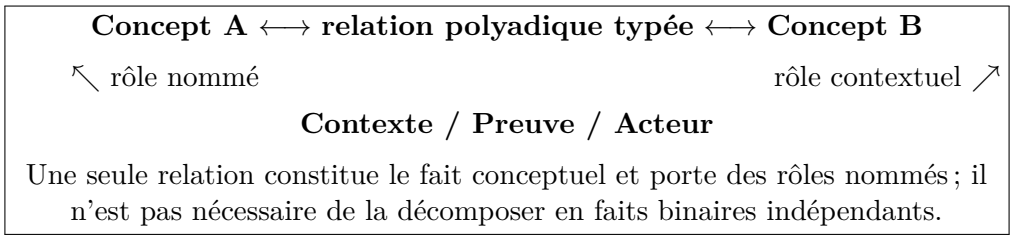


FIGURE 18.1 – Représentation d'un hypergraphe conceptuel : des concepts et des contextes participent à des relations typées par leurs rôles nommés.

Un graphe classique relie un nœud à un autre. Cette représentation convient à de nombreux parcours, mais peut être trop limitée pour exprimer directement une structure métier. Les faits métier sont rarement purement binaires. Une action planifiée peut faire intervenir un chemin d'action, un objet cible, une règle de déclenchement, un état, un utilisateur, une file d'attente et un contexte d'exécution. Une activité de workflow peut dépendre d'un objet de base, d'un état courant, de transitions et d'un canal de publication. L'hypergraphe conceptuel permet de représenter ces ensembles comme des relations typées et porteuses de rôles, sans les aplatir prématurément en tables ou références d'objets.

18.1 Pourquoi une mémoire hypergraphique ?

La mémoire hypergraphique ne remplace ni les fichiers sources, ni les métadonnées applicatives, ni la persistance, ni les mécanismes d'exécution. Elle constitue la représentation SEMANTIC du runtime lorsque les structures conceptuelles doivent être mises en correspondance, reliées, inférées ou interrogées. Le même projet s'observe ainsi à trois niveaux distincts : sources MODEL, métadonnées APPLICATION et hypergraphe SEMANTIC.

Cette distinction est importante, car le mot *hypergraphe* apparaît à deux endroits. Dans un modèle YAML, une facette de type `hypergraph` est une déclaration décrivant une projection conceptuelle. La modifier reste une opération sur le fichier source MODEL. L'hypergraphe sémantique physique est une structure compilée du runtime : il ne fait pas autorité pour l'édition

des sources.

Une fois compilées, les relations conceptuelles peuvent être explorées au moyen d'index, de la navigation, de motifs et d'opérations logiques. L'hypergraphe n'est donc pas une copie passive de la source : il constitue la mémoire structurelle du runtime sémantique, tandis que le fichier source reste l'autorité sur les déclarations MODEL.

18.2 Concepts, relations et hyperarêtes

Un concept désigne un type ou une structure dotée de signification dans le modèle : entité, relation, activité de workflow, action, objet métier ou capacité technique. Une relation relie plusieurs positions conceptuelles ; au-delà de deux, elle est naturellement polyadique. Au lieu de forcer ce lien dans des tables intermédiaires ou des objets artificiels, l'hypergraphe peut le conserver comme une seule structure typée, avec des rôles nommés.

Ce principe est particulièrement important en programmation déclarative. Une déclaration doit dire ce qui existe, quelles contraintes s'appliquent et comment l'élément participe au modèle. Elle ne doit pas déterminer prématurément si l'implémentation sera un champ de classe, une colonne, une table de jointure, un paramètre de service ou un champ d'interface. La mémoire hypergraphique offre à logiCells un substrat conceptuel neutre avant le choix de ces projections.

18.3 Exemple YAML : facettes hypergraphe et base de données

L'exemple simplifié suivant reprend la structure des fichiers de modèle YAML réels de logiCells. La classe expose une facette hypergraphe conceptuelle et une facette de projection vers la base de données. La première conserve les rôles conceptuels ; la seconde décrit les champs persistants.

```yaml
class:
  classId: lgc#activity
  name: Activity
  className: CustomClassItem
  currentGeneratorName: Common.CustomActivity
  type: Entity

  object:
    className: ObjectItemActivity
    objectName: Activity

  concepts:
    - Entity:
        constraints:
          - implies: CustomActivity
        facets:
          - main:
              type: hypergraph
              fields:
                - ActionPath: String
                - BaseObject: Object
          - Activity:
              type: database
              fields:
                - ActionPath:
                    size: 255
                    readOnly: false
                - BaseObject:
                    readOnly: true
                - FormalNameActivity:
                    size: 100
                    caption: '#FormalNameActivity'
                    rolePath: Activity.FormalName
                    nullable: true
```

Listing 18.1 – Classe YAML avec facettes hypergraphe et base de données

Cet exemple illustre une règle centrale de logiCells : un même élément de modèle peut porter plusieurs significations complémentaires. La facette `main` décrit la structure conceptuelle ; la facette `Activity`, une projection de base de données. Le champ `FormalNameActivity` n'est pas qu'une colonne brute : `rolePath` le relie à un chemin conceptuel. logiCells maintient ainsi le lien entre sens conceptuel et données opérationnelles.

18.4 Mémoire hypergraphique et exécution du runtime

La mémoire hypergraphique devient opérationnelle lorsque le runtime exploite les structures déclarées pour instancier des objets, résoudre des chemins, évaluer des relations, sélectionner des fournisseurs, publier des vues ou exécuter des actions. L'essentiel est que l'exécution ne part pas d'une classe ou d'une table isolée, mais d'un modèle structuré.

Un champ déclaré dans une facette hypergraphe participe à la forme conceptuelle de l'objet. Déclaré dans une facette de base de données, il participe à une projection de persistance. En présence d'un chemin de rôle, le runtime peut relier la valeur projetée à une relation ou à un chemin de propriété conceptuel. logiCells conserve ainsi le sens au niveau du modèle tout en répondant à des besoins pratiques : bases de données, formulaires, services et grilles de données.

18.5 Lien avec l'IA et la mémoire sémantique

L'hypergraphe conceptuel est aussi un point de rencontre naturel entre structures symboliques et représentations destinées à l'IA. La documentation des API présente les embeddings comme des annotations ou des projections tensorielles de nœuds, en distinguant le stockage conceptuel de la projection calculatoire. Cette différence est essentielle : un vecteur d'embedding facilite la recherche, le classement ou la comparaison, mais l'hypergraphe demeure la mémoire structurée qui explique la signification des nœuds.

Pour les assistants d'IA, c'est un avantage architectural important. Ils peuvent agir sur des concepts nommés, des rôles typés, des relations contextuelles et des actions déclarées, plutôt que sur du texte brut ou de simples lignes de base de données. L'hypergraphe leur offre une surface d'intervention structurée.

18.6 Évaluation persistante des motifs structurels

Les requêtes structurelles H-Logic utilisent les index persistants de l'hypergraphe. Le runtime choisit un pivot physique sélectif, énumère les relations réelles, valide les positions fixes et génériques, puis applique seulement ensuite les correspondances actives de la hiérarchie des types. Le choix d'index physique est distinct des alternatives logiques, comme le type exact, un sous-type ou un supertype.

Cette séparation est particulièrement importante pour les instances héritées. Une alternative de type transporte une correspondance entre les positions réelles de la relation candidate et les positions apparentes demandées par la requête. La relation demeure la candidate indexée : un nœud de type descendant ne remplace pas le pivot de la relation physique.

La prise en charge de l'ancien stockage transitoire est isolée derrière le symbole de compilation `TransientLegacy`. Le chemin d'exécution normal et la sémantique déclarative présentée dans ce guide reposent sur l'hypergraphe persistant.

Chapitre 19

Couches conceptuelles, projections et vues

Un modèle logiCells n'est pas une description monolithique : il possède plusieurs couches. Un même concept peut se présenter comme un objet conceptuel, un enregistrement persistant, une ligne de collection, un élément de vue, une cible d'action, un nœud de workflow ou un élément de mémoire accessible à l'IA. La programmation conceptuelle déclarative exige de distinguer clairement le modèle lui-même de ses projections.

Ce chapitre présente les couches, les facettes, les projections et les vues, afin d'éviter une erreur d'architecture fréquente : confondre un concept métier avec l'une de ses représentations.

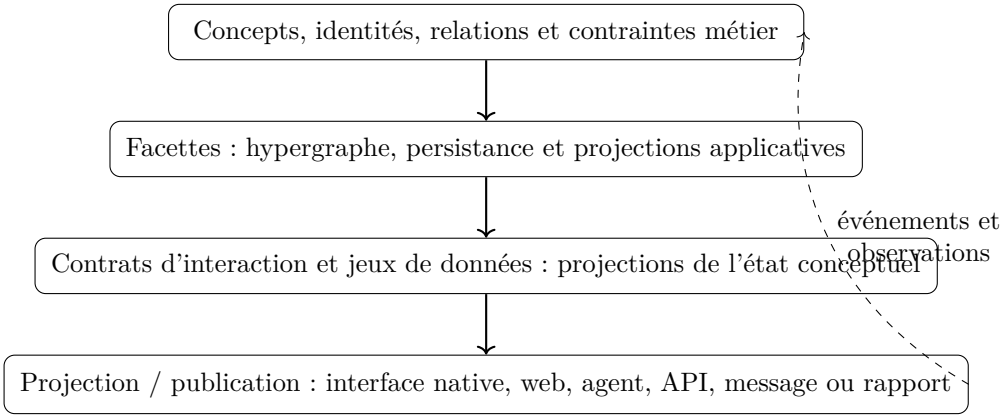


FIGURE 19.1 – La publication projette le sens conceptuel ; elle ne constitue pas l'origine de la signification métier.

19.1 Le concept n'est pas sa projection

Un concept est un élément sémantique stable. Une projection est une façon de l'exposer dans un but précis. Ainsi, le concept `Activity` peut être projeté comme structure hypergraphique, enregistrement de base de données, état de workflow, ligne d'interface, charge de service ou élément lisible par un assistant. Ces projections sont utiles, mais aucune ne définit, à elle seule, le concept.

C'est pourquoi les modèles logiCells distinguent les facettes. Une facette est une surface de projection nommée, associée à un concept. La facette `main` décrit généralement la structure hypergraphique ; une facette portant un nom de base de données peut décrire la persistance. Une couche de vue ou de collection peut ensuite réutiliser la structure déclarée pour exposer le concept aux utilisateurs et aux services.

19.2 Les facettes comme frontières de projection

Une facette délimite un mode d'interprétation des champs dans un contexte de projection donné. Dans une facette hypergraphe, les champs décrivent des rôles conceptuels. Dans une facette de base de données, ils décrivent des colonnes persistantes ou des valeurs projetées. Dans une vue orientée interface, ils peuvent décrire des éléments affichables, des légendes, des boutons d'action ou des liens de navigation.

Le fragment YAML suivant illustre cette séparation : une même classe contient une couche conceptuelle et une couche de base de données.

```yaml
class:
  classId: lgc#activity
  name: Activity
  type: Entity

  concepts:
    - Entity:
        facets:
          - main:
              type: hypergraph
              fields:
                - ActionPath: String
                - BaseObject: Object
          - Activity:
              type: database
              fields:
                - ActionPath:
                    size: 255
                    readOnly: false
                - BaseObject:
                    readOnly: true
                - FormalNameActivity:
                    rolePath: Activity.FormalName
                    nullable: true
```

Listing 19.1 – Séparation des facettes en YAML

L'essentiel n'est pas la syntaxe, mais la discipline de modélisation. La facette `main` exprime le sens conceptuel de l'objet. La facette de base de données exprime comment une partie de ce sens est rendue persistante ou interrogeable dans une projection de stockage.

19.3 Les chemins de rôle relient les couches

Un `rolePath` relie un champ projeté à un chemin conceptuel. Ce mécanisme évite l'isolement sémantique des champs projetés : une valeur affichée ou stockée reste rattachée à la structure conceptuelle qui lui donne sa signification.

Par exemple, `FormalNameActivity` peut être projeté dans une facette de base de données, alors que son `rolePath` pointe vers `Activity.FormalName`. Le champ peut être stocké, affiché ou interrogé, sans perdre son lien avec le rôle conceptuel représenté.

```yaml
FormalNameActivity:
  size: 100
  caption: '#FormalNameActivity'
  rolePath: Activity.FormalName
  nullable: true
```

Listing 19.2 – Champ projeté relié à un chemin de rôle conceptuel

Dans la documentation et les revues de modèles, les chemins de rôle doivent être considérés comme des engagements sémantiques. Ils ne sont pas de simples chemins techniques : ils expliquent pourquoi un champ existe et d'où vient sa signification.

19.4 Les contrats d'interaction comme projections du modèle

La couche VIEW moderne se comprend mieux comme une couche de contrats d'interaction. Un contrat ne définit pas le concept et n'impose pas un écran particulier. Il indique comment un concept ou une collection participe à une interaction : informations disponibles, valeurs éventuellement à fournir, actions invocables, événements observables et composants d'interaction fonctionnels pertinents.

Le participant peut être un utilisateur humain ou un agent autonome. Un moteur de rendu natif peut matérialiser le contrat sous forme de contrôles VCL/FMX ; un moteur web, sous forme de HTML ou de composants réactifs ; un runtime agentique peut l'utiliser comme plan d'interaction : présenter les informations utiles, ne demander que les valeurs manquantes et invoquer les mêmes capacités applicatives déclarées, sans afficher de formulaire.

Ce principe est essentiel à la portabilité. Un même contrat peut être mis en œuvre sur plusieurs canaux, tandis que le modèle conceptuel et la sémantique des actions restent stables. Le terme historique *view* subsiste dans les API et certains types de fichiers ; `meta/views/contracts` explicite son rôle indépendant du canal.

19.5 Évolution des modèles en couches

Les couches sécurisent également l'évolution des modèles. Un concept peut acquérir un champ de base de données sans changer d'identité. Une vue peut être repensée sans modifier la structure relationnelle. Un service peut exposer une partie des champs sans redéfinir le domaine. À l'inverse, un changement conceptuel voulu peut se propager à plusieurs projections.

C'est l'un des bénéfices pratiques de logiCells : séparer le sens métier stable des choix de publication et de persistance, tout en maintenant leurs liens grâce aux métadonnées.

Quatrième partie

H-Logic et expressions déclaratives

Chapitre 20

Fondamentaux de H-Logic

H-Logic est le langage logique compact de la couche hypergraphique de logiCells. Il complète les représentations natives des modèles YAML : YAML déclare les artefacts applicatifs, tandis que H-Logic exprime, dans le vocabulaire du graphe, les structures conceptuelles typées, les faits, les relations logiques, les requêtes, les projections et les commandes exécutables.

La grammaire actuelle privilégie l'explicite : les rôles sont nommés, les contextes sont des rôles ordinaires, les références à des termes nommés utilisent le préfixe `&`, le rattachement d'interface utilise `::` et les requêtes exposent leur résultat au moyen de `return`. H-Logic est ainsi adapté au raisonnement humain comme aux opérations conceptuelles générées automatiquement.

20.1 Pourquoi H-Logic est important

H-Logic fournit à logiCells un langage intermédiaire entre les métadonnées statiques et le code généraliste. Il permet d'exprimer une règle conceptuelle sans la dissimuler dans un script, de formuler une requête sans réduire le modèle à SQL et de décrire le rattachement d'une interface sans modifier l'identité de son objet support.

Cette approche présente plusieurs avantages pratiques :

— **Le sens métier reste visible.** Les concepts, rôles et relations demeurent explicites dans l'expression.

— **Les faits polyadiques sont des objets de premier ordre.** Un fait conserve tous ses participants au lieu d'être transformé prématurément en liens binaires.

— **Les requêtes et commandes partagent un même vocabulaire.** Une forme conceptuelle peut être assertée, interrogée ou utilisée comme corps d'une autre définition.

— **Une interface peut être rattachée sans remplacer l'identité.** L'opérateur `::` distingue l'objet support de la vue conceptuelle qui lui est rattachée.

— **Le langage est testable.** Les exemples publiés peuvent correspondre aux tests du parseur et de l'évaluation, plutôt que constituer du pseudo-code informel.

H-Logic est donc utile pour les règles, la validation sémantique, la recherche conceptuelle, les projections de modèles, l'automatisation explicable et les agents d'IA qui doivent agir sur des structures explicites plutôt que sur des détails d'implémentation opaques.

Il peut aussi intervenir plus tôt dans le cycle de développement, comme pivot de spécification vérifiable entre les besoins et les modèles déclaratifs générés. Ce processus est présenté dans Chapitre 29, “Des spécifications métier à une ontologie H-Logic validée” et appliqué à une étude de cas de gestion temporelle des stocks dans Chapitre 30, “Étude de cas : analyse temporelle des stocks ERP en H-Logic”.

20.2 Articulation avec YAML

Les trois formes présentées ci-dessous remplissent des fonctions différentes.

```yaml
class:
  classId: lgc#activity
  name: Activity
  type: Entity
  concepts:
    - Entity:
        facets:
          - main:
              type: hypergraph
              fields:
                - ActionPath: String
                - BaseObject: Object
```

Listing 20.1 – Déclaration d'un modèle YAML

```
define .:#Activity(me)[actionPath::#string, baseObject:#Object];
```

Listing 20.2 – Définition conceptuelle H-Logic

Les fragments YAML déclarent les métadonnées du modèle. Le fragment H-Logic rend visible la structure conceptuelle : une `Activity` possède un rôle support distinct `me`, une valeur chaîne du runtime encapsulée par `#string` et une référence à un objet conceptuel.

20.3 Termes, types et utilisation structurelle

Un terme typé s'écrit `x:#T`. Une instance conceptuelle nommée utilise un symbole formel tel que `#alice:#Person`. Une instance anonyme s'écrit `_:#Person`.

```
define .:#Person(me)[name::#string];
#alice:#Person;
_:#Person;
```

Listing 20.3 – Termes typés et instances

Le parseur autorise certaines annotations de types déclarées avant leur définition, mais l'utilisation structurelle exige que le type cible soit défini. Rattacher une interface ou fournir des arguments à un concept suppose ainsi une définition structurelle connue.

20.4 Rôles et attributs

Les rôles apparaissent entre parenthèses ; les attributs, entre crochets. Cette distinction traduit celle qui sépare les positions structurelles des propriétés descriptives.

```
define .:#worksFor(person:#Person, company:#Company, context:#Context);
define .:#Order(me)[amount::#float, status:#OrderStatus];
```

Listing 20.4 – Rôles et attributs

Un rôle peut comporter une annotation d'ordre structurel :

```
define .:#SecondOrderFact(item:#Item, context^2:#Context);
define .:#ThirdOrderEvidence(evidence^3:#Evidence, context^2:#Context);
```

Listing 20.5 – Métadonnées d'ordre des rôles

L'annotation `n̂` appartient aux métadonnées du rôle du type déclaré. Elle ne s'écrit pas sur une variable quantifiée.

20.5 Application d'un type connu avec des noms de rôles locaux

H-Logic peut appliquer un type déjà présent dans le graphe sémantique en attribuant à l'occurrence des noms de rôles locaux. Cette faculté est particulièrement utile dans les expressions et les contraintes.

```
.:#add(Amount, Tax)[result_1]
.:#power(Amount, 2)[squared_amount]
```

Listing 20.6 – Noms de rôles locaux sur un type d'opérateur connu

Si `#add` ou `#power` est défini dans l'environnement sémantique actif, ces expressions ne redéfinissent pas le type. Le type existant reste l'autorité pour la structure et l'arité ; `Amount`, `Tax`, `result_1` et `squared_amount` désignent les rôles de ce motif précis.

Un nom de rôle local n'est pas un concept `#variable` implicite. De même, le littéral `2` contraint une position du motif `#power` sans devenir un rôle public. H-Logic conserve ainsi une écriture lisible, tout en permettant au parseur de construire les correspondances internes de rôles nécessaires à la composition.

20.6 Rattachement d'interface et encapsulation des valeurs

L'opérateur `::` rattache une interface conceptuelle à un support existant. Il préserve l'identité du support tout en créant une boîte conceptuelle ou une vue.

```
define .:#Auditable(me)[auditLevel::#integer, lastAuditDate::#Date];
doc42::#Auditable;
```

Listing 20.7 – Rattachement d'une interface

La forme compacte correspond conceptuellement à une instance anonyme d'interface :

```
doc42::#Auditable;
_:#Auditable(doc42);
```

Listing 20.8 – Formes compactes et explicites

Pour nommer la boîte rattachée et pouvoir la réutiliser, employer la forme nommée complète :

```
auditBox:#Auditable(doc42)[auditLevel=2];
```

Listing 20.9 – Instance d'interface nommée

Les valeurs primitives du runtime sont également exposées par encapsulation conceptuelle :

```
define .:#PrimitiveDocument(me)[title::#string, checksum::#string];
doc42:#PrimitiveDocument[title='HDR chapter', checksum='abc123'];
```

Listing 20.10 – Encapsulation d'une valeur primitive

20.7 Termes nommés

Une référence explicite à un terme nommé s'écrit `&name`. Le préfixe possède une signification sémantique ; il ne fait pas partie du nom logique.

```
projectBeta := #projectBeta;
exists target=&projectBeta on .:#VisibleIn(label42, target);
```

Listing 20.11 – Liaison explicite d'un terme nommé

Une référence à un terme nommé ne peut pas servir de nom de rôle. La déclaration suivante est invalide :

```
define .:#B(&x:#A);
```

Listing 20.12 – Nom de rôle invalide

Il faut nommer le rôle puis contraindre sa valeur :

```
define .:#BFixed(x:#A=&a);
define .:#BDefault(x:#A default &a);
```

Listing 20.13 – Rôle avec terme nommé fixé ou par défaut

20.8 Quantificateurs

H-Logic prend en charge les quantificateurs universels, existentiels et d'agrégation. Les formes de modélisation les plus courantes sont `any`, `every`, `exists` et `some`.

```
exists y:#B on .:#R(y);
every y:#B on .:#R(y);
exists target=&projectBeta on .:#VisibleIn(label42, target);
any x:#A, exists y:#B on .:#Pair(x, y);
```

Listing 20.14 – Quantificateurs typés et liés à une valeur

Une variable quantifiée peut porter une restriction de type ou une liaison par égalité, mais pas les deux dans la même déclaration compacte. La forme `any y:#A=&a` est donc invalide.

20.9 Relations logiques et propriétés complémentaires

Les opérateurs logiques binaires comprennent l'implication, l'équivalence, la dérivation et l'exclusion. Les propriétés complémentaires se rattachent à la relation par `_[...]`.

```
#A ->_[weight=1, context=#c] #B;
#Diagnosis :-_[weight=0.8, context=#clinical] #SymptomPattern;
#Candidate !-_[context=#validation, reason=#missing_required_role] #Bottom;
#A <->_[context=#shared_context, confidence=0.95 as conf] #B;
```

Listing 20.15 – Relations logiques avec propriétés complémentaires

Ces compléments conservent la provenance, la confiance, le contexte ou les métadonnées de règle sur la relation logique elle-même.

20.10 Commandes et requêtes

Une assertion sans mot-clé est une commande de déclaration (*tell*). Les formes explicites `tell`, `do`, `ask` et `?` rendent visible l'intention de la commande.

```
projectBeta:#Context;
tell projectBeta:#Context;
do projectBeta:#Context;
ask .:#A(x) and .:#R(x, y) return x;
```

Listing 20.16 – Commandes tell, do et ask

Les requêtes n'exigent plus une liste initiale de variables telle que `?x`. Les rôles libres de l'expression cible constituent l'environnement de requête ; l'expression `return` sélectionne le résultat. L'enveloppe de requête accepte également les modes de raisonnement `homomorphic`, `unifying` et `resolving`, ainsi que des options de portée concernant les instances, types, sous-types, supertypes et valeurs masquées. Le contrat complet d'exécution et de résultat figure dans Chapitre 25, “Modes de requête H-Logic, portée et sémantique des résultats”.

```
? .:#Person(Tom)[name='Tom Hanks']
  and .:#like(Tom, Anne)
return (Tom, Anne);
```

Listing 20.17 – Requête renvoyant des résultats conceptuels

20.11 Définitions composites et projetées

Un concept peut être défini comme une composition logique de structures existantes.

```
define .:#ContextualRule(rule, context) as
  .:#Rule(rule)
  and .:#Context(context)
  and .:#validIn(rule, context);
```

Listing 20.18 – Définition composite

La liste des rôles publics peut projeter un corps interne plus riche, à condition que chaque rôle public soit contraint par celui-ci. Une interface conceptuelle publique stable peut ainsi masquer la complexité des compositions internes.

```
define .:#PersistedU(me, x);
define .:#PersistedV(me_, x, y);
define .:#ProjectedF(x, y) as
  .:#PersistedU(me, x)
  and .:#PersistedV(me_, x, y);
```

Listing 20.19 – Définition composite projetée

20.12 Domaines d'utilisation privilégiés de H-Logic

H-Logic est particulièrement utile lorsque l'application doit préserver et exécuter un sens explicite :

- contraintes et dérivations sémantiques ;
- requêtes conceptuelles sur les structures du graphe ;
- évaluation explicable des règles, avec contexte et degré de confiance ;
- concepts projetés à partir de plusieurs structures sources ;
- rattachement d'interfaces à des objets ou valeurs primitives existants ;
- agents d'IA ayant besoin d'actions typées, interrogeables et gouvernables sur la mémoire conceptuelle.

H-Logic ne remplace ni YAML ni functionalP. Les fichiers YAML sont des sources MODEL ; H-Logic exprime et échange les connaissances SEMANTIC ; functionalP fournit, selon le besoin, les comportements procéduraux ou les sources d'expressions. Une APPLICATION peut compiler à la fois des artefacts de modèle et des artefacts sémantiques, mais ces trois autorités restent distinctes.

20.13 H-Logic comme forme canonique d'échange

Le runtime peut également sérialiser les valeurs conceptuelles prises en charge en H-Logic canonique, puis analyser H-Logic au moyen de la grammaire officielle. Pour le développeur d'application, cela signifie que le langage des règles et requêtes peut aussi fournir une représentation compacte et inspectable des connaissances accumulées. Il n'est pas nécessaire d'inventer une autre notation pour les diagnostics, les assistants ou les synthèses sémantiques générées.

La frontière pratique est la suivante :

`connaissances logiCells` → `sérialisation H-Logic canonique` → `relecture, requête ou contexte d'assistant`

Lorsqu'un texte H-Logic provient d'une source externe ou d'un LLM, la conversion en sens inverse doit être plus stricte :

`H-Logic candidate` → `parseur officiel` → `validation symbolique` → `transaction ou intention opérationnalisable`

Un texte généré reste donc une *proposition*, non une connaissance active. Le développeur n'a pas besoin de gérer les composants internes du parseur ou du sérialiseur ; la règle essentielle est que H-Logic canonique est produit par la plateforme et qu'une proposition H-Logic n'est acceptée qu'après analyse et validation par le moteur.

Chapitre 21

Déclarer les concepts et les instances

Un modèle logiCells commence par la déclaration d'un artefact de classe, puis décrit la structure conceptuelle que le runtime matérialisera. Dans la syntaxe des modèles YAML, l'élément racine d'un fichier de modèle est généralement `class`. Le contrat de structure déclarative lui associe notamment les attributs `classId`, `name`, `className`, `currentGeneratorName` et `type`. La même structure existe en XML, mais YAML est la notation compacte retenue dans cet ouvrage pour déclarer les modèles applicatifs.

Une déclaration de concept n'est pas un simple libellé. Elle indique au runtime le type d'artefact modélisé, son mode d'identification, la classe d'objets du runtime capable de porter ses instances et les facettes conceptuelles qui lui sont associées. Une instance est ensuite une occurrence de cette structure : dossier, activité, équipement, objet métier, élément d'action planifiée ou tout autre objet régi par les mêmes métadonnées.

21.1 Déclaration de classe

Le fragment suivant est adapté du modèle `Activity`. Il montre la structure de tête minimale d'une classe de modèle logiCells.

```yaml
class:
  classId: lgc#activity
  name: Activity
  className: CustomClassItem
  currentGeneratorName: Common.CustomActivity
  type: Entity
  object:
    className: ObjectItemActivity
    objectName: Activity
```

Listing 21.1 – Déclaration de classe en YAML

La déclaration présente l'identifiant de classe du modèle, son nom lisible, l'objet de classe du runtime qui gère ce modèle et la représentation objet employée pour les instances. L'essentiel n'est pas ici la classe d'implémentation, mais la séparation entre le modèle conceptuel et les structures d'exécution capables de le matérialiser.

21.2 Concepts déclarés dans une classe

Dans une déclaration de classe, la section `concepts` définit l'interprétation conceptuelle. La grammaire accepte un élément `concept` sous `class/concepts` ; en YAML, celui-ci peut se présenter comme une entrée de table nommée `Entity`. logiCells s'en sert pour associer des contraintes et des facettes à la classe déclarée.

```yaml
class:
```

```yaml
  classId: lgc#activity
  name: Activity
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: CustomActivity
        facets:
          - main:
              type: hypergraph
              fields:
                - ActionPath: String
                - BaseObject: Object
```

Listing 21.2 – Déclaration de concept YAML dans une classe

Le concept `Entity` précise la participation de la classe à la couche conceptuelle. La contrainte exprime qu'`Activity` implique ou spécialise ici une autre classe conceptuelle, `CustomActivity`. La facette `main`, de type `hypergraph`, définit les rôles ou positions conceptuelles portés par le modèle.

21.3 Type de classe Entity et concept Entity

L'attribut de classe `type: Entity` et l'enregistrement conceptuel `Entity` ont des fonctions différentes. Le type de classe catégorise la déclaration de modèle ; à lui seul, il ne crée pas le concept sémantique `Entity` utilisé lors de l'enregistrement conceptuel et de l'application des contraintes.

Lorsqu'une entité applicative doit participer à la couche sémantique, déclarer explicitement le concept `Entity`.

```yaml
class:
  classId: app#work_item
  name: WorkItem
  className: CustomClassItem
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
        facets:
          - main:
              type: hypergraph
```

Listing 21.3 – Classe Entity avec concept sémantique Entity explicite

Cette distinction fournit un critère de diagnostic : si les journaux du runtime signalent l'absence d'enregistrement `Entity` pour une classe, examiner d'abord la déclaration `concepts` avant de modifier l'implémentation procédurale.

21.4 Identité implicite `me` des entités

En H-Logic, une entité non singleton possède, dans sa représentation propositionnelle, une identité d'instance implicite, conventionnellement nommée `me`. Cette identité relève de la représentation sémantique : ce n'est pas un champ applicatif à ajouter au modèle YAML.

```
define .:#WorkItem(me);
```

Listing 21.4 – Identité implicite d'une entité en H-Logic

La facette hypergraphique `main` du modèle déclaratif correspondant peut rester vide. Ne pas ajouter artificiellement un champ `me` ou `Me` pour reproduire l'écriture H-Logic. Les champs d'identification métier et les identifiants de persistance sont des projections distinctes.

21.5 L'implication comme correspondance sémantique

Une contrainte `implies` n'est pas une simple annotation d'héritage orienté objet. Les concepts source et cible doivent être compatibles sémantiquement pour permettre au runtime de construire la correspondance de rôles requise.

Pour les entités non singletons ordinaires, l'identité implicite `Me` fournit généralement le rôle positif commun :

```
UnitTest(Me)     -> Command(Me)
AgentProcess(Me) -> Process(Me)
```

Listing 21.5 – Implication conceptuelle sur l'identité d'une entité

Si les diagnostics indiquent qu'aucune correspondance polyadique ne peut être construite entre deux concepts, vérifier d'abord que les deux concepts sémantiques sont déclarés et que leurs rôles positifs sont compatibles. Ne pas ajouter de champs arbitraires pour masquer l'erreur de correspondance.

21.6 Champs conceptuels

Les champs d'une facette hypergraphique ne sont pas de simples colonnes de base de données : ils décrivent les rôles conceptuels de l'instance. Dans le modèle `Folder`, la facette `main` comporte notamment `Name`, `Description`, `ParentFolder`, `LastChanged` et `Model`. Ces noms respectent les conventions de génération actuelles ; sémantiquement, ils précisent ce que l'instance conceptuelle est capable de porter.

```yaml
class:
  classId: lgc#folder
  name: Folder
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
        facets:
          - main:
              type: hypergraph
              fields:
                - Name: String
                - Description: String
                - ParentFolder: lgc#folder
                - LastChanged: DateTime
                - Model: Boolean
```

Listing 21.6 – Facette hypergraphique YAML d'un concept de dossier

Le rôle `ParentFolder` est particulièrement révélateur : il permet à un dossier de référencer un autre dossier. Il s'agit déjà d'une relation conceptuelle au niveau du modèle, avant même de choisir sa sérialisation en table, son affichage dans une vue ou sa navigation dans une interface utilisateur.

21.7 Instances comme occurrences du runtime

Le modèle ne crée pas à lui seul un dossier ou une activité concrets : il déclare ce qu'est un dossier ou une activité. Une instance du runtime est une occurrence de la classe déclarée. Ainsi, un dossier donné peut posséder un nom, une description, un dossier parent et un indicateur de modèle. Ces valeurs respectent les rôles définis par la facette conceptuelle.

Il est essentiel de distinguer les éléments suivants :

— la déclaration de classe définit le type conceptuel ;

— la déclaration d'objet indique la représentation des instances dans le runtime ;
— la déclaration de concept fournit l'interprétation sémantique ;
— les facettes décrivent les différentes projections de ce concept ;
— les instances du runtime portent des valeurs conformes à ces déclarations.

21.8 Facettes de base de données et instances

La facette de base de données décrit la projection des instances conceptuelles en champs persistants. Elle ne se confond pas avec le concept. Un dossier ne se réduit pas à la table `lgcFolder`, laquelle n'est qu'une projection du modèle conceptuel.

```yaml
facets:
  - Folder:
      type: database
      map: lgcFolder
      fields:
        - IdFolder:
            generatorName: Common.Folder
            size: 32
            caption: '#Identifier'
            rolePath: Id
        - ClassIdFolder:
            size: 32
            defaultValue: lgc#folder
            caption: ClassId
            rolePath: ClassId
        - Name:
            fieldName: FolderName
            size: 100
            caption: '#FolderName'
            defaultValue: '#NewFolder'
        - IdParentFolder:
            size: 32
            packLinkedObject: true
            caption: '#ParentFolder'
            rolePath: ParentFolder.Id
            nullable: true
```

Listing 21.7 – Facette de persistance en base de données d'un dossier

Cette déclaration associe l'identité conceptuelle et les chemins de rôles aux champs persistants. `IdParentFolder` représente, côté stockage, le rôle conceptuel `ParentFolder.Id`. Le modèle conserve le chemin explicite au lieu de le dissimuler derrière une colonne quelconque de clé étrangère.

21.9 Lecture en H-Logic

Le même modèle peut être lu dans sa dimension conceptuelle en H-Logic. Cette écriture ne remplace pas le fichier YAML : elle propose une lecture logique compacte de sa structure conceptuelle.

```
define .:#Folder(me, parentFolder:#Folder)
  [name::#string, description::#string,
   lastChanged::#Date, model::#boolean];
```

Listing 21.8 – Lecture en H-Logic d'un concept de dossier

La vue H-Logic met en évidence le type conceptuel et ses rôles. La vue YAML décrit le modèle applicatif complet : classe du runtime, représentation objet, facettes, projection de persistance, libellés, valeurs par défaut et chemins de rôles.

Chapitre 22

Déclarer des relations et des faits réifiés

logiCells modélise les relations de deux façons complémentaires. Certaines correspondent à des rôles au sein d'un concept, tel le rôle `ParentFolder` d'un dossier. D'autres sont déclarées comme des classes de modèle autonomes, par exemple `BelongsTo`. Cette seconde forme est essentielle lorsqu'une relation doit porter sa propre identité, ses métadonnées, sa projection de persistance, ses actions, ses règles de visibilité ou son cycle de vie.

Ce chapitre distingue la référence par rôle de la relation réifiée. Une référence par rôle relie un concept à un autre au sein de la structure d'une entité. Une relation réifiée est elle-même un modèle de type entité dont la fonction est de représenter un fait entre plusieurs participants.

22.1 Références par rôle

Le modèle `Folder` contient un rôle pointant vers un autre dossier. Dans la facette hypergraphique, il est déclaré comme un champ conceptuel.

```yaml
facets:
  - main:
      type: hypergraph
      fields:
        - Name: String
        - ParentFolder: lgc#folder
```

Listing 22.1 – Référence par rôle YAML dans une facette hypergraphique

La facette de base de données projette ce rôle conceptuel au moyen de chemins de rôles.

```yaml
facets:
  - Folder:
      type: database
      map: lgcFolder
      fields:
        - ClassIdParentFolder:
            size: 32
            packLinkedObject: true
            caption: '#ParentFolderClass'
            rolePath: ParentFolder.ClassId
            nullable: true
            defaultValue: lgc#folder
        - IdParentFolder:
            size: 32
            packLinkedObject: true
            caption: '#ParentFolder'
            rolePath: ParentFolder.Id
            nullable: true
```

Listing 22.2 – Chemins de rôles YAML vers un dossier parent

Les noms des champs relèvent du stockage. Les attributs `rolePath` conservent le chemin conceptuel : le runtime sait ainsi que ces champs ne sont pas des chaînes isolées, mais des projections de l'objet associé `ParentFolder`.

22.2 Relations réifiées

Une relation réifiée est une relation représentée par son propre modèle. `BelongsTo` illustre ce modèle de conception : il déclare une entité conceptuelle avec deux participants, `EntityA` et `EntityB`. La relation reçoit son propre identifiant de classe, une représentation objet dans le runtime et des champs persistants.

```yaml
class:
  classId: lgc#belongs_to
  name: BelongsTo
  className: CustomClassItem
  currentGeneratorName: Common.BelongsTo
  type: Entity
  object:
    className: CustomObjectItem
    objectName: BelongsTo
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
        facets:
          - main:
              type: hypergraph
              fields:
                - EntityA: lgc#entity
                - EntityB: lgc#entity
```

Listing 22.3 – Déclaration YAML d'un modèle de relation réifiée

Bien que la déclaration utilise `type: Entity`, son intention conceptuelle est relationnelle. Il n'y a aucune contradiction : une relation réifiée est un fait que l'on peut identifier, stocker, filtrer, afficher, protéger et enrichir.

22.3 Pourquoi réifier une relation ?

Un rôle simple suffit quand la relation n'est qu'une référence structurelle. La réification devient pertinente lorsque le fait doit être manipulé comme un objet du modèle : cycle de vie propre, présence dans une collection, actions attachées, audit nécessaire ou rôles additionnels.

Par exemple, `BelongsTo(EntityA, EntityB)` peut exprimer l'appartenance, l'inclusion, le regroupement ou la propriété. Grâce à sa réification, la relation peut ensuite recevoir un contexte, une période de validité, une source, un degré de confiance, un état ou des informations de workflow sans rompre le modèle conceptuel.

22.4 Persistance des relations réifiées

La facette de base de données de `BelongsTo` projette la relation et ses participants dans des champs persistants. La relation possède ses propres champs d'identité et chaque participant est projeté par un chemin de rôle.

```yaml
facets:
  - Agency:
      type: database
      map: lbsAgency
      fields:
        - IdBelongsTo:
            generatorName: Common.Agency
```

```
            size: 32
            caption: '#Identifier'
            rolePath: Id
        - ClassIdBelongsTo:
            size: 32
            defaultValue: lgc#belongs_to
            caption: ClassId
            rolePath: ClassId
        - IdEntityA:
            size: 32
            caption: '#IdEntityA'
            rolePath: EntityA.Id
            nullable: true
        - ClassIdEntityA:
            size: 32
            caption: '#ClassIdEntityA'
            defaultValue: lgc#entity
            rolePath: EntityA.ClassId
            nullable: true
        - IdEntityB:
            size: 32
            caption: '#IdEntityB'
            rolePath: EntityB.Id
            nullable: true
        - ClassIdEntityB:
            size: 32
            caption: '#ClassIdEntityB'
            defaultValue: lgc#entity
            rolePath: EntityB.ClassId
            nullable: true
```

Listing 22.4 – Projection en base de données d'une relation réifiée

Ce modèle de conception est important : la relation possède sa propre identité et chaque rôle sa propre identité projetée. Il est plus riche qu'une clé étrangère classique, car le chemin de rôle maintient explicitement le lien sémantique.

22.5 Lecture en H-Logic

En H-Logic, le sens conceptuel de `BelongsTo` peut s'exprimer comme une relation comportant deux rôles typés.

```
define .:#BelongsTo(entityA:#Entity, entityB:#Entity);
```

Listing 22.5 – Lecture H-Logic de la relation réifiée BelongsTo

Lorsque la relation doit elle-même être adressable, on peut attribuer une identité à un fait nommé au lieu d'utiliser une occurrence anonyme.

```
define .:#BelongsTo(entityA:#Entity, entityB:#Entity);
fact1:#BelongsTo(entityA, entityB);
```

Listing 22.6 – Fait relationnel nommé

Un fait nommé est adressable indépendamment de ses participants. La relation peut dès lors porter ses propres attributs, provenance, confiance, validité ou état de workflow.

22.6 Les chemins de rôles comme ponts conceptuels

Les chemins de rôles constituent l'un des principaux ponts entre modélisation conceptuelle et persistance. Le champ `IdEntityA` n'est pas une simple colonne : son attribut `rolePath: EntityA.Id` indique qu'il correspond à l'identifiant du participant occupant le rôle `EntityA` dans la relation.

Le modèle conserve ainsi une sémantique stable d'une projection à l'autre. La même relation peut être lue comme un fait de l'hypergraphe, stockée dans une ligne de base de données, affichée dans une vue, exposée par un service ou parcourue par un outil.

Chapitre 23

Contextes, interfaces et termes nommés

Les contextes, interfaces et termes nommés répondent à trois problèmes distincts : déterminer où un fait est valide, exposer conceptuellement un support existant et référencer explicitement un terme logique connu. La grammaire H-Logic actuelle maintient cette séparation.

23.1 Les contextes sont des rôles ordinaires

H-Logic n'utilise plus de suffixe contextuel particulier. Un contexte est modélisé comme un rôle typé ordinaire.

```
define .:#Context(me)[label::#string];
define .:#VisibleIn(item:#Item, context:#Context);
```

Listing 23.1 – Déclaration d'un contexte comme rôle typé

Les contextes deviennent ainsi composables. Un projet, un locataire, une exécution de workflow, un périmètre de sécurité ou un cadre temporel peuvent être représentés par un concept ordinaire et transmis explicitement au travers des relations.

23.2 Ordre des rôles

Lorsqu'un contexte intervient à un ordre structurel supérieur, la déclaration de rôle peut porter une métadonnée `n̂`.

```
define .:#SecondOrderFact(item:#Item, context^2:#Context);
define .:#ThirdOrderEvidence(evidence^3:#Evidence, context^2:#Context);
```

Listing 23.2 – Annotation d'ordre sur un rôle contextuel

L'annotation d'ordre appartient au rôle déclaré. Les variables quantifiées continuent d'utiliser de simples noms de variables.

23.3 Les interfaces préservent l'identité du support

Un concept d'interface se définit généralement par un rôle `me` distinct. L'opérateur `::` rattache cette interface à un support existant.

```
define .:#Auditable(me)[auditLevel::#integer];
doc42::#Auditable;
```

Listing 23.3 – Rattachement d'une interface au support

Le support `doc42` reste le même objet. Le rattachement introduit une vue conceptuelle sur ce support. Ce mécanisme est utile lorsqu'un même objet doit participer à plusieurs interfaces conceptuelles sans duplication ni changement destructif de type.

23.4 Instance directe et rattachement d'interface

Les opérateurs `:` et `::` ont des significations différentes.

```
#alice:#Person;
#alice::#Customer;
```

Listing 23.4 – Instance directe et rattachement d'interface

La première instruction déclare `#alice` comme une instance directe de `#Person`. La seconde rattache l'interface `#Customer(me)` au support `#alice` déjà existant.

23.5 Termes nommés explicites

Une référence comme `&projectBeta` résout un terme nommé à partir de l'environnement.

```
projectBeta := #projectBeta;
exists target=&projectBeta on .:#VisibleIn(label42, target);
```

Listing 23.5 – Référence explicite à un terme nommé

Le préfixe désambiguïse une référence de valeur par rapport à une déclaration de rôle. Il est particulièrement utile dans les règles générées et les requêtes conceptuelles paramétrées.

23.6 Valeurs fixées et valeurs par défaut

Un rôle peut être lié à un terme nommé ou recevoir une valeur par défaut.

```
define .:#FixedScope(context:#Context=&globalContext);
define .:#DefaultScope(context:#Context default &globalContext);
```

Listing 23.6 – Valeurs fixées et valeurs par défaut

Une valeur fixée contraint la résolution. Une valeur par défaut fournit une valeur lorsqu'aucune n'est indiquée explicitement. Il ne faut pas confondre les deux.

23.7 Domaines de rôles complexes

Le type d'un rôle peut être un type nommé ou une expression logique de type, délimitée par des parenthèses.

```
define .:#Choice(role:(#A or #B));
```

Listing 23.7 – Domaine de rôle constitué d'une expression de type

Pour réutiliser une sémantique de sous-ensemble, définir un type nommé dans le graphe, puis employer son nom comme domaine du rôle.

```
.:#SubsetContext -> .:#Context;
define .:#ContextFamily(context:#SubsetContext);
```

Listing 23.8 – Type nommé comme domaine de rôle

Les domaines de rôle ressemblant à des appels de constructeur, tels que `Subset(#Context)`, ne constituent pas la forme stabilisée.

23.8 Pourquoi ces distinctions comptent

Les contextes explicites renforcent la traçabilité. Le rattachement d'interfaces permet plusieurs vues conceptuelles d'un même support. Les termes nommés lèvent toute ambiguïté sur les liaisons de valeurs. Ensemble, ces mécanismes rendent H-Logic pertinent pour les règles explicables, les autorisations contextuelles, les projections sémantiques et les opérations d'agents sur la mémoire conceptuelle.

Chapitre 24

Requêtes sur les structures conceptuelles

Les requêtes H-Logic opèrent sur des structures conceptuelles plutôt que sur des tables de stockage. Elles exposent directement les rôles, relations, contextes et compositions logiques, ce qui rend l'intention de la requête lisible et traçable.

24.1 Formes `ask` et point d'interrogation

Les mots-clés `ask` et `?` introduisent tous deux une requête. L'expression qui suit `return` en définit le résultat.

```
ask .:#A(x) and .:#R(x, y) return x;
```

Listing 24.1 – Requête conceptuelle simple

```
? .:#Person(Tom)[name='Tom Hanks']
  and .:#like(Tom, Anne)
return (Tom, Anne);
```

Listing 24.2 – Requête renvoyant plusieurs termes

Les rôles libres de l'expression forment l'environnement de la requête. Les anciennes formes à variables préfixées, comme `?x := ...`, ne font plus partie de la grammaire actuelle. Une requête peut également choisir un mode de raisonnement et des options de portée. La correspondance homomorphe est utilisée par défaut ; le mode unifiant combine les opérandes compatibles, tandis que le mode résolvant démontre les buts à partir des faits et des règles. Voir Chapitre 25, "Modes de requête H-Logic, portée et sémantique des résultats" pour les détails sémantiques.

24.2 Quantification typée

Les quantificateurs restreignent les rôles parcourus par un motif conceptuel.

```
exists y:#B on .:#R(y);
every y:#B on .:#R(y);
any x:#A, exists y:#B on .:#Pair(x, y);
```

Listing 24.3 – Contraintes existentielles et universelles

`exists` exige au moins une valeur correspondante ; `every` contraint l'ensemble des valeurs associées par la relation cible ; `any` établit le domaine source du motif qui suit.

24.3 Liaison à un terme nommé

Un rôle quantifié peut être lié à un terme nommé explicite.

```
projectBeta := #projectBeta;
exists target=&projectBeta on .:#VisibleIn(label42, target);
```

Listing 24.4 – Requête liée à une valeur d'environnement

Cette possibilité facilite la construction de requêtes paramétrées par le projet, le locataire, l'exécution d'un processus ou le contexte de sécurité sélectionnés.

24.4 Composition logique

Une requête peut associer des motifs conceptuels avec `and`, `or`, des expressions arithmétiques et des opérateurs relationnels.

```
? .:#Person(tom, #Yesterday)[name='Tom Hanks']
  and r:#like(tom, Anne, #Yesterday)
  and s:#hate(Anne, John, #Yesterday)
return (tom, r, s);
```

Listing 24.5 – Requête conceptuelle composée

Nommer les faits relationnels, par exemple `r` et `s`, permet à la requête de retourner ou de contraindre ces faits eux-mêmes, et non seulement leurs participants.

24.5 La requête comme passerelle exécutable

Une requête peut retourner une commande.

```
ask .:#A(me, name, context) and (name='toto')
return do _:#love(me, #you, context);
```

Listing 24.6 – Requête suivie d'une action conceptuelle

Cette forme se prête à l'automatisation déclarative : la condition et l'opération conceptuelle qui en découle restent exprimées dans un même langage ; elles peuvent être inspectées, tracées et soumises à des règles de gouvernance.

24.6 Intérêt des requêtes H-Logic

Par rapport aux requêtes orientées stockage, H-Logic préserve la visibilité des noms de rôles et des relations conceptuelles. Il facilite ainsi les recherches explicables, la validation sémantique, les assistants conscients du modèle et les règles capables de survivre aux changements de persistance physique. Les fournisseurs du runtime peuvent néanmoins traduire ou optimiser l'exécution ; l'expression source reste ancrée dans le modèle conceptuel. Le contrat de cardinalité des résultats est polymorphe : aucune réponse correspond à `nil`, une réponse unique est retournée directement et plusieurs réponses forment une liste conceptuelle chaînée.

Chapitre 25

Modes de requête H-Logic, portée et sémantique des résultats

La commande `ask` fournit un cadre déclaratif commun à plusieurs opérations de raisonnement. Toute requête définit une cible conceptuelle et une projection de résultat. Son *mode* choisit l'opération de raisonnement ; ses *options* délimitent les types et les instances susceptibles d'y participer.

Cette distinction est essentielle. L'expression de requête décrit *ce qui est recherché*. Le mode détermine *comment* trouver des structures compatibles. Les options précisent *dans quel périmètre* la recherche peut s'effectuer.

25.1 Les trois modes de requête

Le runtime actuel reconnaît trois modes.

`homomorphic` Recherche les structures conceptuelles persistées qui correspondent à un motif structurel. Il s'agit du mode par défaut.

`unifying` Unifie au moins deux opérandes conceptuels compatibles et renvoie le résultat canonique matérialisé.

`resolving` Résout un objectif à partir de faits et de règles de type Horn, y compris par récursion et mémoïsation tabulaire lorsque le programme le permet.

Le mode modifie l'algorithme de raisonnement, et non le vocabulaire de la requête. Concepts, rôles, faits nommés et projections de retour conservent leur signification conceptuelle.

```
ask homomorphic .:#Person(p)[name='Ada']
  return p;

ask unifying (.:#Candidate(a), .:#Candidate(b))
  return (a, b);

ask resolving .:#ancestor(alice, descendant)
  return descendant;
```

Listing 25.1 – Une même forme de requête avec différents modes de raisonnement

En l'absence d'un mode explicite, la requête est interprétée en mode `homomorphic`.

25.2 Recherche de motifs homomorphes

Le mode homomorphe parcourt l'hypergraphe conceptuel à la recherche de valeurs persistées compatibles avec le motif de la requête. Une valeur de rôle fixée constitue une contrainte exacte. Un rôle omis ou non contraint agit comme un joker.

```
// Exact structural pattern
ask .:#worksFor(#alice, #logiCells, #production)
  return this;

// Partial pattern: company and context remain unconstrained
ask .:#worksFor(#alice, company, context)
  return (company, context);
```

Listing 25.2 – Motifs homomorphes exacts et partiels

Un joker représente l'absence de contrainte. Il ne demande pas de comparer la valeur candidate à une valeur `nil` particulière. Cette distinction permet d'utiliser le même itérateur structurel pour les motifs partiels et exacts, sans écarter les candidats dont les positions sont renseignées.

Lorsqu'une requête comporte plusieurs éléments de motif, ils sont évalués dans l'ordre de la requête. Les valeurs correspondantes sont ajoutées au résultat dans ce même ordre.

25.3 Mode d'unification

Le mode d'unification combine des opérandes compatibles en une valeur conceptuelle canonique unique. Il préserve les égalités imposées par les variables répétées, unifie récursivement les valeurs construites imbriquées et rejette les incompatibilités de constructeur, d'arité ou de valeur rigide.

```
ask unifying (
  .:#Pair(x, #right),
  .:#Pair(#left, y)
)
return (x, y);
```

Listing 25.3 – Unification d'opérandes conceptuels compatibles

Une requête d'unification nécessite au moins deux opérandes. Au-delà de deux, l'unification porte sur toute la séquence : l'incompatibilité d'un seul opérande rend le résultat vide. La projection peut renvoyer des descripteurs, des constantes ou une liste composée d'éléments issus de plusieurs opérandes.

Le résultat matérialisé est canonique : des demandes équivalentes peuvent partager la même valeur conceptuelle au lieu de créer des doublons structurels.

25.4 Mode de résolution

Le mode de résolution traite la requête comme un objectif logique. Le résolveur sélectionne les clauses selon leur constructeur canonique, introduit des variables fraîches à chaque application de règle, propage les substitutions de gauche à droite dans les corps conjonctifs et ne projette que les variables demandées.

```
ask resolving .:#reachable(#warehouseA, destination)
  return destination;
```

Listing 25.4 – Résolution d'un objectif à partir de faits et de règles

Plusieurs faits ou clauses alternatives peuvent produire plusieurs réponses. Les preuves en double sont dédupliquées. Un cycle récursif dépourvu de fait de base ne produit aucune réponse ; les programmes récursifs tabulés peuvent fournir des réponses transitives sans développer indéfiniment le même objectif non résolu.

25.5 Options de portée

Les options de portée sont indépendantes du mode. Elles déterminent si le résultat peut inclure le type demandé lui-même, ses instances, sa position dans la hiérarchie des types ou une combinaison de ces éléments.

`masked` Masque les valeurs exclues par la politique active de visibilité ou de masquage.

`instances` Inclut les instances correspondantes en plus du périmètre par défaut.

`instances only` Limite les résultats aux instances.

`types` Inclut les types correspondants en plus du périmètre par défaut.

`types only` Limite les résultats aux types.

`subtypes` Inclut le type demandé et ses descendants.

`subtypes only` Limite la partie de la hiérarchie parcourue aux descendants du type demandé.

`supertypes` Inclut le type demandé et ses ancêtres.

`supertypes only` Limite la partie de la hiérarchie parcourue aux ancêtres du type demandé.

Les parcours `subtypes` et `supertypes` sont réflexifs : le type demandé reste une possibilité exacte, sauf si une restriction `only` l'exclut dans le cadre de la politique de résultats retenue.

```
ask homomorphic subtypes instances .:#Document(item)[title=title]
  return (item, title);
```

Listing 25.5 – Recherche homomorphe d'instances parmi les descendants

25.6 Sens du parcours dans la hiérarchie des types

L'héritage est représenté par la relation logique `#supertype(Map, Source, Target, Context)`. `Source` désigne le sous-type et `Target` le supertype.

Pour un type demandé `T` :

— le sens interne `>` suit les relations dont la cible vaut `T` et récupère leur source : il découvre ainsi les sous-types ;

— le sens interne `<` suit les relations dont la source vaut `T` et récupère leur cible : il découvre ainsi les supertypes ;

— `=` conserve le type exact ;

— `*` ne contraint pas la catégorie recherchée.

Ces symboles expriment le sens de parcours de l'itérateur ; ils sont utiles pour interpréter les traces et les diagnostics du moteur. Les auteurs de requêtes H-Logic déclaratives obtiennent normalement le même comportement au moyen des options `subtypes` et `supertypes`.

25.7 Instances héritées et vues adaptées

Un sous-type peut réordonner, projeter ou identifier entre eux les rôles de son supertype. Par conséquent, une instance descendante interrogée au moyen d'un motif du type ancêtre ne peut pas toujours être renvoyée sous sa forme brute. Le runtime fournit alors une vue conceptuelle adaptée qui préserve l'identité de l'instance réelle tout en exposant les rôles dans l'ordre apparent du type demandé.

```
define .:#Parent(left:#Value, right:#Value);
define .:#Child(first:#Value, second:#Value);

// Conceptual inheritance mapping: Child.first -> Parent.right,
// Child.second -> Parent.left
```

```
#Child ->_[map=#[[0,1],[1,0]]] #Parent;
```

Listing 25.6 – Sous-type dont les rôles sont exposés selon le mappage du type parent

Une requête sur `Parent` avec l'option `subtypes` peut trouver un `Child(first=A, second=B)` persisté. La valeur retournée désigne toujours cette instance enfant, mais les positions apparentes de `Parent` deviennent `left=B` et `right=A`.

L'invariant du mappage est le suivant :

> Les positions réelles de la valeur candidate sont mises en correspondance avec les positions apparentes demandées.

Cet invariant est maintenu dans les parcours exacts, descendants et ascendants. En remontant la hiérarchie, le moteur inverse ou compose les mappages selon les besoins, avant d'exposer le résultat.

25.8 Les mappages polyadiques préservent les informations d'égalité

Un mappage polyadique ne se réduit pas à une permutation. Sa forme normale peut représenter une projection, des positions répétées et des égalités entre champs. Ainsi, plusieurs positions réelles peuvent appartenir à une même classe d'égalité, ou un type public peut omettre une position privée d'un descendant.

La forme polyadique normale constitue donc la référence sémantique. L'accès à l'exécution détermine une correspondance unique, dans le sens nécessaire :

— la comparaison d'un candidat associe une position réelle à la position apparente du motif ;
— l'accès à une valeur adaptée associe une position apparente demandée à la position réelle de l'instance.

Une correspondance n'est utilisée que si elle est unique. Les mappages ambigus demeurent représentés comme des informations d'égalité : ils ne sont jamais réduits silencieusement à une permutation simple, mais incorrecte. L'implémentation peut mettre en cache les correspondances validées pour améliorer les performances ; ce cache est dérivé du mappage polyadique normal et ne s'y substitue pas.

25.9 Contrat de résultat

Toute valeur renvoyée par une requête suit un même contrat polymorphe :

— aucun résultat : `nil` ;
— un seul résultat : le `PNodeValue` correspondant, directement ;
— plusieurs résultats : une liste chaînée `PListValue`.

Toute valeur de nœud peut donc être interprétée selon la sémantique d'une liste. Un résultat unique se comporte comme une liste à un élément, dont la tête est la valeur elle-même et la queue vaut `nil`. L'ajout d'une seconde valeur transforme le résultat en véritable liste chaînée.

L'expression retournée peut elle-même être une liste. Le runtime ne lui ajoute pas artificiellement une liste englobante à un seul élément. La forme déclarative du résultat reste donc fidèle à l'expression écrite après `return`.

```
// No match: nil
ask .:#Missing(x) return x;

// One match: the value itself
ask .:#Person(p)[name='Ada'] return p;

// Several matches: linked list of values
ask .:#Person(p) return p;
```

```

// One projected tuple: the tuple itself, not [[p, name]]
ask .:#Person(p)[name=name] return (p, name);
```

Listing 25.7 – Exemples de formes de résultats

25.10 Itération persistante et performances

L'évaluation d'un motif structurel utilise les index persistants des relations de l'hypergraphe. L'itérateur sélectionne d'abord un pivot physique grâce à un index sélectif, énumère les relations candidates réelles, vérifie les contraintes exactes ou les jokers, puis applique tout mappage de type actif avant de retourner la valeur.

Le pivot physique et les alternatives de type logique sont deux éléments distincts. Un type descendant constitue une possibilité de filtrage assortie d'un mappage ; il ne remplace pas le pivot fourni par l'index des relations. Cette séparation permet aux recherches exactes et héritées d'emprunter le même parcours d'exécution indexé.

Les branches héritées de l'hypergraphe transitoire sont isolées derrière le symbole de compilation `TransientLegacy`. Les compilations ordinaires empruntent le chemin persistant. Le code de compatibilité restant est ainsi clairement identifié et peut être supprimé progressivement sans modifier la sémantique déclarative des requêtes.

25.11 Choisir le mode adapté

Utilisez `homomorphic` lorsque la réponse recherchée existe déjà sous forme de structure conceptuelle persistée ; `unifying` lorsque plusieurs descriptions doivent fusionner en une valeur canonique compatible ; et `resolving` lorsqu'il faut démontrer la réponse à l'aide de faits et de règles.

Ajoutez ensuite des options de portée uniquement pour la région conceptuelle nécessaire. En particulier, n'utilisez pas `subtypes` pour compenser une déclaration de type imprécise : le parcours des descendants modifie à la fois la sélection des candidats et le mappage des rôles. Il doit donc correspondre à une décision sémantique explicite.

25.12 H-Logic, autorité unique pour la recherche de connaissances

Utilisez `KnowledgeQuery.SemanticPattern` pour transmettre une requête H-Logic canonique. Les autres champs, comme `WorldContextSelector`, `EvidenceRequirements` et `PolicyContext`, définissent la portée et les contrôles d'exécution ; ils ne constituent pas un second langage de requête sur graphe.

Lorsque des contraintes FunctionalP participent à une requête de connaissances, le runtime traduit leur expression compilée en H-Logic avant l'évaluation. Les prédicats intégrés, tels que `#add`, possèdent leurs propres contrats d'arité et de types. Une expression H-Logic locale ne les redéfinit pas.

Un mappage polyadique décrit à la fois les correspondances entre rôles et les contraintes d'égalité. Des rôles de même nom peuvent être associés implicitement si la structure le permet. Pour des noms différents ou des classes d'équivalence plus complexes, déclarez un mappage explicite afin de préserver la sémantique des projections et des types hérités.

Utilisez les modes `ask` disponibles dans le runtime installé pour les requêtes interactives. Les fonctions distinctes de compilation de requêtes, lorsqu'elles sont proposées, doivent être vérifiées auprès de la CLI ou de l'API cible. L'exécution d'une requête `ask` ne démontre pas, à elle seule, la présence d'une commande de compilation séparée.

Chapitre 26

Règles, contraintes et modèles éprouvés

Les règles et contraintes rendent un modèle plus sûr qu'un simple ensemble de formulaires et de scripts. Elles définissent les conditions de validité, ce qui peut être inféré et les combinaisons de rôles, d'états et de valeurs qui ont un sens.

26.1 Les contraintes comme engagements du modèle

Une contrainte exprime un engagement porté par le modèle. Elle peut stipuler qu'un concept en implique un autre, qu'une valeur est obligatoire, qu'une valeur par défaut doit s'appliquer ou qu'un rôle doit respecter un type particulier.

```
constraints:
  - implies: lgc#entity
```

Listing 26.1 – Implication au niveau d'un concept

Cette implication indique que le concept participe au contrat sémantique et au contrat d'exécution de `lgc#entity`. Elle dépasse la simple notion d'héritage dans une hiérarchie de classes : elle exprime une compatibilité conceptuelle.

26.2 Les règles comme interprétation exécutable

Une règle peut valider un état, dériver une valeur, autoriser une action, acheminer un événement ou sélectionner une vue. Dans logiCells, les règles doivent être exprimées au plus près de l'élément du modèle qu'elles gouvernent : une règle de validation appartient au voisinage du concept ou de l'action concernée ; une règle d'acheminement se place près de l'événement ou du processus qu'elle commande.

26.3 Contraintes d'expression et compatibilité des rôles

Certaines contraintes utilisent une expression plutôt que la seule désignation d'un autre concept. Pour le développeur, le contrat essentiel est le suivant : l'expression est compilée en une cible sémantique, puis logiCells relie le concept source à cette cible au moyen de rôles conceptuels compatibles.

```
source roles
  + expression roles
  -> compatible role mapping
  -> semantic constraint
```

Listing 26.2 – Principe de liaison d'une contrainte

Un mappage explicite, lorsqu'il existe, fait autorité. Dans les autres cas, la correspondance repose sur les rôles compatibles exposés de part et d'autre. Si l'expression ne mentionne que des rôles absents du concept source, la contrainte n'est pas valide pour cette source : elle doit être signalée ou écartée plutôt que faire l'objet d'une correspondance positionnelle supposée.

Cette règle est utile lors de la revue d'une contrainte : les noms des rôles font partie de son interface conceptuelle. Renommer un rôle dans une expression sans disposer d'un rôle source correspondant, ni d'un mappage explicite, peut rendre la contrainte inapplicable.

26.4 Modèles éprouvés

Un modèle éprouvé est une petite structure réutilisable dont le fonctionnement a été validé sur plusieurs modèles. Exemples :

— une entité possédant une facette hypergraphique et une projection en base de données ;
— une relation réifiée avec des chemins de rôles pour ses deux participants ;
— un objet de processus doté des actions `Start`, `Validate` et `Finalize` ;
— une action publiée avec des paramètres typés ;
— un modèle de message portant une date de création, une date de réception et les métadonnées de l'expéditeur ;
— une vue liée à un curseur et à un chemin de collection.

Ces modèles éprouvés sécurisent l'évolution : l'équipe peut ajouter des concepts sans réinventer l'identité, la persistance, le traitement des événements et la liaison des vues à chaque modification.

26.5 Emplacement des contraintes

Une contrainte doit être placée au niveau le plus bas où elle demeure vraie. Une contrainte toujours applicable à un concept lui appartient ; une contrainte propre au déroulement d'un processus appartient au processus ; une contrainte limitée à l'exposition d'un service appartient à la couche de publication.

26.6 Règles de conception

Ne laissez pas les scripts porter seuls la validité. Ils doivent exécuter les opérations, non constituer l'unique source de vérité sémantique. Rendez les contraintes visibles dans le modèle, testez les structures récurrentes indépendamment du processus applicatif et préférez de petites contraintes précisément délimitées aux grandes validations procédurales.

26.7 Synthèse

Les règles et contraintes transforment un modèle déclaratif en structure exécutable maîtrisée. Les modèles éprouvés rendent ces structures reproductibles et évolutives.

Chapitre 27

Erreurs d'interprétation courantes de H-Logic

H-Logic est concis, mais chacun de ses opérateurs possède une signification précise. Les distinctions suivantes reflètent la grammaire actuelle et les tests du parseur.

27.1 Le nom d'un rôle n'est pas une référence à un terme nommé

Le préfixe `&` désigne une valeur nommée de l'environnement. Il ne peut pas remplacer le nom d'un rôle.

```
// Invalid
// define .:#B(&x:#A);
```

Listing 27.1 – Formes invalides et valides

La forme valide est la suivante :

```
define .:#B(x:#A=&a);
```

Listing 27.2 – Rôle contraint par un terme nommé

27.2 La restriction de type et la liaison par égalité sont distinctes

Une variable quantifiée accepte soit une restriction de type, soit une liaison par égalité.

```
exists y:#A on .:#R(y);
exists y=&a on .:#R(y);
```

Listing 27.3 – Formes de quantification valides

La forme compacte combinée `exists y:#A=&a` n'appartient pas à la grammaire.

27.3 Les contextes sont des rôles, non des appels de fonction

L'écriture `context(projectBeta)` ne représente pas la syntaxe contextuelle actuelle. Utilisez un rôle de contexte et liez-le explicitement.

```
define .:#VisibleIn(item:#Item, context:#Context);
projectBeta:#Context;
_:#VisibleIn(label42, projectBeta);
```

Listing 27.4 – Relation contextualisée explicite

27.4 Les opérateurs : et :: ont des sens différents

: déclare ou type un terme ; :: rattache une interface à un support existant.

```
#alice:#Person;
#alice::#Customer;
```

Listing 27.5 – Typage et rattachement d'interface

Confondre ces formes fait disparaître la distinction entre l'objet et la boîte conceptuelle qui lui est rattachée.

27.5 Un usage structurel exige un type défini

Une simple annotation anticipée peut être acceptée, mais une application structurelle nécessite une définition connue.

```
define .:#Context(me);
trace42::#Context;
```

Listing 27.6 – Définir un type avant son usage structurel

27.6 Les variables de requête ne portent pas de point d'interrogation

La syntaxe actuelle place `ask` ou `?` devant l'expression, sans employer d'identifiants tels que `?x`.

```
? .:#A(x) and .:#R(x, y) return x;
```

Listing 27.7 – Forme actuelle d'une requête

27.7 L'ordre des rôles appartient aux métadonnées de déclaration

L'annotation `context^2` appartient à la déclaration d'un rôle. Les variables quantifiées utilisent le nom du rôle sans cette annotation.

```
define .:#VisibleIn(item:#Item, context^2:#Context);
any context:#Context, exists item on .:#VisibleIn(item, context);
```

Listing 27.8 – Ordre déclaré et nom de rôle quantifié

27.8 Les définitions composites doivent contraindre leurs rôles publics

Une définition projetée peut masquer des rôles internes, mais tous ses rôles publics doivent être contraints par le corps de l'expression. Ajouter un rôle public qui n'apparaît jamais dans ce corps est invalide : l'interface exposerait une position dépourvue de contrainte.

27.9 Un joker n'est pas une contrainte imposant `nil`

Dans un motif homomorphe, un rôle omis ou non contraint signifie que toute valeur candidate est acceptée à cette position. Cela ne signifie pas que le rôle candidat doit contenir `nil`. Interpréter un joker comme un filtre sur la valeur nulle rejetterait à tort les relations renseignées.

27.10 Une recherche sur un supertype ne retourne pas les descendants bruts

Un sous-type peut présenter ses rôles dans un ordre différent ou identifier plusieurs positions grâce à un mappage polyadique. Lorsqu'une instance descendante est renvoyée par une requête sur le type parent, le runtime peut fournir une vue conceptuelle adaptée. L'identité sous-jacente demeure celle du descendant, tandis que l'ordre des positions visibles suit celui du type parent demandé.

27.11 Un mappage polyadique n'est pas une simple permutation

Un mappage polyadique normal préserve la projection, les positions répétées et les composantes d'égalité. Le réduire systématiquement à une permutation simple fait perdre des informations et peut inverser le sens apparent des accès. Un cache positionnel rapide n'est valide que s'il dérive d'une correspondance unique dans le mappage normal.

27.12 Les noms locaux de rôles ne redéfinissent pas un type connu

Lorsque l'environnement sémantique connaît déjà un type, un motif peut donner des noms locaux aux rôles de l'occurrence sans changer la définition du type.

```
1 .:#add(Amount, Tax)[result_1]
```

Listing 27.9 – Type connu avec noms de rôles locaux

Ici, `Amount`, `Tax` et `result_1` désignent les rôles de cette occurrence. Ils ne constituent ni de nouvelles métadonnées structurelles de `#add` ni des nœuds `#variable`.

27.13 Un littéral n'est pas un rôle public

Dans `.:#power(Amount, 2)[result_1]`, le littéral `2` contraint la position d'un opérande. Il n'est pas exposé comme un rôle vers lequel un autre concept pourrait établir un mappage.

27.14 L'absence de rôle commun peut interdire tout mappage valide

Un mappage polyadique inféré doit reposer sur des rôles compatibles. Si le concept source et l'expression de contrainte ne partagent aucun rôle alignable, la contrainte ne s'applique pas à cette source. N'inventez pas de mappage positionnel ni de rôle artificiel pour forcer la correspondance.

27.15 Les groupes logiques imbriqués doivent rester groupés

Les expressions `A and (B or C)` et `A or (B and C)` ne sont pas équivalentes à une liste plate de trois opérandes. Le parseur doit préserver les sous-composites imbriqués. La sérialisation canonique peut aplatir des répétitions du même opérateur, mais des opérateurs différents doivent conserver les regroupements imposés par leur sens.

27.16 Un résultat unique n'est pas forcément un conteneur à un élément

Le contrat des résultats de requête est polymorphe : un résultat unique est renvoyé directement, sans liste supplémentaire ; plusieurs résultats forment une liste chaînée et aucun résultat correspond à `nil`. Une expression `return` qui construit déjà un tuple ou une liste conserve cette forme sans enveloppe extérieure supplémentaire.

27.17 H-Logic ne remplace ni YAML, ni le XML des vues, ni functionalP

Les sources YAML déclarent les MODEL. H-Logic exprime la connaissance SEMANTIC et peut également être compilé depuis, ou vers, des structures applicatives. functionalP fournit des comportements procéduraux et peut servir de langage source pour des expressions dont la forme sémantique est compilée séparément. Maintenir ces autorités distinctes évite qu'un éditeur de sources prétende connaître l'application assemblée ou que le code procédural devienne l'unique source de vérité sémantique.

Cinquième partie

Des spécifications métier aux applications

Chapitre 28

Deux parcours, des spécifications métier aux applications logiCells

logiCells propose deux démarches complémentaires pour transformer une spécification métier en application exécutable. Toutes deux partent du langage du domaine, privilégient les concepts plutôt que les objets d'implémentation et aboutissent à des modèles déclaratifs logiCells. Elles se distinguent surtout par le *moment* où le projet introduit H-Logic et par ce qu'il souhaite valider en premier.

Ces deux parcours sont les suivants :

1. **Modèles d'abord** (*model-first*) : texte métier → modèles déclaratifs → application. Une génération facultative de H-Logic permet ensuite d'obtenir une synthèse sémantique des connaissances accumulées et d'en vérifier la cohérence.
2. **H-Logic d'abord** (*H-Logic-first*) : texte métier → H-Logic → vérification de cohérence symbolique par le moteur logiCells → squelettes de modèles générés → code d'opérationnalisation rattaché aux actions associées aux verbes de l'ontologie.

Boucle « modèles d'abord »
Texte métier → modèles déclaratifs → application → retours du runtime → synthèse H-Logic facultative → examen de cohérence → modèles affinés

Boucle « H-Logic d'abord »
Texte métier → H-Logic candidat → validation symbolique → squelettes de modèles → opérationnalisations → application → nouvelles connaissances → H-Logic affiné

FIGURE 28.1 – Les deux parcours de développement sont des boucles de rétroaction susceptibles de s'alimenter mutuellement.

Aucun des deux parcours n'est universellement préférable. Le choix dépend des incertitudes du domaine, de la quantité de connaissances déjà disponibles, de l'importance de la cohérence formelle et de la rapidité avec laquelle le projet doit disposer d'une application fonctionnelle.

Ces parcours sont également *cycliques*. Ils décrivent deux points d'entrée utiles et deux boucles de retour dominantes, non deux chaînes de traitement irréversibles. Un projet peut parcourir plusieurs fois une boucle, passer à l'autre lorsque ses risques évoluent, puis revenir à la première. Ce qui demeure stable, c'est la connaissance conceptuelle portée par la plateforme, et non l'ordre de production initial des artefacts.

28.1 Deux points d'entrée, deux boucles de rétroaction

Dans une démarche « modèles d'abord », l'apprentissage provient généralement de l'application en fonctionnement. Le modèle évolue à partir des retours des utilisateurs, des observations du runtime et des distinctions métier nouvellement identifiées. H-Logic peut ensuite synthétiser les connaissances accumulées et faire apparaître des lacunes sémantiques difficiles à déceler dans les fichiers de modèles pris isolément.

`texte métier` → `modèles déclaratifs` → `application` → `retours du runtime et du métier` → `synthèse H-Logic si utile` → `examen de cohérence` → `modèles affinés`

Dans une démarche « H-Logic d'abord », l'apprentissage vient principalement de la validation sémantique. La spécification des connaissances est affinée jusqu'à ce que son vocabulaire, ses liaisons, ses contraintes et ses déductions soient suffisamment cohérents pour produire des structures applicatives. Une fois ces structures exécutables, les retours opérationnels peuvent révéler de nouvelles connaissances et ouvrir un nouveau cycle H-Logic.

`texte métier` → `H-Logic` → `validation symbolique` → `squelettes de modèles` → `opérationnalisations et application` → `retours du runtime et du métier` → `H-Logic enrichi`

Dans les deux cas, l'équipe alterne entre *l'expression du sens*, *sa traduction en comportements exécutables* et *l'apprentissage tiré de l'exécution*. C'est un principe fondamental du développement avec logiCells.

28.2 Parcours A : les modèles d'abord, H-Logic comme synthèse sémantique

Privilégier les modèles d'abord lorsque la structure de l'application est déjà relativement bien comprise et que l'objectif principal consiste à produire un logiciel métier opérationnel avec un minimum de code.

`spécification métier` → `modèle conceptuel` → `YAML déclaratif` → `vues, actions, processus et services` → `application opérationnelle` → `synthèse H-Logic facultative`

Le développeur commence par identifier les concepts métier, leurs relations, leurs états, leurs champs, leurs actions et leurs projections. Le modèle déclaratif constitue le principal support de construction. À partir de ces déclarations, la plateforme génère ou exécute autant d'infrastructure que possible.

Lorsque l'équipe a besoin d'une représentation sémantique plus compacte, H-Logic peut être produit à partir des connaissances conceptuelles accumulées. Cette synthèse sert notamment à :

- examiner le vocabulaire et les relations répartis entre plusieurs modèles ;
- vérifier la cohérence des concepts introduits par différentes fonctionnalités ;
- rendre visibles les dépendances conceptuelles implicites ;
- interroger les connaissances à un niveau sémantique plus élevé ;
- offrir à un expert métier ou à un assistant une représentation concise du sens de l'application ;
- préparer de futures capacités symboliques ou neuro-symboliques sans les imposer dès la première itération.

L'essentiel est que H-Logic demeure *facultatif dans la boucle de construction*. Une équipe peut réaliser déclarativement une application logiCells utile avant d'avoir besoin d'une analyse logique avancée.

28.2.1 Contextes adaptés au parcours « modèles d'abord »

Ce parcours convient particulièrement lorsque :
— le domaine métier est connu et les principales entités ainsi que les processus font déjà consensus ;
— le projet doit livrer rapidement des écrans, des services ou des processus ;
— la première priorité est de remplacer le code CRUD impératif et le code d'intégration par des modèles déclaratifs ;
— l'ontologie est appelée à émerger progressivement du développement applicatif ;
— l'inférence formelle est utile, sans être indispensable au choix de la structure initiale des modèles ;
— l'équipe maîtrise bien le métier, mais possède une expérience limitée de la programmation logique.

Une application de service, une application de processus, un tableau de bord opérationnel ou un système métier fortement orienté CRUD peuvent souvent démarrer de cette façon.

28.3 Parcours B : H-Logic d'abord, les modèles comme opérationnalisation

Privilégier H-Logic d'abord lorsque l'ambiguïté sémantique et les risques d'incohérence constituent les principales difficultés du projet.

`spécification métier` → `spécification H-Logic` → `contrôles de cohérence logiCells` → `connaissances conceptuelles validées` → `squelettes de modèles générés` → `opérationnalisation des actions` → `application opérationnelle`

Le projet commence par exprimer la spécification métier en H-Logic. L'analyseur et le runtime symbolique de logiCells servent ensuite à vérifier la structure, les liaisons, les relations, les contraintes et les déductions avant de générer les artefacts applicatifs.

Lorsque la spécification sémantique est suffisamment stable, elle peut servir de point de départ à la couche des modèles déclaratifs. Le générateur produit les squelettes de concepts, de relations, de champs, d'actions et d'autres artefacts dont la dérivation est suffisamment sûre. Les choix propres à l'application, notamment la persistance, la publication, les vues et les interactions, restent des décisions explicites de génération ou de conception.

28.3.1 Verbes métier et opérationnalisation

Une distinction utile consiste à séparer *ce que signifie l'intention métier* de *la manière dont le runtime la réalise.*

Les verbes de l'ontologie expriment des intentions telles que `Validate`, `Compute`, `Reserve`, `Approve`, `Publish` ou `Notify`. Ces intentions peuvent être projetées sous forme d'actions logiCells. La déclaration d'action générée reste conceptuelle ; le code d'implémentation n'est ajouté que lorsqu'une opérationnalisation est réellement nécessaire.

`verbe de l'ontologie` → `action déclarée` → `opérationnalisation candidate` → `code d'implémentation minimal`

Le code spécifique reste ainsi à la périphérie du modèle conceptuel. Il réalise une intention métier ; il ne devient pas le lieu où sont enfouis le vocabulaire et les règles du domaine.

28.3.2 Contextes adaptés au parcours « H-Logic d'abord »

Ce parcours convient particulièrement lorsque :
— le domaine comporte de nombreuses règles, contraintes ou conclusions dérivées qui interagissent ;
— plusieurs équipes ou sources emploient des vocabulaires partiellement communs ;
— les contradictions, la portée temporelle ou les contextes doivent être identifiés avant l'implémentation ;
— l'application s'appuie sur des connaissances issues de spécifications, de documents ou de systèmes hétérogènes ;
— l'inférence symbolique, l'explication ou l'interrogation sémantique constituent des exigences centrales ;
— un LLM ou un autre système génératif participe à l'acquisition des connaissances et ses propositions doivent être validées avant de devenir exécutables ;
— les modèles applicatifs générés doivent représenter une projection d'une spécification sémantique déjà validée.

Les applications intensives en connaissances, l'aide à la décision, les domaines fortement réglementés, l'intégration sémantique et les systèmes neuro-symboliques sont des candidats naturels.

28.4 Convergence des deux parcours

Ces deux parcours ne créent pas deux architectures de plateforme. Ils convergent vers le même runtime logiCells.

Modèles d'abord	**H-Logic d'abord**
Le texte précise la fonctionnalité métier.	Le texte précise les connaissances métier.
Le YAML conceptuel est produit tôt.	H-Logic est produit tôt.
Les retours de l'application affinent le modèle.	La validation symbolique affine la sémantique.
H-Logic peut ultérieurement synthétiser les connaissances accumulées.	Les squelettes de modèles déclaratifs sont projetés après validation.
Le code n'est ajouté que pour les comportements opérationnels manquants.	Le code opérationnalise les actions et les verbes déclarés.

À leur point de convergence, les deux parcours reposent sur les mêmes principes :
— les concepts et relations portent le sens métier ;
— les métadonnées déclaratives décrivent la structure applicative ;
— les actions expriment des intentions avant de recevoir une implémentation ;
— les vues et services sont des projections du modèle conceptuel ;
— H-Logic fournit la représentation symbolique structurée lorsqu'une analyse logique est nécessaire ;
— le code spécifique est réservé aux opérations qui ne peuvent pas être obtenues déclarativement ;
— la validation symbolique demeure l'autorité pour la validité logique ;
— les capacités neuronales ou les LLM peuvent proposer, classer, hiérarchiser ou assister, sans contourner les frontières symboliques et transactionnelles du runtime.

28.5 Choisir un parcours

Le tableau suivant propose des choix de départ pratiques. Il aide à démarrer un projet ; il ne constitue pas une contrainte architecturale définitive. Un projet peut commencer par les modèles, introduire H-Logic plus tard, ou commencer par H-Logic et consacrer ensuite l'essentiel de son cycle de vie à faire évoluer ses modèles déclaratifs.

Contexte du projet	**Point de départ recommandé**	**Justification**
Processus métier connu, cycle de livraison court	Modèles d'abord	Produire rapidement une application conceptuelle fonctionnelle ; introduire la synthèse sémantique lorsque le modèle s'enrichit.
Modernisation d'une application CRUD ou de processus existante	Modèles d'abord	Remplacer progressivement la structure d'implémentation par des concepts, des actions et des projections.
Ontologie émergente issue de la découverte produit	Modèles d'abord, puis synthèse H-Logic	L'application révèle les concepts utiles ; le vocabulaire et sa cohérence peuvent ensuite être consolidés.
Domaine fortement réglementé ou comportant beaucoup de règles	H-Logic d'abord	Valider le vocabulaire, les contraintes et les déductions avant de générer les artefacts opérationnels.
Intégration de connaissances issues de sources hétérogènes	H-Logic d'abord	Résoudre les chevauchements sémantiques et les contradictions avant de définir les projections applicatives.
Acquisition de connaissances assistée par LLM	H-Logic d'abord	Traiter les connaissances générées comme des propositions H-Logic, les analyser et les valider avant toute activation ou opérationnalisation.
Prototype centré sur l'interface utilisateur et à sémantique simple	Modèles d'abord	Ne pas introduire de complexité logique avant qu'elle apporte de la valeur.
Aide à la décision exigeant explications et inférence	H-Logic d'abord	Le modèle symbolique fait partie du produit, et pas seulement d'une vue diagnostique ultérieure.

28.6 Passer d'une boucle à l'autre est normal

De nombreux projets passent volontairement d'une boucle à l'autre. Une équipe peut réaliser ses premiers modules à partir des modèles, produire une synthèse H-Logic lorsque le périmètre conceptuel s'élargit, détecter des incohérences, puis utiliser les connaissances H-Logic stabilisées pour générer une nouvelle famille de squelettes de modèles. Inversement, un projet peut commencer par H-Logic pour stabiliser un noyau réglementé ou riche en connaissances, puis faire évoluer la majorité des fonctionnalités suivantes directement dans les modèles déclaratifs

lorsque les fondations sémantiques deviennent fiables.

Un cycle mixte peut donc prendre la forme suivante :

`texte` → `modèles` → `retours de l'application` → `synthèse H-Logic` → `examen de cohérence` → `squelettes nouveaux ou régénérés` → `retours de l'application`

Le changement de boucle doit rester pragmatique :

— se tourner vers H-Logic lorsque l'ambiguïté sémantique, les contradictions, l'interaction des règles ou la cohérence entre modèles deviennent les principaux risques ;
— privilégier les itérations sur les modèles lorsque les connaissances sont suffisamment stables et que la livraison de vues, services, processus ou comportements opérationnels devient prioritaire ;
— revenir à H-Logic dès que les connaissances accumulées par l'application méritent une nouvelle synthèse sémantique.

Il ne s'agit pas d'une incapacité à choisir une méthode. C'est l'un des avantages d'un runtime conceptuel commun : structure applicative et connaissances symboliques peuvent se renforcer progressivement sans changer de plateforme ni abandonner le travail déjà réalisé.

28.7 Adapter le niveau d'expertise au projet

Pour choisir un parcours, un développeur métier ne devrait pas avoir besoin de connaître les détails internes de l'analyseur, les algorithmes d'unification, les mécanismes de classement ou les moteurs neuronaux.

Le guide propose donc la progression suivante :

— commencer par le vocabulaire et les intentions métier ;
— utiliser les modèles déclaratifs pour la structure applicative ;
— introduire H-Logic lorsque la synthèse ou la validation sémantique apporte une valeur concrète ;
— activer les fonctions symboliques à travers des modèles d'usage éprouvés plutôt qu'en manipulant les mécanismes internes du moteur ;
— activer les fonctions neuronales au moyen des capacités de la plateforme et de réglages par défaut sûrs ;
— n'exposer les détails d'implémentation que lorsque le développeur franchit volontairement une frontière d'extension.

L'objectif n'est pas de transformer chaque développeur d'applications en spécialiste de la logique ou de l'apprentissage automatique. Il consiste à lui donner accès à davantage de capacités sémantiques, symboliques et neuronales tout en maintenant l'essentiel de son travail au niveau conceptuel.

28.8 Utiliser les Skills pour entrer dans l'une ou l'autre boucle

Les Skills d'IA sont particulièrement utiles aux points d'entrée de ces cycles. Ils peuvent transformer une spécification informelle en analyse métier structurée, recommander la boucle qui apporte les retours les plus utiles à ce stade, puis appliquer la méthode de génération correspondante.

Dans un cycle « modèles d'abord », un Skill peut proposer des concepts, du YAML déclaratif, des vues, des actions et des services, puis produire ultérieurement une synthèse H-Logic destinée à l'examen de cohérence. Dans un cycle « H-Logic d'abord », un Skill peut traduire la spécification en H-Logic, la préparer à une validation par l'analyseur officiel et le runtime symbolique, puis générer des squelettes de modèles déclaratifs ainsi que les points d'opérationnalisation explicites associés aux verbes de l'ontologie.

Le développeur n'a pas à connaître le fonctionnement interne de l'unification, du classement, de la sérialisation ou des moteurs neuronaux pour utiliser ces parcours. Les Skills présentent des opérations sûres, orientées projet ; la grammaire déclarative exécutable et le moteur H-Logic restent les autorités de validation. L'équipe peut également demander à l'assistant de changer de boucle lorsque le risque dominant du projet évolue, sans reprendre le projet de zéro avec une autre méthode.

28.9 Pour aller plus loin

Pour le parcours « modèles d'abord », poursuivre avec les chapitres de construction d'applications et la référence de grammaire déclarative, notamment Chapitre 38, “Guide de décision : que créer et à quel moment ?”. Lorsque le modèle accumulé appelle une synthèse sémantique, consulter les chapitres H-Logic de Partie IV, “H-Logic et expressions déclaratives”.

Pour le parcours « H-Logic d'abord », poursuivre avec Chapitre 29, “Des spécifications métier à une ontologie H-Logic validée”, puis Chapitre 31, “De H-Logic validé aux modèles déclaratifs”. L'étude de cas de gestion des stocks illustre l'ensemble du parcours d'analyse sémantique.

Chapitre 29

Des spécifications métier à une ontologie H-Logic validée

Ce chapitre approfondit le parcours *H-Logic d'abord* présenté dans Chapitre 28, "Deux parcours vers les applications logiCells". Avant de modéliser l'application, H-Logic peut servir de représentation intermédiaire maîtrisée entre les spécifications métier et les artefacts YAML exécutables d'une application logiCells. Dans cette démarche, H-Logic n'est pas simplement un langage de règles ajouté une fois le modèle conçu : c'est le support sur lequel la signification métier est explicitée, examinée, complétée et vérifiée avant sa mise en œuvre. Un projet dont la priorité est de construire rapidement une application peut, à l'inverse, commencer par les modèles et produire ensuite une synthèse sémantique en H-Logic.

La chaîne de développement devient alors :

`spécification métier` → `analyse H-Logic` → `revue humaine` → `inférence assistée par LLM` → `validation LGC` → `modèles déclaratifs générés`

Cette méthode réduit le risque de générer des métadonnées structurellement valides à partir d'exigences ambiguës ou incomplètes. Le modèle YAML n'est produit qu'une fois les décisions sémantiques explicites et vérifiables.

Il ne s'agit pas nécessairement d'une étape initiale unique. Après la mise en œuvre et l'exécution des modèles générés, les retours métier peuvent susciter de nouveaux énoncés H-Logic, de nouvelles contraintes ou une révision du vocabulaire. Le projet peut alors reprendre l'analyse sémantique, valider les connaissances ajoutées, régénérer uniquement les squelettes concernés et poursuivre le cycle de développement applicatif. Inversement, un projet construit d'abord à partir de modèles peut rejoindre ce processus ultérieurement, en produisant une synthèse H-Logic des modèles accumulés.

29.1 Pourquoi introduire une représentation logique intermédiaire ?

Les exigences en langage naturel sont faciles à rédiger et à discuter, mais elles laissent régulièrement des questions essentielles sans réponse :

- Quels substantifs désignent des entités dotées d'une identité, et lesquels désignent des relations ou des valeurs ?
- Quels vocabulaires sont ouverts, et lesquels constituent des énumérations exhaustives ?
- Quelles conclusions dépendent du temps, du contexte ou de la provenance ?
- Quelles règles sont des définitions, des contraintes, des déductions ou des politiques opérationnelles ?
- Quels faits dérivés faut-il matérialiser pour répondre aux requêtes fréquentes ?

— Quels énoncés ont été confirmés par un expert du domaine, et lesquels ont été inférés par un assistant ?

Générer directement un fichier `.model.yaml` à partir d'une description en prose masque ces décisions à l'intérieur de l'artefact produit. H-Logic permet de les exposer sous une forme proche du vocabulaire métier, tout en se prêtant à l'analyse syntaxique, au typage, à la propagation logique et aux tests.

29.2 Trois représentations complémentaires

Le processus s'appuie sur trois représentations, chacune ayant une responsabilité propre.

29.2.1 Spécification en langage naturel

La spécification exprime les intentions dans les termes des experts métier. Elle constitue la base de la validation humaine, mais peut contenir des synonymes, des omissions, des hypothèses implicites et des énoncés dont la portée n'est pas clairement définie.

29.2.2 Spécification sémantique H-Logic

H-Logic explicite la structure du domaine :
— les types et les entités identifiées ;
— les rôles, les attributs et les relations polyadiques ;
— les vocabulaires exhaustifs et les instances ;
— les arguments temporels et contextuels ;
— les implications, les diagnostics et les projections destinées aux requêtes ;
— les faits attendus à l'issue de l'inférence.

Cette représentation convient à la revue humaine, à l'analyse assistée par LLM et à la validation par le moteur.

29.2.3 Artefacts déclaratifs logiCells

Les fichiers YAML concrétisent la sémantique validée sous forme d'artefacts du runtime :
— classes, concepts et facettes ;
— champs et chemins de rôles ;
— projections de persistance ;
— manifestes et packages ;
— vues, actions, événements, services et workflows.

Le passage de H-Logic à YAML ne se réduit pas à une traduction textuelle. Il ajoute des choix propres au runtime : persistance, publication, conception des vues et cycle de vie de l'exécution.

29.3 Les trois boucles de validation

Une analyse fiable distingue la revue humaine, l'inférence assistée par LLM et la vérification par le moteur.

29.3.1 Revue sémantique humaine

Le responsable de la revue conserve l'autorité sur la signification métier. Il confirme :
— le vocabulaire et ses définitions ;
— les objets qui doivent posséder une identité durable ;
— les participants et les rôles de chaque relation ;

— les informations de temps et de contexte associées aux faits ;
— les invariants, les exceptions et leurs conséquences métier ;
— l'objet auquel s'applique réellement chaque conclusion.

Une erreur de modélisation fréquente consiste à rattacher une conclusion à la preuve qui la justifie plutôt qu'à l'objet métier concerné. Ainsi, une position de stock peut démontrer qu'un produit doit être réapprovisionné, mais la recommandation concerne le produit, l'entrepôt et l'instant considérés.

29.3.2 Inférence assistée par LLM sous contraintes symboliques

Un LLM peut accélérer l'analyse en proposant :
— des concepts et des relations candidats ;
— les déclarations de types manquantes ;
— les rôles temporels restés implicites dans la spécification ;
— des règles de cohérence et des faits de diagnostic ;
— des motifs de requêtes et des projections indexées utiles ;
— des questions de clarification à soumettre au responsable de la revue ;
— des correspondances possibles avec les modèles logiCells.

Le LLM doit respecter l'ontologie existante, la grammaire H-Logic, les conventions de modélisation retenues et les retours du moteur. Il ne constitue pas l'autorité qui fixe la sémantique exécutable. Ses résultats restent des propositions dont l'origine doit pouvoir être retracée et la validité vérifiée.

29.3.3 Vérification syntaxique et sémantique par le moteur LGC

Le moteur fournit deux niveaux de validation distincts.

Validation syntaxique. Elle vérifie que la grammaire accepte la source : les définitions sont bien formées, les rôles et attributs respectent la syntaxe prise en charge, les implications peuvent être analysées et les commandes utilisent des formes valides.

Validation sémantique et d'exécution. Elle vérifie la signification résolue et le comportement : existence des types, compatibilité des arguments, liaison des variables, unification correcte des valeurs temporelles, production des faits attendus par propagation, cohérence des projections indexées et disparition des conclusions rétractées lorsque leur dernière justification est supprimée.

La réussite de l'analyse syntaxique ne démontre pas le bon comportement de l'ontologie. L'acceptation par l'analyseur et l'exécution logique doivent être consignées séparément.

29.4 Statut et provenance des énoncés de l'analyse

Le processus de spécification doit distinguer les énoncés de la source, les propositions de l'assistant et les résultats validés par le moteur. Un registre d'analyse simple peut utiliser trois statuts :

confirmed énoncé accepté par un responsable de la revue ou par une source faisant autorité ;

inferred énoncé proposé par un LLM ou dérivé pendant l'analyse, mais pas encore accepté comme signification exécutable ;

validated énoncé accepté par l'analyseur concerné ou par un test du moteur sémantique.

Ce registre peut lui-même être représenté de manière déclarative :

```yaml
analysis:
  - id: STOCK-RULE-003
    statement: Reserved stock cannot exceed physical stock.
    status: confirmed
    source: business-specification

  - id: STOCK-QUERY-004
    statement: Invalid positions should be searchable by warehouse and time.
    status: inferred
    proposedBy: llm-analysis
    basedOn:
      - STOCK-RULE-003
      - operational-diagnosis-needs

  - id: STOCK-PROJECTION-004
    statement: InvalidStockPositionByWarehouseTimeProduct is valid.
    status: validated
    validator: lgc-hlogic-engine
    tests:
      - invalid-position-propagation
      - temporal-index-preservation
```

Listing 29.1 – Registre d'analyse des exigences et des déductions

Le registre YAML ne remplace pas l'ontologie H-Logic. Il indique pourquoi chaque énoncé existe, qui l'a introduit et à quelle étape il a été accepté.

29.5 Une méthode de construction reproductible

La méthode suivante peut être réutilisée pour les ERP, les workflows, la gestion des risques, la planification et d'autres domaines applicatifs.

29.5.1 Étape 1 : rédiger une spécification concise

Commencez par un court texte identifiant les objets métier, les observations, les politiques, les conclusions et les invariants. À ce stade, évitez les termes d'implémentation.

29.5.2 Étape 2 : extraire et normaliser le vocabulaire

Établissez un tableau de vocabulaire. Relevez les synonymes et retenez un nom canonique pour chaque concept. Déterminez si chaque terme désigne une entité, une relation, une valeur, un événement, un état, une politique ou une conclusion.

29.5.3 Étape 3 : déterminer les besoins d'identité

N'utilisez `me` que lorsqu'un concept doit correspondre à un objet identifié.

```
define .:#StockMovement(me)[
  product:#Product,
  warehouse:#Warehouse,
  quantity:#Quantity,
  occurredAt:#Time
];

define .:#ReorderRequired(
  product:#Product,
  warehouse:#Warehouse,
  at:#Time
);
```

Listing 29.2 – Entité identifiée et relation dérivée sans identité propre

Un mouvement identifié pourra ensuite porter des références documentaires, une provenance, des corrections et un état d'audit. Le fait dérivé imposant un réapprovisionnement possède des participants et un contexte, mais ne requiert pas d'identité indépendante.

29.5.4 Étape 4 : déclarer les types fondamentaux avant leur utilisation structurelle

Déclarez les types qui contraignent les rôles et les attributs avant de construire des définitions plus riches.

```
define .:#Product(me);
define .:#Warehouse(me);
define .:#Time(me);
define .:#Quantity(me);
```

Listing 29.3 – Types fondamentaux du domaine

Cette précaution empêche un assistant ou un générateur de traiter silencieusement des concepts métier comme des valeurs non typées.

29.5.5 Étape 5 : identifier les vocabulaires métier exhaustifs

Un vocabulaire fini et explicitement fermé doit être représenté par un type complet et ses instances directes.

```
define .:#MovementKind(me);
#MovementKind::#Complete;

#Receipt:#MovementKind;
#Reservation:#MovementKind;
#Release:#MovementKind;
#Shipment:#MovementKind;
```

Listing 29.4 – Vocabulaire exhaustif des types de mouvements

La distinction est essentielle : `::` attache la propriété de complétude d'ordre supérieur, tandis que `:` déclare les instances directes.

29.5.6 Étape 6 : propager le temps et le contexte dans toute la règle

Lorsqu'une conclusion dépend d'un état temporel, la même variable de temps doit figurer dans les faits sources comme dans la conclusion.

```
.:#StockPosition(position)[
  product=product,
  warehouse=warehouse,
  at=t,
  available=available
]
and
.:#ReorderPolicy(policy)[
  product=product,
  warehouse=warehouse,
  validAt=t,
  threshold=threshold
]
and
.:#LessThan(available, threshold)
->
.:#ReorderRequired(product, warehouse, t);
```

Listing 29.5 – Propagation temporelle dans une règle de réapprovisionnement

On évite ainsi d'associer par erreur une position de stock passée à une politique future.

29.5.7 Étape 7 : exprimer les calculs par des relations auxiliaires typées

Les calculs arithmétiques ou externes peuvent être exposés sous forme de relations auxiliaires typées.

```
define .:#Subtract(
  left:#Quantity,
  right:#Quantity,
  result:#Quantity
);

define .:#LessThan(
  left:#Quantity,
  right:#Quantity
);
```

Listing 29.6 – Relations auxiliaires arithmétiques typées

L'ontologie exprime la dépendance conceptuelle. Le moteur ou une capacité enregistrée prend en charge le calcul effectif.

29.5.8 Étape 8 : dériver des faits de diagnostic positifs

Représentez l'invalidité par un fait explicite, et non uniquement par l'absence de validité.

```
define .:#InvalidStockPosition(
  position:#StockPosition,
  product:#Product,
  warehouse:#Warehouse,
  at:#Time
);
```

Listing 29.7 – Fait de diagnostic explicite

Un fait de diagnostic positif peut être interrogé, expliqué, indexé et utilisé par les vues, les services, les agents et les workflows.

29.5.9 Étape 9 : concevoir les accès aux requêtes à partir des usages réels

Utilisez les index natifs de l'hypergraphe lorsqu'ils répondent déjà aux besoins d'accès. N'introduisez des projections `#Indexed` que pour matérialiser un attribut dans un emplacement de relation, proposer un ordre différent des rôles principaux ou rendre accessibles efficacement des conclusions dérivées fréquemment interrogées.

```
define .:#ReorderRequiredByWarehouseTimeProduct(
  warehouse:#Warehouse,
  at:#Time,
  product:#Product
);

#ReorderRequiredByWarehouseTimeProduct::#Indexed;

.:#ReorderRequired(product, warehouse, at)
->
.:#ReorderRequiredByWarehouseTimeProduct(
  warehouse,
  at,
  product
);
```

Listing 29.8 – Projection indexée adaptée à un ordre de requête

Une projection indexée ne doit pas introduire de nouvelle signification métier. Elle matérialise un fait existant dans un ordre adapté à son interrogation.

29.5.10 Étape 10 : définir les déductions attendues avant la génération

Constituez de petits jeux de faits assortis de résultats attendus explicites. Ces exemples servent à la fois de support à la revue, de tests sémantiques et de critères d'acceptation pour la génération.

29.6 Le rôle des questions dans une analyse assistée par LLM

Un assistant utile ne se contente pas de générer des déclarations : il identifie les décisions qui exigent une confirmation humaine. Les questions suivantes en sont représentatives :

— Cet état doit-il posséder une identité durable ou constitue-t-il une relation dérivée ?
— Ce vocabulaire est-il exhaustif ou les projets peuvent-ils l'étendre ?
— La règle s'applique-t-elle à la date de l'événement, à celle du traitement ou à la période d'effet d'une politique ?
— Un état invalide doit-il empêcher la persistance ou rester observable sous forme de diagnostic ?
— Quels ordres de requêtes sont suffisamment fréquents pour justifier une matérialisation ?
— La conclusion porte-t-elle sur l'objet qui fournit la preuve ou sur un autre objet métier ?

Les questions sans réponse constituent elles aussi des livrables de l'analyse. La génération des modèles ne doit pas les trancher implicitement.

29.7 Points de contrôle avant génération

Le passage à la génération de l'application ne doit intervenir qu'après franchissement de points de contrôle explicites.

Contrôle 1 — Vocabulaire Les termes du domaine et les identités sont acceptés.

Contrôle 2 — Grammaire L'analyseur H-Logic courant accepte l'ontologie.

Contrôle 3 — Résolution sémantique Les types, les rôles et les références sont correctement résolus.

Contrôle 4 — Propagation Les jeux de faits représentatifs produisent les déductions et diagnostics attendus.

Contrôle 5 — Rétraction et justification Les faits dérivés restent justifiés et disparaissent correctement lorsque leur dernière justification est supprimée.

Contrôle 6 — Préparation à la génération Les décisions propres au runtime, absentes de l'ontologie, sont consignées pour la phase de génération des modèles.

Le chapitre suivant applique cette méthode à la gestion temporelle des stocks d'un ERP. Chapitre 31, "De H-Logic validé aux modèles déclaratifs" montre ensuite comment une signification validée est transposée en modèles YAML et en artefacts applicatifs logiCells.

Chapitre 30

Étude de cas : analyse temporelle des stocks ERP en H-Logic

Cette étude de cas transforme une spécification concise de gestion des stocks en une ontologie H-Logic destinée à la revue humaine, à l'enrichissement assisté par LLM, à la validation syntaxique, à la propagation logique et, ultérieurement, à la génération de modèles logiCells.

L'ontologie utilisée dans cet exemple a été acceptée par l'analyseur H-Logic. Cette acceptation valide la grammaire, mais ne démontre pas à elle seule le comportement complet de l'inférence, de l'indexation, des justifications ou des rétractions. Ces aspects nécessitent des tests du moteur.

30.1 Spécification métier

Le module gère des produits entreposés dans différents entrepôts. Il enregistre quatre types de mouvements : réception, réservation, libération de réservation et expédition. Pour chaque produit et chaque entrepôt, le système doit déterminer l'état du stock à un instant donné :

— la quantité physique disponible en stock ;
— la quantité réservée ;
— la quantité disponible.

Une politique de réapprovisionnement définit un seuil et un niveau cible pour un produit, un entrepôt et un instant donnés. Les règles métier sont les suivantes :

— le stock disponible est égal au stock physique diminué du stock réservé ;
— les quantités physique et réservée ne peuvent pas être négatives ;
— le stock réservé ne peut pas dépasser le stock physique ;
— un réapprovisionnement est nécessaire lorsque le stock disponible est inférieur au seuil applicable ;
— toutes les déductions conservent l'instant de validité des faits sur lesquels elles reposent.

30.2 Examiner le vocabulaire avant de rédiger les règles

La première revue distingue les objets métier dotés d'une identité des relations pures.

Terme	Forme proposée	Justification
Produit	entité identifiée	Identité métier durable et cible des références.
Entrepôt	entité identifiée	Participant commun aux mouvements, aux positions et aux politiques.
StockMovement	entité identifiée	Peut porter l'historique d'audit, de documents et de corrections.
StockPosition	état temporel identifié	Doit pouvoir être retrouvé, expliqué et, le cas échéant, corrigé.
ReorderPolicy	politique temporelle identifiée	Possède une période d'effet et peut évoluer indépendamment.
ReorderRequired	relation dérivée	Conclusion reliant un produit, un entrepôt et un instant.
MovementKind	vocabulaire exhaustif	La spécification actuelle définit une liste fermée.

30.3 Types fondamentaux et vocabulaire exhaustif

```
define .:#Product(me);
define .:#Warehouse(me);
define .:#Time(me);
define .:#Quantity(me);

define .:#MovementKind(me);
#MovementKind::#Complete;

#Receipt:#MovementKind;
#Reservation:#MovementKind;
#Release:#MovementKind;
#Shipment:#MovementKind;
```

Listing 30.1 – Vocabulaire fondamental de la gestion des stocks

L'emploi de `#Complete` indique que les quatre types de mouvements constituent l'intégralité du vocabulaire de cette spécification. Si un projet doit autoriser son extension par configuration, ce vocabulaire ne doit pas être déclaré complet.

30.4 Entités métier temporelles identifiées

```
define .:#StockMovement(me)[
  product:#Product,
  warehouse:#Warehouse,
  kind:#MovementKind,
  quantity:#Quantity,
  occurredAt:#Time
];

define .:#StockPosition(me)[
  product:#Product,
  warehouse:#Warehouse,
  at:#Time,
  onHand:#Quantity,
  reserved:#Quantity,
  available:#Quantity
];

define .:#ReorderPolicy(me)[
  product:#Product,
  warehouse:#Warehouse,
```

```
  validAt:#Time,
  threshold:#Quantity,
  targetLevel:#Quantity
];
```

Listing 30.2 – Entités de mouvement, de position de stock et de politique de réapprovisionnement

Le temps figure sur les mouvements, les états et les politiques, car leur signification varie au cours du cycle de vie du système ERP.

30.5 Faits dérivés et relations auxiliaires

```
define .:#ConsistentStockPosition(
  position:#StockPosition,
  product:#Product,
  warehouse:#Warehouse,
  at:#Time
);

define .:#BelowReorderThreshold(
  position:#StockPosition,
  product:#Product,
  warehouse:#Warehouse,
  at:#Time
);

define .:#ReorderRequired(
  product:#Product,
  warehouse:#Warehouse,
  at:#Time
);

define .:#InvalidStockPosition(
  position:#StockPosition,
  product:#Product,
  warehouse:#Warehouse,
  at:#Time
);

define .:#Subtract(left:#Quantity, right:#Quantity, result:#Quantity);
define .:#LessThan(left:#Quantity, right:#Quantity);
define .:#GreaterThan(left:#Quantity, right:#Quantity);
define .:#Negative(value:#Quantity);
```

Listing 30.3 – Faits dérivés et relations auxiliaires arithmétiques

Les relations arithmétiques constituent des capacités auxiliaires. Elles rendent explicite le vocabulaire des règles, tout en laissant l'implémentation du calcul numérique relever d'un moteur remplaçable.

30.6 Classification des mouvements

Le type d'un mouvement est stocké comme attribut d'un mouvement identifié. Les classifications conceptuelles spécialisées sont déduites de cette valeur.

```
.:#StockMovement(movement)[kind=#Receipt]
-> .:#ReceiptMovement(movement);

.:#StockMovement(movement)[kind=#Reservation]
-> .:#ReservationMovement(movement);

.:#StockMovement(movement)[kind=#Release]
-> .:#ReleaseMovement(movement);

.:#StockMovement(movement)[kind=#Shipment]
-> .:#ShipmentMovement(movement);
```

Listing 30.4 – Classification à partir de l'attribut de type de mouvement

Ainsi, un objet métier persistant conserve une identité de mouvement unique, tout en recevant les classifications conceptuelles dérivées de son type déclaré.

30.7 Cohérence des stocks et propagation des besoins de réapprovisionnement

```
.:#StockPosition(position)[
  product=product,
  warehouse=warehouse,
  at=t,
  onHand=onHand,
  reserved=reserved,
  available=available
]
and
.:#Subtract(onHand, reserved, available)
->
.:#ConsistentStockPosition(position, product, warehouse, t);
```

Listing 30.5 – Propagation de positions de stock cohérentes

La règle de réapprovisionnement ne combine que des faits portant sur le même produit, le même entrepôt et le même instant.

```
.:#StockPosition(position)[
  product=product,
  warehouse=warehouse,
  at=t,
  available=available
]
and
.:#ReorderPolicy(policy)[
  product=product,
  warehouse=warehouse,
  validAt=t,
  threshold=threshold
]
and
.:#LessThan(available, threshold)
->
.:#BelowReorderThreshold(position, product, warehouse, t);

.:#BelowReorderThreshold(position, product, warehouse, t)
->
.:#ReorderRequired(product, warehouse, t);
```

Listing 30.6 – Propagation temporelle des besoins de réapprovisionnement

Le fait intermédiaire est utile : il conserve la position de stock qui justifie la recommandation. La conclusion finale reste toutefois rattachée au produit, à l'entrepôt et à l'instant concernés.

30.8 Représenter les états invalides par des faits explicables

```
.:#StockPosition(position)[
  product=product,
  warehouse=warehouse,
  at=t,
  onHand=onHand
]
and .:#Negative(onHand)
```

```
-> .:#InvalidStockPosition(position, product, warehouse, t);

.:#StockPosition(position)[
  product=product,
  warehouse=warehouse,
  at=t,
  reserved=reserved
]
and .:#Negative(reserved)
-> .:#InvalidStockPosition(position, product, warehouse, t);

.:#StockPosition(position)[
  product=product,
  warehouse=warehouse,
  at=t,
  onHand=onHand,
  reserved=reserved
]
and .:#GreaterThan(reserved, onHand)
-> .:#InvalidStockPosition(position, product, warehouse, t);
```

Listing 30.7 – Déductions de positions de stock invalides

Un fait de diagnostic peut être affiché, interrogé, indexé, renvoyé par un service et relié à ses justifications logiques. La décision de bloquer également sa persistance relève d'une politique applicative distincte, fixée lors de la génération des modèles.

30.9 Déduire les index des motifs de requêtes

L'hypergraphe maintient déjà les index associés aux emplacements des relations. Des projections `#Indexed` supplémentaires se justifient lorsque des attributs d'entité doivent devenir des emplacements de relation, ou lorsqu'un accès fréquent requiert un autre ordre des rôles principaux.

Les requêtes de stock représentatives sont notamment :

— les mouvements par entrepôt, instant et type ;
— les positions de stock par produit, entrepôt et instant ;
— toutes les positions d'un entrepôt à un instant donné ;
— les alertes de réapprovisionnement par entrepôt et instant ;
— les positions invalides par entrepôt et instant.

```
define .:#StockPositionByWarehouseTimeProduct(
  warehouse:#Warehouse,
  at:#Time,
  product:#Product,
  position:#StockPosition
);

#StockPositionByWarehouseTimeProduct::#Indexed;

.:#StockPosition(position)[
  product=product,
  warehouse=warehouse,
  at=at
]
->
.:#StockPositionByWarehouseTimeProduct(
  warehouse,
  at,
  product,
  position
);
```

Listing 30.8 – Projection indexée d'un attribut vers un tuple

```
define .:#ReorderRequiredByWarehouseTimeProduct(
```

```
  warehouse:#Warehouse,
  at:#Time,
  product:#Product
);

#ReorderRequiredByWarehouseTimeProduct::#Indexed;

.:#ReorderRequired(product, warehouse, at)
->
.:#ReorderRequiredByWarehouseTimeProduct(
  warehouse,
  at,
  product
);
```

Listing 30.9 – Projection indexée d'une alerte dérivée

Le fait métier et sa projection indexée ne remplissent pas la même fonction. Le premier porte la signification ; la seconde fournit un chemin d'accès performant pour les requêtes.

30.10 Scénario manuel de propagation

Considérons les valeurs suivantes à l'instant `T1` :

```
Product    = P1
Warehouse  = W1
onHand     = 10
reserved   = 4
available  = 6
threshold  = 7
```

Listing 30.10 – Valeurs des faits d'entrée du scénario de propagation

Les capacités auxiliaires établissent :

```
.:#Subtract(10, 4, 6);
.:#LessThan(6, 7);
```

Listing 30.11 – Faits auxiliaires pour les calculs arithmétiques

Les résultats logiques attendus sont :

```
.:#ConsistentStockPosition(Position1, P1, W1, T1);
.:#BelowReorderThreshold(Position1, P1, W1, T1);
.:#ReorderRequired(P1, W1, T1);
.:#ReorderRequiredByWarehouseTimeProduct(W1, T1, P1);
```

Listing 30.12 – Faits inférés attendus

Pour un état invalide où `onHand=5` et `reserved=7`, le résultat attendu est :

```
.:#GreaterThan(7, 5);
.:#InvalidStockPosition(Position2, P1, W1, T1);
```

Listing 30.13 – Déduction attendue pour un état invalide

30.11 Liste de contrôle pour la revue métier

Avant d'accepter l'ontologie, un expert du domaine doit confirmer :

— si les positions de stock et les politiques de réapprovisionnement nécessitent des identités indépendantes ;
— si le vocabulaire des types de mouvements est réellement fermé ;
— la sémantique temporelle applicable aux politiques et aux mouvements ;
— si la disponibilité est stockée, dérivée ou les deux ;

— si les positions invalides sont conservées à des fins de diagnostic ou immédiatement rejetées ;
— si les ordres de requêtes proposés correspondent aux usages opérationnels réels ;
— si les recommandations de réapprovisionnement doivent intégrer une quantité cible, un délai d'approvisionnement ou un fournisseur.

Ce dernier point illustre pourquoi l'ontologie demeure un artefact d'analyse : de nouvelles questions métier peuvent l'enrichir avant la génération des fichiers du runtime.

30.12 Tâches d'analyse assistée par LLM

À partir de cette ontologie et de ses règles acceptées, un LLM peut contribuer à des tâches précisément délimitées :

— repérer, dans de nouvelles exigences, les termes qui ne sont pas définis dans l'ontologie ;
— proposer un rôle de temps d'effet lorsqu'un énoncé comporte une ambiguïté temporelle ;
— suggérer une relation de diagnostic correspondant à un invariant ;
— proposer un index uniquement après avoir reçu un motif de requête concret ;
— produire des faits de test et les conclusions attendues ;
— préparer des correspondances candidates vers les modèles logiCells.

Il doit signaler que ces résultats sont des propositions, tant qu'un responsable de la revue ne les a pas acceptés et que le moteur n'a pas validé le H-Logic correspondant.

30.13 Plan de tests du moteur

Une campagne complète de validation doit vérifier :

1. la classification des mouvements ;
2. l'évaluation des relations arithmétiques ;
3. l'unification temporelle entre position de stock et politique ;
4. la propagation du besoin de réapprovisionnement ;
5. la propagation des états invalides ;
6. la création des tuples indexés ;
7. la répétition de la propagation et le traitement des doublons ;
8. la coexistence de plusieurs justifications pour un même fait dérivé ;
9. la rétraction après disparition de la dernière justification ;
10. l'absence de cycles de projection capables de recréer leurs propres faits.

L'ontologie ne peut servir de base à la génération des modèles qu'avec un état de validation explicite : analyseur syntaxique accepté, résolution sémantique réussie, tests de propagation réussis, ou capacités encore manquantes documentées comme telles.

Chapitre 31

De H-Logic validé aux modèles déclaratifs

Une ontologie H-Logic validée décrit un sens exécutable, mais ne constitue pas encore une application logiCells complète. Elle relève de l'autorité `SEMANTIC`. Les fichiers YAML générés sont des artefacts `MODEL`, qui doivent faire l'objet de leur propre validation déclarative ; l'assemblage de l'`APPLICATION` intervient encore à une étape distincte. Ce chapitre explique comment une spécification sémantique validée peut guider la génération de sources sans confondre ces différentes autorités.

31.1 Opérationnaliser plutôt que traduire textuellement

La transition peut se résumer ainsi :

`concepts et règles validés` → `décisions de génération` → `artefacts MODEL candidats` → `nouvelle analyse et validation déclaratives` → `compilation APPLICATION si nécessaire`

Plusieurs constructions H-Logic possèdent des équivalents naturels dans les modèles, mais leur correspondance nécessite néanmoins des décisions explicites.

Sens en H-Logic	**Artefact candidat**	**Décision supplémentaire**
Type identifié avec `me`	Concept d'entité	`ClassId`, espace de noms et facettes.
Relation polyadique pure	Concept de relation	Persistance réifiée ou fait limité au graphe.
Attribut	Champ de modèle	Type, taille, nullabilité et `rolePath`.
Vocabulaire complet	Ensemble de concepts de type énumération	Politique d'extension et de migration.
Relation dérivée	Règle H-Logic ou relation générée	Matérialisation et cycle de vie.
Projection `#Indexed`	Relation interrogeable ou index du runtime	Stockage, rafraîchissement et exposition.
Fait diagnostique	Relation de validation, événement ou donnée de vue	Caractère bloquant ou non pour la persistance.

31.2 Générer des concepts à partir des types identifiés

Un type identifié fondamental :

```
define .:#Product(me);
```

Listing 31.1 – Type H-Logic validé

peut servir de point de départ à un modèle logiCells :

```
class:
  classId: lgc#product
  name: Product
  type: Entity

concepts:
  - Entity:
      facets:
        - main:
            type: hypergraph
            fields: []
```

Listing 31.2 – Squelette généré du modèle Product

L'espace de noms est un paramètre de génération. Il doit provenir des conventions du projet cible et ne doit pas être inventé par un LLM.

31.3 Générer un modèle d'entité temporelle

La source sémantique est la suivante :

```
define .:#StockPosition(me)[
  product:#Product,
  warehouse:#Warehouse,
  at:#Time,
  onHand:#Quantity,
  reserved:#Quantity,
  available:#Quantity
];
```

Listing 31.3 – Définition StockPosition validée

Une facette hypergraphique générée peut préserver les rôles et les attributs conceptuels :

```
class:
  classId: lgc#stock_position
  name: StockPosition
  type: Entity

concepts:
  - Entity:
      facets:
        - main:
            type: hypergraph
            fields:
              - Product:
                  kind: slot
                  type: Object
                  target: lgc#product
                  rolePath: Product
              - Warehouse:
                  kind: slot
                  type: Object
                  target: lgc#warehouse
                  rolePath: Warehouse
              - At:
                  kind: attribute
                  type: DateTime
                  rolePath: At
              - OnHand:
                  kind: attribute
                  type: Decimal
                  rolePath: OnHand
```

```
            - Reserved:
                kind: attribute
                type: Decimal
                rolePath: Reserved
            - Available:
                kind: attribute
                type: Decimal
                rolePath: Available
```

Listing 31.4 – Facette hypergraphique StockPosition générée depuis H-Logic

La source H-Logic ne détermine pas si `Available` doit également être enregistré dans une facette de base de données. La politique de génération doit décider si cette valeur est persistée, recalculée, mise en cache ou maintenue par des règles.

31.4 Générer des relations à partir des faits dérivés

La relation validée :

```
define .:#ReorderRequired(
  product:#Product,
  warehouse:#Warehouse,
  at:#Time
);
```

Listing 31.5 – Conclusion validée de réapprovisionnement

peut devenir un modèle de relation hypergraphique :

```
class:
  classId: lgc#reorder_required
  name: ReorderRequired
  type: Relation

concepts:
  - Relation:
      facets:
        - main:
            type: hypergraph
            fields:
              - Product:
                  kind: slot
                  type: Object
                  target: lgc#product
                  rolePath: Product
              - Warehouse:
                  kind: slot
                  type: Object
                  target: lgc#warehouse
                  rolePath: Warehouse
              - At:
                  kind: attribute
                  type: DateTime
                  rolePath: At
```

Listing 31.6 – Modèle de relation ReorderRequired généré

Le choix de matérialiser durablement les instances dépend de la stratégie du runtime. Une relation exclusivement inférée peut rester dans la mémoire conceptuelle, tandis qu'une exigence de reporting ou d'audit peut justifier sa persistance.

31.5 Les décisions que H-Logic ne prend pas seul

Le générateur doit recevoir une politique explicite concernant les aspects qui ne relèvent pas de l'ontologie sémantique.

```yaml
generationPolicy:
  namespace: lgc
  targetFormat: yaml

  concepts:
    StockMovement:
      persistence: database
      generateViews: true
    StockPosition:
      persistence: database
      generatedFields:
        Available:
          storage: derived-cache
    ReorderRequired:
      persistence: conceptual-memory
      publishAsService: true
    InvalidStockPosition:
      persistence: conceptual-memory
      blocksPost: false
      generateDiagnosticView: true

  indexes:
    generateFromIndexedTypes: true

  traceability:
    includeRequirementIds: true
    includeHLogicSourceLocations: true
```

Listing 31.7 – Exemple de politique de génération

Grâce à cette politique, les choix du générateur restent examinables au lieu d'être dissimulés dans un prompt de Skill ou dans un script procédural.

31.6 Deux modes de génération

31.6.1 Génération pilotée par logiCells

Une transformation déterministe du moteur utilise H-Logic validé ainsi qu'une politique de génération :

$$\texttt{H-Logic} + \texttt{politique de génération} \rightarrow \texttt{générateur de modèles} \rightarrow \texttt{artefacts YAML} \rightarrow \texttt{validation par le runtime}$$

Ce mode convient aux correspondances stables, reproductibles et entièrement spécifiées. Lorsqu'une construction sémantique ne possède pas de correspondance prise en charge, il doit produire une erreur structurée.

31.6.2 Génération assistée par LLM et Skills

Un Skill spécialisé peut produire des artefacts initiaux plus riches :
— les noms de modèles et de fichiers ;
— les manifests et les entrées de packages ;
— les premières vues et les premiers services ;
— les commentaires et la documentation ;
— plusieurs organisations possibles du runtime.

Le LLM reste tenu par les entrées validées et les grammaires cibles. Les artefacts produits doivent être compilés et vérifiés par logiCells avant d'être acceptés.

$$\texttt{H-Logic validé} + \texttt{politique de génération} + \texttt{conventions du projet} \rightarrow \texttt{Skill LLM} \rightarrow \texttt{YAML candidat} \rightarrow \texttt{vérification logiCells}$$

Le Skill génère ; il n'est pas l'autorité de validation.

31.7 Vérifier les artefacts générés

Les fichiers générés doivent passer plusieurs contrôles :

1. validation de la grammaire YAML ;
2. résolution des références de manifests et de packages ;
3. validation de `classId` et `rolePath` ;
4. compilation de chaque modèle ;
5. chargement des `ClassItems` générés ;
6. exécution des tests de propagation H-Logic ;
7. vérification que la persistance et les vues générées ne modifient pas le comportement sémantique accepté.

Un modèle valide du point de vue syntaxique peut néanmoins opérationnaliser un sens erroné. Les tests d'acceptation doivent comparer le comportement du runtime aux déductions attendues établies pendant l'analyse H-Logic.

31.8 Traçabilité de bout en bout

Chaque artefact généré devrait conserver un lien avec l'exigence et la construction H-Logic qui le justifient.

```yaml
traceability:
  - requirement: STOCK-RULE-003
    statement: Reserved stock cannot exceed physical stock.

    hlogic:
      source: stock-management.hlogic
      rule: reserved-greater-than-on-hand
      result: InvalidStockPosition

    generatedArtifacts:
      - models/StockPosition.model.yaml
      - models/InvalidStockPosition.model.yaml
      - views/InvalidStockPositions.view.xml

    validation:
      parser: passed
      semanticResolution: passed
      propagationTest: stock-invalid-003
      modelCompilation: passed
```

Listing 31.8 – Enregistrement de traçabilité de l'exigence au runtime

Cette traçabilité facilite la revue, la régénération, l'analyse d'impact et l'évolution assistée par l'IA.

31.9 Une stratégie de test à quatre niveaux

Tests de l'analyseur Vérifier que la source H-Logic est acceptée par la grammaire cible.

Tests sémantiques Vérifier la résolution des types, la compatibilité des rôles, l'unification temporelle et les dépendances valides.

Tests de propagation Vérifier les faits attendus, les index, les justifications et les rétractations.

Tests des modèles générés Vérifier la compilation, le chargement et l'équivalence comportementale des artefacts applicatifs générés.

Dans l'exemple de stock, un test avec `onHand=10`, `reserved=4`, `available=6` et `threshold=7` doit produire `ConsistentStockPosition` et `ReorderRequired`. Un test avec `onHand=5` et `reserved=7` doit produire `InvalidStockPosition`.

31.10 Artefacts de projet recommandés

Un projet utilisant ce parcours peut séparer l'analyse sémantique des fichiers applicatifs générés :

```
specification/
  requirements/
    stock-management.md
  hlogic/
    stock-management.hlogic
  analysis/
    decisions.yaml
    questions.yaml
    traceability.yaml
  tests/
    parser/
    propagation/

generation/
  policy.yaml
  reports/

src/
  models/
  views/
  services/
  manifests/
  packages/
```

Listing 31.9 – Organisation proposée des spécifications et de la génération

Le dossier de spécification n'est pas un espace de brouillon informel. Il constitue la source d'une sémantique contrôlée à partir de laquelle les artefacts applicatifs peuvent être régénérés.

31.11 Pourquoi ce parcours est important

Cette démarche change le rôle de la génération assistée par l'IA. Au lieu de demander à un LLM de déduire directement une application à partir de prose, le projet établit d'abord un contrat sémantique vérifié.

`exigences` → `sémantique contrôlée` → `inférence vérifiée` → `modèles déclaratifs générés` → `application exécutable`

Les humains valident le sens métier, les LLM accélèrent l'analyse structurée et la génération, tandis que le moteur LGC/logiCells demeure l'autorité pour la syntaxe, l'exécution sémantique et la compilation des modèles.

Sixième partie

Construire des applications

Chapitre 32

Formats de sources et conventions de projet

Cette édition adopte une convention unique pour les nouveaux développements : **YAML pour les métadonnées déclaratives au niveau MODEL, et XML uniquement pour les sources visuelles `*.view.xml`**. Cette distinction est délibérée : elle rend visible, dans le système de fichiers, la répartition des responsabilités entre comportement et présentation. Elle aligne aussi les nouveaux modules sur la chaîne de compilation actuelle de logiCells.

32.1 Règle de choix du format source

Pour les nouveaux modules, utilisez les formats suivants :

Préoccupation	**Source canonique**	**Contenu habituel**
MODEL métier	`*.model.yaml`	entités, relations, facettes, champs, actions, événements, collections, états et contraintes
MODEL multiclasse	`*.models.yaml`	plusieurs classes MODEL, notamment les contrôleurs VIEW et les classes associées aux objets affichés
Manifeste MODEL	`_.manifest.yaml`	découverte et chargement des classes au sein du package
Package	`*.package.yaml`	dépendances, répertoires de schémas, éléments de package et services
Package VIEW	`*.view.package.yaml`	responsabilité d'une arborescence `meta/views` et de ses éléments de vue
Source visuelle	`*.view.xml`	cadres, dispositions, contrôles, jeux et sources de données, curseurs et liaisons explicites vers les classes MODEL associées
Ressource de page HTTP	`*.lgp`	extension de page HTTP ; route déclarée séparément dans `virtualPages`

Lanceur du runtime	YAML / `.lgc`	`kernel.config`, `kernel.abi`, chemins du runtime, services et configuration du portefeuille
Descripteur d'application	`*.application.yaml`	assemblage de packages, stockage, modèles et services

Des sources MODEL au format XML peuvent subsister dans les anciens projets, et les outils de conversion savent encore les lire. Toutefois, ce guide ne les présente plus comme le format de référence des nouveaux développements MODEL. Lors de la migration d'une application existante, conservez la source historique le temps d'établir l'équivalence comportementale, puis publiez la représentation YAML moderne.

32.2 Convention YAML pour les champs

Le nom du champ constitue la clé YAML. Ce guide emploie systématiquement cette syntaxe simplifiée, où la clé identifie directement le champ.

```yaml
fields:
  - Code: String
  - Customer:
      type: Object
      nullable: false
  - DisplayName:
      type: String
      rolePath: Customer.Name
      nullable: true
```

Listing 32.1 – Syntaxe recommandée des champs MODEL

32.3 VIEW est volontairement différent

La source VIEW reste en XML parce que l'arborescence visuelle est hiérarchique et que la grammaire VIEW du runtime repose sur XML. Une vue moderne n'est pas un formulaire monolithique : elle résulte de l'assemblage des éléments suivants :

```
CustomerEdit.view.xml      # visual structure, datasets, view bindings
CustomerEdit.models.yaml   # Page / Master / component controllers
_.manifest.yaml   # discovery of the MODEL-behind classes
<module>.view.package.yaml # ownership of the view tree
```

Listing 32.2 – Répartition des sources d'une vue moderne

La source XML référence explicitement les classes MODEL associées par `source`, `class` et `classId`. Ces classes sont compilées par la chaîne MODEL habituelle ; la compilation VIEW ne crée pas un deuxième compilateur MODEL.

32.4 Règle de migration des formulaires et cadres historiques

Les fichiers historiques `*.form` et `*.frame` deviennent des fichiers `*.view.xml`. Si une conversion intermédiaire a perdu la disposition, les liaisons ou la structure des contrôleurs, revenez au fichier `.form` ou `.frame` d'origine : il demeure l'autorité visuelle.

La migration se décompose en trois résultats vérifiables indépendamment :

1. **Parité visuelle** : la disposition significative, les composants, les jeux de données et leurs liaisons sont conservés dans `*.view.xml` ;

2. **Parité MODEL associé** : les zones de page, contrôleur et modèle sont externalisées dans `*.models.yaml` sans perte de comportement ;
3. **Parité des liaisons** : chaque référence XML correspond à une seule classe MODEL déclarée, avec préservation du `classId` attendu.

32.5 Ressources d'extension de pages HTTP

Les ressources de pages HTTP portent l'extension `.lgp`, par exemple `Users.lgp`. Il faut les distinguer des sources visuelles `*.view.xml`, des lanceurs du runtime `.lgc` et des descripteurs de services HTTP. Une route HTTP virtuelle est déclarée dans la définition du service, sous `virtualPages` ; la présence d'un fichier `.lgp` n'active pas, à elle seule, un point d'entrée HTTP. Un exemple au niveau du service figure dans Chapitre 48, "Services déclaratifs du runtime et modèles d'exécution".

32.6 Ne pas confondre les autorités

MODEL fait autorité pour les sources déclaratives ; APPLICATION, pour les métadonnées assemblées ; SEMANTIC, pour le graphe sémantique et H-Logic ; VIEW, pour les sources visuelles et les références explicites vers les classes MODEL associées. L'outil ou le développeur doit toujours savoir quelle autorité il consulte ou modifie.

Chapitre 33

Manifestes de classes : chargement, modes et conventions

Un manifeste de classes constitue la liste des points d'entrée du runtime pour un dossier de modèles. Ce n'est pas un simple index de répertoire : il indique au chargeur les classes conceptuelles susceptibles de devenir des objets `ClassItem`, les identifiants qu'elles exposent et les déclarations locales relevant du périmètre de chargement du package.

33.1 Ce que garantit un manifeste

Le manifeste définit explicitement le contrat de chargement du runtime :

```yaml
package:
  plugginCaption: TaskManagement
  uri: http://www.logicells.net/package/TaskManagement
  classes:
    - Task:
        classId: lgc#task
        type: Entity
```

Listing 33.1 – Manifeste de classes YAML minimal

La seule présence du fichier `Task.model` sur le disque ne suffit pas. La classe doit également être déclarée dans le manifeste et accessible depuis un package actif.

33.2 Attributs du manifeste

Attribut	Rôle	Recommandation
`classId`	Identifiant conceptuel stable	Considérez-le comme une identité publique. Ne le modifiez pas à la légère dès lors que des données, des relations ou des intégrations en dépendent.
`name`	Nom de classe dans le runtime et les API publiques	Maintenez la cohérence avec la déclaration de classe et le chemin complet employé par les collections et les actions.
`type`	Catégorie générale de classe	Utilisez une catégorie prise en charge par la déclaration du runtime, par exemple une entité ou une relation. Ne la déduisez pas uniquement du nom de fichier.

`fileName`	Conteneur source physique	Facultatif lorsque chaque classe possède son propre fichier ; nécessaire lorsqu'une entrée du manifeste se trouve dans un conteneur partagé `*.models.yaml`. Cet attribut désigne le fichier, tandis que le nom de l'entrée du manifeste sélectionne la classe à l'intérieur.
`mode`	Mode facultatif de chargement ou d'interprétation	Les valeurs dépendent de la version du runtime. Ne reprenez une valeur que d'un modèle validé de la même génération et documentez sa nécessité. En l'absence de besoin particulier, préférez le comportement par défaut sans `mode`.
`plugginCaption`	Libellé descriptif du package	Choisissez un nom de package stable et intelligible. Il facilite l'inspection et la traçabilité sans se substituer à l'activation du package.
`uri`	Identité d'espace de noms ou de document	Adoptez une convention d'URI stable par produit ou projet. L'URI organise l'identité ; celle de la classe exécutable demeure déterminée par `classId` et par la déclaration effectivement chargée.

33.3 Chargement d'une classe depuis un conteneur `.models` multiclasse

Un fichier `*.models.yaml` est un conteneur physique réunissant plusieurs classes MODEL ordinaires. Le manifeste maintient la séparation entre l'identité logique d'une classe et son stockage physique.

```yaml
package:
  uri: view#meta.views.Common.Folder
  classes:
    - FieldPanelSearch:
        classId: view#meta.views.Common.Folder.List.FieldPanelSearch
        fileName: List.models.yaml
        type: Entity
    - GridPanel:
        classId: view#meta.views.Common.Folder.List.GridPanel
        fileName: List.models.yaml
        type: Entity
```

Listing 33.2 – Deux entrées de manifeste chargées depuis un même conteneur de modèles

Le chargeur résout l'entrée du manifeste, ouvre `List.models.yaml`, sélectionne par son nom la classe demandée dans le conteneur `classes:`, puis transmet cette classe au chargeur MODEL standard. Il ne s'agit pas d'un format de modèle VIEW distinct.

Chaque dossier de modèles exposant des classes chargeables utilise le fichier de manifeste `_.manifest.yaml`. Les chemins de schéma des packages actifs déterminent quels manifestes sont chargés.

33.4 Choisir le préfixe d'un identifiant de classe

Les concepts de plateforme utilisent habituellement le préfixe `lgc#`. Les concepts applicatifs doivent utiliser un préfixe propre au projet plutôt que de s'approprier celui de la plateforme. Les trois propriétés essentielles sont l'unicité, la stabilité et la maîtrise de l'identifiant.

```yaml
platformConcept: lgc#entity
applicationConcept: tsk#task
applicationRelation: tsk#task_belongs_to_project
```

Listing 33.3 – Séparation des identifiants de plateforme et d'application en YAML

Consignez tout nouveau préfixe dans les conventions du projet, puis réutilisez-le de manière cohérente dans les modèles, les relations, les tests et les contrats de services.

33.5 Doublons et références entre dossiers

Une entrée de classe répétée peut être un doublon accidentel, une substitution historique ou un mécanisme d'extension délibéré. Ne présumez pas de sa signification : vérifiez le résultat effectivement obtenu au runtime.

1. comparez les deux déclarations et leurs chemins source ;
2. déterminez quel package et quel manifeste sont chargés en premier ;
3. vérifiez si le runtime signale un remplacement, une extension ou une duplication ;
4. résolvez la classe par `classId` et examinez son nom complet au runtime ;
5. supprimez le doublon, sauf si le comportement d'extension est explicitement nécessaire et couvert par des tests.

Certains projets anciens ou spécialisés autorisent un manifeste à référencer des déclarations situées hors de son dossier. Pour un nouveau développement, préférez un manifeste local exposant les déclarations locales et laissez la configuration des packages assembler les dossiers. Le chargement entre dossiers doit constituer un choix d'architecture explicite, non un raccourci invisible.

33.6 Unicité globale des identifiants de classe

Un `classId` désigne une identité conceptuelle à l'échelle de l'application assemblée ; il ne s'agit pas d'un nom local au fichier. Deux déclarations indépendantes et non atomiques ne doivent pas réutiliser silencieusement le même identifiant de plateforme ou d'application.

Pour les nouveaux modules, appliquez la règle suivante :

```
un sens unique       -> un classId stable
un autre sens         -> un autre classId
remplacement voulu    -> mecanisme explicite de redefinition
```

Listing 33.4 – Règle d'identité des classes

Ainsi, un agent métier et un participant contextuel interne ne devraient pas partager un identifiant au seul motif que le langage courant les désigne tous deux comme des « agents ». Des concepts distincts exigent des identités distinctes.

Lorsqu'un doublon est détecté, examinez toute la chaîne de chargement des packages et manifestes avant de modifier les rôles sémantiques. Les anciens projets peuvent comporter des redéfinitions volontaires ; elles doivent être explicites et testées, et non déduites de l'ordre de chargement.

33.7 Événements dans les manifestes

Certains manifestes historiques comprennent des déclarations d'événements ou des hooks de chargement. Ce ne sont pas des événements métier ordinaires : ils interviennent dans le chargement ou la spécialisation des packages et des modèles. Pour les nouveaux modules, placez les événements métier dans le modèle concerné et ne recourez à un comportement au niveau du manifeste que si le contrat du runtime l'exige explicitement.

33.8 Validation recommandée

```
1. L'application charge le package attendu.
2. L'element du package est actif.
3. Le manifeste est trouve et compile.
4. Le fichier de modele est trouve et compile.
5. Le ClassItem est resolu par classId.
6. Le nom complet resolu correspond au modele attendu.
```

Listing 33.5 – Séquence de validation d'un manifeste

Si l'une de ces étapes échoue, diagnostiquez d'abord la couche correspondante avant d'examiner les vues, les services ou les actions métier.

Chapitre 34

Modèles d'application

Le modèle d'application constitue la frontière d'assemblage entre les concepts métier, la configuration du runtime, les vues, les actions, les services et les packages. Ce n'est pas un simple fichier de projet : il indique comment le runtime conceptuel doit charger et interpréter un ensemble cohérent de modèles pour constituer une application exécutable.

34.1 Rôle d'un modèle d'application

Un modèle d'application logiCells rassemble plusieurs préoccupations : le graphe ou hypergraphe conceptuel, les alias de bases de données, les packages à charger, les services disponibles, les profils de publication desktop ou web, la langue et les paramètres de localisation par défaut, ainsi que les conteneurs du runtime.

Il ne doit pas masquer la sémantique métier. Les concepts métier restent déclarés dans les fichiers `.model.yaml` ou `.model.yaml`. Le modèle d'application se charge de leur assemblage.

34.2 Espaces de noms

Les concepts de la plateforme utilisent généralement `lgc#`. Les exemples applicatifs peuvent utiliser `lbs#`. Ces deux espaces de noms peuvent coexister, à condition que leurs responsabilités soient nettes : les concepts de plateforme apportent une sémantique réutilisable du runtime ; les espaces de noms applicatifs portent les concepts propres au métier.

34.3 Règles de conception

Concentrez le modèle d'application sur l'assemblage. Placez les objets métier dans les fichiers de modèles et les mécanismes de plateforme réutilisables dans leurs packages. Déclarez explicitement les services et les packages. Évitez de confondre l'identité visuelle propre à un client avec une architecture applicative destinée à être réutilisée.

34.4 Synthèse

Un modèle d'application est l'assemblage exécutable d'une application conceptuelle. Il relie les packages, les ressources du runtime, la persistance et les contextes de publication sans devenir lui-même le modèle métier.

Chapitre 35

Collections et vues conceptuelles

Collections et vues répondent à deux aspects d'un même besoin de navigation. Une collection détermine ce que l'on peut parcourir à partir d'un concept ; une vue précise comment présenter une partie de ce parcours à un utilisateur, à un service ou à un agent.

35.1 Les collections comme parcours porteurs de sens

Une collection doit correspondre à un chemin d'accès significatif : les phases d'une tâche, les messages d'une conversation, les rôles d'un utilisateur ou les éléments en attente dans une file. Elle ne doit pas être considérée comme un raccourci SQL arbitraire.

35.2 Les vues comme projections

Une vue conceptuelle projette un concept dans une interaction. Elle peut recourir à des champs, listes, boutons ou grilles ; il s'agit cependant de composants fonctionnels. Le rendu concret est choisi en enregistrant une implémentation adaptée à la plateforme cible.

35.3 Collections, curseurs et contexte

Lorsqu'une vue utilise `collectionPath`, elle demande au runtime d'évaluer une collection dans le contexte courant. Ce contexte peut être l'objet sélectionné, le curseur actif, l'utilisateur courant ou l'état du processus en cours.

35.4 Règles de conception

Définissez les collections avec le vocabulaire du modèle. Utilisez des jointures lorsque la relation doit elle-même être parcourue. Reliez les vues aux collections et aux curseurs plutôt que de dupliquer le code d'accès aux données. Distinguez l'apparence visuelle de la déclaration du composant fonctionnel.

35.5 Synthèse

Les collections rendent le modèle navigable ; les vues rendent cette navigation utilisable. Toutes deux doivent préserver le sens conceptuel.

Chapitre 36

Les actions comme intentions métier

Dans logiCells, une action n'est pas qu'un bloc de code. C'est une capacité déclarée d'un concept. Son nom, le concept qui la porte, ses paramètres, sa visibilité et ses règles de routage expriment d'abord une intention métier, avant de décrire un choix de mise en œuvre.

Cette distinction est fondamentale. Un script peut implémenter une action, mais ne constitue pas l'action elle-même. Il en va de même d'un composant natif. L'action reste le point d'entrée conceptuel qui indique ce que le concept est capable de faire.

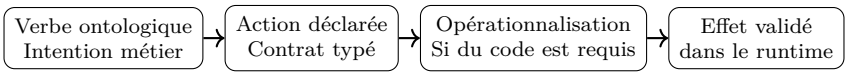


FIGURE 36.1 – Le code reste à la frontière d'opérationnalisation ; la capacité métier demeure nommée et typée de manière déclarative.

36.1 Pourquoi les actions appartiennent aux concepts

Un objet métier doit exposer les opérations qui ont un sens pour lui. Un message peut proposer `NewMessage`, une conversation `StartConversation`, un élément de travail `Validate`, et un processus `Start`, `Finalize` ou `Retry`.

La déclaration rend cette intention explicite :

```yaml
actions:
- action:
    name: Start
    type: ObjectItem
    published: true
    params:
    - Result:
        dataType: Boolean
        paramType: ptReturn
```

Listing 36.1 – Déclaration d'une action comme intention métier

Le modèle indique que l'objet peut être démarré. L'implémentation détermine comment ce démarrage s'effectue.

36.2 Actions d'instance et actions de classe

Les actions d'instance s'appliquent à un objet déterminé. Elles emploient généralement `Self` pour désigner l'instance conceptuelle courante. Elles conviennent lorsque le comportement dépend de l'état de cet objet.

Les actions de classe portent sur le concept dans son ensemble. Elles conviennent à la recherche, à la création, à la validation, à la résolution de rôles ou aux opérations de type fabrique. Une action de classe peut, par exemple, retrouver un objet par son nom, créer un rôle absent ou renvoyer le réseau par défaut de l'espace de noms de l'application.

36.3 Implémentations scriptées et natives

De nombreux exemples utilisent des actions scriptables. Elles rapprochent l'orchestration du modèle et rendent les scénarios plus faciles à comprendre. Cependant, logiCells n'impose pas de scripter toutes les actions : une classe de runtime enregistrée, un adaptateur de service, un composant de messagerie ou un composant visuel peuvent également les implémenter. Lorsqu'un script `functionalP` implémente une action, il n'en représente que le corps d'exécution. Les bases pratiques de ce langage sont présentées au Chapitre 37, "Premiers pas avec functionalP dans logiCells".

La déclaration d'une action doit donc être comprise comme un contrat stable :

Ici, `Verify` exprime l'intention ; la classe d'implémentation fournit le mécanisme d'exécution.

36.4 Les paramètres comme frontières conceptuelles

Les paramètres font partie de la signification publique d'une action. Ils définissent ses entrées et ses sorties. Il peut s'agir de types primitifs, comme `String` ou `Boolean`, ou de types conceptuels, tels que `lgc#contact`, `lgc#conversation` ou `lbs#work_item`.

Des paramètres bien conçus rendent l'orchestration explicite et facilitent la publication ultérieure de l'action comme service, son appel depuis un workflow ou son association à un composant de vue.

36.5 Actions publiées

L'attribut `published` exprime une décision de visibilité. Une action publiée appartient à la surface d'appel du concept. Une action non publiée correspond généralement à un auxiliaire interne, à un gestionnaire d'événement ou à une étape privée de l'implémentation.

Réservez les actions publiées aux capacités métier stables ; employez les actions non publiées pour le routage interne, les callbacks de vue et les étapes intermédiaires d'un processus.

36.6 Actions au sein des processus

Un objet processus coordonne les actions de plusieurs objets participants. Il en découle une règle de conception simple :

```
If one object owns the behavior:
  declare the action on that object.

If several objects must collaborate:
  declare a process model and put the coordination action there.
```

Listing 36.2 – Choisir où déclarer une action de processus

L'action d'un processus peut appeler de manière synchrone des actions locales ou émettre des événements de manière asynchrone. Dans les deux cas, la coordination reste explicite dans le modèle.

36.7 Actions et événements

Actions et événements sont complémentaires. Un événement signale qu'un fait significatif s'est produit ou qu'une demande a été émise. Une action décrit ce que fait un concept en réponse.

Une action de processus peut émettre un événement :

```
Event := Self.ObjectsContainer.BuildMessage('lbs.Process.Start');
Self.ObjectsContainer.SendEvent(Event, Self.EventProxy);
```

Listing 36.3 – Action émettant un événement

Un contrôleur peut s'abonner à cet événement et le diriger vers une action :

Cette séparation évite les dépendances cachées entre le processus et le contrôleur.

Chapitre 37

Premiers pas avec functionalP dans logiCells

Les métadonnées MODEL de logiCells sont écrites en YAML ; les sources visuelles VIEW utilisent des fichiers `*.view.xml`. Parfois, le modèle a également besoin d'un petit fragment exécutable : initialiser l'état d'une page, valider un dataset avant son enregistrement, créer un objet, actualiser une vue parente ou appeler une action du runtime. Dans ces situations, les métadonnées utilisent fréquemment des scripts `functionalP`.

Ce chapitre n'est pas une référence exhaustive du langage. Il présente le sous-ensemble pratique dont un débutant a besoin pour lire les scripts du premier tutoriel de gestion des tâches et des vues alimentées par des datasets. La référence linguistique complète figure dans Chapitre 91, "Éléments essentiels du langage functionalP". Une cartographie des sources est disponible pour les mainteneurs dans le dépôt, mais elle ne fait volontairement pas partie du parcours destiné aux développeurs métier.

37.1 Où rencontre-t-on functionalP ?

Un débutant rencontre généralement `functionalP` à quatre endroits :

```
script d'action
  Implemente une action logiCells declaree.

gestionnaire d'evenement pageModel
  Reagit aux evenements de page, comme OnShow.

gestionnaire d'evenement de dataset
  Reagit au cycle de vie d'un dataset, comme BeforeDataSetOpened
  ou BeforeDataSetPosted.

liaison de service ou d'integration
  Appelle une capacite publiee ou prepare une frontiere requete/reponse.
```

Listing 37.1 – Emplacements habituels des scripts functionalP

La règle essentielle est la suivante : `functionalP` n'est pas la syntaxe du modèle. MODEL reste en YAML. Le script est un fragment exécutable associé à une action ou à un gestionnaire d'événement déclaré.

37.2 Script minimal dans les métadonnées

Un script est normalement stocké dans une action. Le MODEL YAML environnant déclare l'action ; le script en fournit le corps exécutable.

En pratique, de nombreux scripts n'affectent pas explicitement `Result`. Ils produisent un effet, tel que l'initialisation d'un dataset ou l'enregistrement d'un objet. Dans les exemples pour débutants, utilisez `Result` lorsqu'une action possède une valeur de retour attendue : l'intention sera plus facile à diagnostiquer.

37.3 Variables et affectation

L'opérateur d'affectation est `:=`. Les variables sont souvent introduites directement par une affectation :

```
1 lTitle := 'New task';
2 lPriority := 1;
3 Result := lTitle;
```

Listing 37.2 – Variables locales

Choisissez des noms de variables locales qui rendent les objets du runtime reconnaissables. Une convention courante consiste à préfixer les variables locales par `l` : `lDs`, `lTask`, `lMem`, `lEvent`.

37.4 Self, Sender et Result

La plupart des scripts de métadonnées logiCells utilisent trois noms du runtime :

Nom	Signification habituelle
`Self`	Objet courant associé à l'action ou à la page. Il permet généralement d'accéder à `ObjectsContainer`, `PageControler` et à l'objet conceptuel courant.
`Sender`	Objet ayant déclenché l'événement. Pour les événements de dataset, il s'agit souvent du dataset en cours d'ouverture, d'enregistrement ou d'actualisation.
`Result`	Valeur de retour explicite du script lorsque le contrat de l'action en attend une.

Une erreur classique consiste à considérer que `Sender` est toujours un composant visuel. Pour les événements de dataset, c'est normalement un dataset. Pour les événements visuels, il peut s'agir du composant graphique ou de son objet sous-jacent (*object-behind*).

37.5 Accéder à PageControler

`PageControler` est l'objet local du runtime reliant le modèle de page, le cache de données de la vue et l'arborescence des composants visuels.

```
1 lDs := Self.PageControler.DataSetsManager.DataSetByName('TaskEntityDs');
```

Listing 37.3 – Accéder à un dataset de page depuis PageControler

Utilisez `PageControler` lorsque le script appartient à une page, une vue, l'objet sous-jacent d'un composant ou un événement de vue lié à un dataset. Il ne remplace pas la modélisation métier : c'est un objet de coordination locale de la page courante.

37.6 Lire un dataset

`DataSetsManager` donne accès aux datasets déclarés dans le cache de données de la page.

```
lDs := Self.PageControler.DataSetsManager.DataSetByName('TaskEntityDs');
lTask := lDs.ObjectItem;
```

Listing 37.4 – Lire le dataset de la tâche en cours d'édition

Ce code ne requête pas directement une base de données : il lit la surface de données courante de la page. Le fournisseur et la collection associés au dataset restent extérieurs au composant visuel.

37.7 Initialiser l'état local d'une page

Le premier tutoriel de gestion des tâches initialise un dataset en mémoire à l'ouverture de la page. C'est un bon exemple d'introduction, car il distingue les données métier de l'état de la page.

```
lMem := Self.PageControler.DataSetsManager.DataSetByName('TaskMemoryDs');
if lMem.IsEmpty then begin
  lMem.Append();
  lMem.OnlyOpen := True;
  lMem.PostData();
end;
```

Listing 37.5 – Initialiser un dataset de filtrage local

`TaskMemoryDs` n'est pas une table persistante de tâches. Il stocke l'état des filtres locaux de la page. Si cette valeur doit exister au-delà de la page, modélisez-la comme un champ de concept ou une facette persistante.

37.8 Filtrer avant l'ouverture d'un dataset

Les événements de dataset permettent à une vue d'ajuster sa surface de données avant leur lecture par les composants.

```
if Sender.Name = 'TaskListDs' then begin
  Sender.SetOrdering('Priority desc, DueDate asc');
  lMem := Self.PageControler.DataSetsManager.DataSetByName('TaskMemoryDs');
  if lMem.OnlyOpen then
    Sender.AddStringFilterItem('StatusName','<>','Done');
  if lMem.SearchText <> '' then
    Sender.AddStringFilterItem('Title','contains', lMem.SearchText);
end;
```

Listing 37.6 – Filtrer un dataset avant son ouverture

Ce script exprime une règle de données de vue. Il ne doit pas contenir de politique métier fondamentale. Si cette règle doit s'appliquer partout, déplacez-la vers une contrainte du modèle, une action ou une frontière de service.

37.9 Enregistrer l'objet courant

Une action de sauvegarde lit généralement le dataset courant, récupère son objet et l'enregistre.

```
lDs := Self.PageControler.DataSetsManager.DataSetByName('TaskEntityDs');
lTask := lDs.ObjectItem;
lTask.PostData();
```

Listing 37.7 – Enregistrer l'objet courant

L'enregistrement des données est une opération intentionnelle du runtime. La modification d'un champ à l'écran ne signifie pas nécessairement que l'objet conceptuel a été persisté ou que le changement a été propagé.

37.10 Créer un objet

Les actions de classe ou de fabrique créent souvent un nouvel objet par l'intermédiaire du conteneur d'objets.

```
lClassTask := Self.ObjectsContainer.ClassItemByName('TaskManagement.Task');
lTask := Self.ObjectsContainer.CreateObjectItem(lClassTask, 1, '');
lTask.SetValue('Title', 'New task');
lTask.PostData();
```

Listing 37.8 – Créer un nouvel objet Task

Dans les métadonnées de production, privilégiez une action de création déclarée lorsqu'elle existe. La création directe d'un objet par script est utile pour les tutoriels et l'orchestration interne ; les workflows publics doivent cependant exposer des actions stables.

37.11 Dispatcher un événement visuel

Après l'enregistrement d'un objet, une page peut devoir informer sa vue parente qu'une liste doit être actualisée.

```
lEvent := Self.PageControler.ParentPageControler.CreateVisualEvent(
  'lgc.Visual.View.Refresh', nil, nil,
  Self.ObjectsContainer.MetaDataContainer, false);
Self.PageControler.ParentPageControler.DispatchVisualEvent(lEvent, StopNotification);
```

Listing 37.9 – Dispatcher un événement d'actualisation visuelle au parent

Les événements visuels servent à la coordination de l'interface. Ne les utilisez pas comme bus d'événements métier dissimulé. Si un fait métier a changé, représentez-le par une action, un événement métier ou une transition de processus.

37.12 Appeler l'objet sous-jacent d'un composant visuel

Certains scripts doivent appeler l'objet sous-jacent local d'un composant de vue.

```
Self.PageControler.GetVisualComponentObjectBehindByName('Master').CloseForm();
```

Listing 37.10 – Fermer la vue principale courante

Il s'agit d'une opération locale d'interface. Elle appartient à une action de page, pas à une action de modèle de domaine destinée aux services ou aux exécutions en arrière-plan.

37.13 Quand déplacer la logique ailleurs ?

Un script `functionalP` doit rester court. Déplacez sa logique lorsqu'il devient une application dissimulée dans les métadonnées.

Si le script...	**Placez la logique dans...**
Impose une règle métier partout	une contrainte du modèle, une action de validation ou une étape de processus

Si le script...	Placez la logique dans...
Coordonne plusieurs objets dans la durée	un modèle de workflow ou de processus
Publie une capacité externe stable	une action déclarée et un contrat de service
Modifie uniquement l'état local de la vue	le modèle de page ou un gestionnaire d'événement de dataset
Actualise ou route seulement des composants visuels	un événement visuel ou un modèle de cadre réactif

37.14 Erreurs fréquentes chez les débutants

Erreur	Correction
Prendre `functionalP` pour le langage du modèle	Conserver les métadonnées MODEL en YAML ; n'utiliser `functionalP` qu'à l'intérieur de fragments exécutables déclarés.
Placer une politique métier dans un événement de vue	Définir les politiques durables dans les concepts, contraintes, actions ou workflows.
Supposer que `Sender` a toujours le même type	Vérifier la famille d'événements : dataset, visuel, processus, service ou message.
Appeler un composant depuis une action de service	Les appels aux objets sous-jacents de l'interface appartiennent aux actions locales de page, pas aux frontières de service.
Oublier d'enregistrer les données	Renseigner les valeurs de l'objet courant, puis appeler `PostData()` lorsque la modification doit être validée.
Laisser la valeur de retour implicite	Affecter `Result` lorsque le contrat de l'action attend une valeur.

37.15 Parcours de lecture pratique

Pour un premier projet, apprenez `functionalP` dans cet ordre :

1. lire les variables et les affectations ;
2. comprendre `Self`, `Sender` et `Result` ;
3. accéder à `PageControler.DataSetsManager` ;
4. lire l'objet courant d'un dataset ;
5. enregistrer les données ;
6. dispatcher un événement visuel local ;
7. créer un objet seulement après avoir compris les parcours de chargement et de vue.

Chapitre 38

Guide de décision : que créer et à quel moment ?

Ce chapitre rassemble les règles de décision auparavant répétées dans plusieurs parties de l'ouvrage. Les chapitres suivants approfondissent la théorie et les détails du runtime, mais ce guide répond d'abord à la question pratique du débutant : *que faut-il créer ensuite ?*

Utilisez-le comme une carte d'orientation. Lorsqu'un chapitre explique que les concepts précèdent le code, que les vues sont des projections ou que les événements représentent des changements observables, vous trouverez ici la réponse opérationnelle.

38.1 La chaîne de décision centrale

Pour ajouter une fonctionnalité, ne partez pas de l'écran, de la table ou de la route de service. Commencez par la plus petite intention conceptuelle qui doit conserver son sens dans la durée.

```
besoin
-> concept ou relation
-> facettes et champs
-> collections
-> projection de vue ou d'API
-> actions
-> evenements
-> workflow, service ou cadre reactif si necessaire
-> tests et diagnostics
```

Listing 38.1 – Ordre de conception pratique d'une nouvelle fonctionnalité

Cet ordre n'est pas un modèle en cascade rigide, mais un contrôle de cohérence. Si une vue, un service ou un script devient difficile à expliquer, revenez au concept, à la relation ou à l'action qui devrait porter le sens.

38.2 Choisir le parcours de construction avant les artefacts

Avant de décider quel artefact de modèle créer, choisissez comment le projet passera du besoin métier exprimé en langage naturel à une connaissance exécutable. logiCells propose deux parcours de départ de premier plan, décrits dans Chapitre 28, "Deux parcours vers les applications logiCells".

— **Modèles d'abord (Model-first) :** à privilégier lorsque le domaine est suffisamment compris et que la priorité est d'obtenir rapidement une application déclarative fonctionnelle. Produisez ensuite du H-Logic si une synthèse sémantique compacte ou un contrôle de cohérence devient utile.

— **H-Logic d'abord (H-Logic-first) :** à privilégier lorsque l'ambiguïté sémantique, l'interaction des règles, la diversité des vocabulaires ou la cohérence formelle constituent les principaux risques. Validez d'abord la connaissance, générez ensuite les squelettes de modèle et opérationnalisez les verbes métier nécessaires sous forme d'actions.

Un projet peut alterner plusieurs fois entre ces parcours. La décision porte sur la *boucle de retour la plus adaptée à l'instant présent*, pas sur deux architectures applicatives incompatibles. L'approche Model-first privilégie le retour de l'application en fonctionnement ; l'approche H-Logic-first privilégie celui de la validation sémantique. Réévaluez le choix lorsque le risque dominant du projet change.

38.3 Tableau de décision

Créer	**Quand ?**	**À éviter lorsque...**
Concept	Le vocabulaire métier exige une entité stable possédant identité, champs, contraintes, actions ou vues.	Il ne s'agit que d'une valeur temporaire d'interface, d'un filtre local ou d'une valeur calculée pour l'affichage.
Relation	Le sens lie au moins deux participants et doit pouvoir être interrogé, contraint ou conservé comme fait.	Une simple propriété descriptive suffit et le lien n'a pas de signification propre.
Facette	Un même concept exige plusieurs projections : mémoire, persistance, recherche, import, API externe ou données calculées.	Les champs appartiennent exclusivement au cache d'une vue ou à l'état transitoire d'un composant.
Champ	Le modèle requiert une position nommée dans un concept ou une facette. Respectez les conventions du projet pour les slots et attributs.	La valeur n'est qu'un état de widget : « déplié », « onglet sélectionné », « saisie en cours », etc.
Collection	Le runtime doit parcourir, énumérer, filtrer, joindre ou publier un ensemble d'objets.	Il suffit de rechercher ponctuellement un objet, sans contrat de navigation réutilisable.
Vue	Un concept ou une collection doit être projeté vers l'utilisateur, une arborescence de composants ou une interface publiée.	La structure est uniquement une règle serveur ou un service sans projection pour l'utilisateur.
DataSet / DataSource / DataCursor	Une vue exige un cache local de données, une surface de binding, un curseur de navigation, un état de filtre ou un contexte maître/détail.	La déclaration relève du modèle métier : les champs de ce modèle ne sont pas les champs du cache de vue.
Action	Le système doit exprimer une intention : créer, valider, comptabiliser, actualiser, publier, calculer, notifier ou exécuter.	Le comportement relève seulement d'une propagation automatique gérée par un événement de dataset ou un binding de vue.

Créer	Quand ?	À éviter lorsque...
Événement	Un fait observable doit déclencher une réaction : clic, cycle de vie d'un dataset, changement métier, transition de processus ou arrivée de message.	Un appel d'action direct suffit, sans observateur, routage ni frontière de propagation.
Workflow / Processus	Plusieurs actions doivent être coordonnées avec état, ordre, intervention humaine, reprises, asynchronisme ou traçabilité.	Une seule action atomique suffit.
Service	Une capacité du modèle doit être exposée hors du runtime local ou de l'interface par une frontière de publication stable.	Vous souhaitez exposer un utilitaire interne ou un script devant rester privé.
Cadre de bureau réactif	Une page exige une zone d'accueil dynamique, un routage de composants enfants, une remontée d'événements parent/enfant, une prévisualisation, une enveloppe de workflow ou une coordination visuelle liée à un dataset.	Un formulaire simple, une grille, une vue partagée ou des onglets suffisent.
Assistant d'IA	Le modèle doit être consulté, expliqué ou manipulé par des actions conceptuelles contrôlées et un contexte traçable.	L'assistant contournerait la sémantique du modèle ou appellerait directement des détails d'implémentation privés.

38.4 Questions rapides avant toute création

Avant d'ajouter un artefact, demandez-vous :

— Cette idée métier doit-elle survivre aux changements d'interface ou de base de données ? Si oui, créez ou étendez un concept ou une relation.
— S'agit-il d'une représentation d'un concept existant pour le stockage, la mémoire, la recherche ou l'accès API ? Si oui, créez ou étendez une facette.
— Faut-il une liste ou un parcours réutilisable ? Si oui, créez une collection.
— S'agit-il d'une intention de l'utilisateur ou du système ? Si oui, créez une action.
— S'agit-il d'une réaction à un fait observable ? Si oui, créez un événement ou abonnez-vous à un événement existant.
— Faut-il coordonner plusieurs étapes ? Si oui, créez un modèle de workflow ou de processus.
— Faut-il composer une interface ? Si oui, choisissez un composant de vue, un cache DataSet ou un cadre réactif.
— S'agit-il d'une frontière externe ? Si oui, publiez intentionnellement un service ou un adaptateur d'API.

38.5 Décisions fréquentes chez les débutants

38.5.1 Concept ou champ ?

Créez un champ lorsque la valeur décrit un concept existant. Créez un concept lorsqu'elle possède sa propre identité, son comportement, son cycle de vie, ses contraintes ou ses vues. Par exemple, `Priority` peut être un champ de `Task`. `Project` doit généralement être un concept : il regroupe des tâches, possède son propre cycle de vie et peut avoir ses propres vues.

38.5.2 Relation ou clé étrangère ?

Une clé étrangère de base de données est une projection de stockage. Une relation logiCells porte une signification conceptuelle. Utilisez une relation lorsque le lien lui-même importe : appartenance, affectation, propriété, dépendance, délégation, approbation ou participation temporelle.

38.5.3 Action ou événement ?

Une action exprime ce que le système veut accomplir. Un événement exprime ce qui s'est produit ou ce qui doit être observé. Un bouton déclenche généralement une action. L'enregistrement d'un dataset, un changement de sélection ou une transition d'état métier peuvent émettre ou dispatcher un événement.

38.5.4 Action ou workflow ?

Utilisez une action pour une intention unique. Utilisez un workflow lorsque cette intention comporte des étapes, un état, un routage, des interactions humaines, des traitements asynchrones, des reprises ou des exigences d'audit.

38.5.5 Vue ou cadre réactif ?

Utilisez une vue classique pour une projection stable : formulaire, liste ou bloc de tableau de bord. Utilisez un cadre réactif de bureau lorsque la zone héberge des composants enfants dynamiques, route des événements locaux, réagit aux modifications du dataset ou sert d'enveloppe au workflow.

38.6 Mauvaises pratiques et corrections

Mauvaise pratique	Pourquoi cela pose problème	Correction
Partir d'un écran et inventer des champs au fil de sa construction.	Le concept apparaît par accident et ne peut plus être réutilisé de manière cohérente.	Nommer d'abord le concept, puis déclarer ses facettes, collections et vues.
Placer le sens métier uniquement dans un script.	Le runtime ne peut ni inspecter, ni publier, ni gouverner l'intention.	Déclarer l'action, la relation ou la contrainte ; laisser les scripts implémenter le comportement.
Exposer chaque action comme un service.	Des utilitaires internes deviennent des contrats publics.	Publier seulement des capacités stables, assorties d'entrées, de sorties et d'exigences de sécurité explicites.
Traiter un champ de dataset comme un champ du modèle.	Cache de vue et modèle de domaine se confondent.	Conserver les champs métier dans les facettes et ceux du DataSet dans le cache de vue.
Créer un workflow pour une opération atomique unique.	Le modèle devient plus complexe que le comportement.	Utiliser une action, puis ne la transformer en processus que si un état de coordination devient nécessaire.

Mauvaise pratique	Pourquoi cela pose problème	Correction
Faire transiter tous les événements d'interface par le serveur.	L'interface ralentit et les composants deviennent trop couplés.	Utiliser le dispatch local `PageControler` pour la coordination visuelle ; ne franchir la frontière que pour les changements de modèle ou de service.
Copier les anciens noms de packages dans les exemples publics.	La documentation révèle l'historique du projet et déroute les débutants.	Utiliser des espaces de noms neutres et décrire le modèle réutilisable.

38.7 Pour aller plus loin

Pour le parcours de base, poursuivez avec les chapitres de construction d'applications de ce guide. Pour le cache de vues, les bindings DataSet, le routage local des événements visuels et les interfaces réactives avancées, reportez-vous à l'ouvrage complémentaire *Guide des vues pour client riche logiCells*.

Chapitre 39

Les événements comme changements conceptuels observables

Les événements constituent la face observable du changement conceptuel. Ils indiquent qu'un fait s'est produit dans le runtime : sélection d'un objet, affichage d'un cadre, démarrage ou achèvement d'un processus, réception d'un message ou changement d'état métier.

Dans logiCells, les événements ne sont pas de simples fonctions de rappel. Ils sont déclarés dans les métadonnées. Ils peuvent posséder des paramètres, associer ceux-ci à des actions, restreindre leur source et aiguiller la réaction vers un objet, une classe ou un contrôleur particulier.

39.1 Événements et actions

Le modèle élémentaire est une déclaration associant une classe d'événement à un chemin d'action. Les paramètres de l'événement sont transmis à l'appel de l'action.

```
events:
  - lbs.ServiceVisit.Completed:
      sources: lbs#service_visit_process/*
      actions:
        - action:
            path: ./Action[@name='OnServiceVisitCompleted']
            params:
              - Source:
                  eventParam: Source
              - Sender:
                  eventParam: Sender
```

Listing 39.1 – Liaison d'un événement à une action

Cette déclaration rend visible le comportement réactif. L'application ne dissimule pas la règle dans un framework de programmation : le modèle indique qu'un événement déterminé déclenche une action déterminée.

39.2 Événements locaux et distants

Certains événements sont locaux à une vue ou à un objet du runtime. D'autres sont destinés à franchir les frontières de processus, de composants ou de services. Un modèle peut distinguer les événements locaux, distants ou relevant des deux catégories.

```
events:
  - App.Conversation.Open:
      isLocal: true
      isRemote: true
      params:
        - Sender:
```

```
            dataType: Object
            paramType: ptIn
```

Listing 39.2 – Déclaration d'un événement local et distant

Ce modèle est essentiel aux applications asynchrones : un même événement conceptuel peut déclencher une réaction dans l'interface locale et participer à la messagerie distribuée.

39.3 Les événements comme frontières de processus

Un processus doit émettre des événements aux frontières significatives de son cycle de vie métier : démarrage, attente, validation, achèvement, échec et annulation. Ces événements constituent le contrat observé par les contrôleurs, vues, services et autres processus.

```
actions:
- action:
    name: Start
    class: ObjectItemScriptableObjectElementAction
    type: ObjectItem
    published: true
    scripts:
    - clientScript: true
      serverScript: false
      language: functionalP
      classes: Desktop
      cdata: |
        Event := Self.ObjectsContainer.BuildMessage('lbs.Process.Started');
        Self.ObjectsContainer.SendEvent(Event, Self.EventProxy);

- action:
    name: Finalize
    class: ObjectItemScriptableObjectElementAction
    type: ObjectItem
    published: true
    scripts:
    - clientScript: true
      serverScript: false
      language: functionalP
      classes: Desktop
      cdata: |
        Event := Self.ObjectsContainer.BuildMessage('lbs.Process.Completed');
        Self.ObjectsContainer.SendEvent(Event, Self.EventProxy);
```

Listing 39.3 – Événements marquant les étapes du processus

L'essentiel n'est pas le nom précis de l'événement, mais l'exposition d'étapes de cycle de vie porteuses de sens métier.

39.4 Les contrôleurs comme observateurs

Un contrôleur doit observer les événements de processus et adapter l'expérience applicative. Il ne doit pas prendre en charge le processus métier lui-même.

```
class:
  classId: lbs#application_controller
  name: ApplicationController
  type: Entity

  events:
    - lbs.Process.Started:
        sources: lbs#service_visit_process/*
        actions:
          - action:
              path: ./Action[@name='OnProcessStarted']

    - lbs.Process.Completed:
```

```
14          sources: lbs#service_visit_process/*
15          actions:
16            - action:
17                path: ./Action[@name='OnProcessCompleted']
```

Listing 39.4 – Contrôleur observant les événements de processus

Cette séparation simplifie le raisonnement sur l'architecture : le processus porte la progression métier, et le contrôleur assure la réaction applicative.

39.5 Événements visuels

Les vues et cadres peuvent également déclarer des événements. Un bouton peut émettre un événement d'action visuelle, un cadre un événement d'affichage, et un cadre parent recevoir des événements de navigation.

Les événements visuels doivent conserver cette fonction. Ils peuvent demander le démarrage d'un processus ou une navigation au contrôleur, mais ne doivent pas contenir eux-mêmes tout le déroulement métier.

Chapitre 40

Flux de travail et processus

Un flux de travail logiCells ne se réduit ni à une succession d'écrans ni à un fragment de code procédural. C'est un objet conceptuel exécutable. Il peut porter un état, référencer des objets métier, exposer des actions, réagir aux événements et coordonner plusieurs participants dans la durée.

Cette distinction est fondamentale : la complexité d'une application apparaît souvent aux frontières entre les objets. Un objet métier peut prendre en charge ses données et son comportement locaux ; une opération impliquant plusieurs objets nécessite en revanche un lieu de coordination explicite. Dans logiCells, ce rôle revient au modèle de processus.

40.1 Le processus comme objet conceptuel

On introduit un modèle de processus dès qu'une opération dépasse la responsabilité locale d'un objet. Plusieurs indices le justifient :

— plusieurs objets métier participent à une même opération ;
— l'opération possède un cycle de vie observable : démarrée, en attente, validée, terminée ou en échec ;
— elle doit pouvoir reprendre après un événement externe ;
— l'interface utilisateur doit refléter l'avancement métier ;
— la coordination doit pouvoir être journalisée, auditée ou testée indépendamment.

Le modèle de processus ne remplace pas les objets participants : il les référence et les coordonne. Chaque objet métier demeure responsable de sa propre cohérence. Le processus prend en charge l'ordre des opérations, leur état et les modalités d'interaction.

40.2 Coordination synchrone et asynchrone

Un processus peut coordonner les objets de deux manières complémentaires.

Dans un processus synchrone, une action invoque directement une séquence d'autres actions. Le script constitue une orchestration explicite. Cette approche convient lorsque l'ensemble de l'opération doit s'exécuter immédiatement et de manière déterministe.

Dans un processus asynchrone, les actions émettent des événements ou des messages. D'autres objets les écoutent et réagissent lorsqu'ils sont prêts. Cette approche convient lorsque le processus franchit les frontières de l'interface utilisateur, de services distants, de files de messages, d'équipements, d'utilisateurs ou d'opérations de longue durée.

Dans les deux cas, un modèle de processus reste nécessaire. Sans lui, l'orchestration synchrone se retrouve dissimulée dans des scripts disparates et l'orchestration asynchrone se disperse entre les abonnés aux événements. Avec lui, la coordination demeure visible dans les métadonnées.

40.3 Le rôle de `lgc#process`

La plateforme définit la notion de processus comme une racine conceptuelle réutilisable. Un processus peut exposer une action `Execute` pour l'exécution directe et une action `Infer` pour la progression d'état ou l'évolution fondée sur des règles. Les processus propres à une application peuvent impliquer ou spécialiser cette racine lorsqu'ils ont besoin d'un contrat de coordination réutilisable.

Le modèle suivant est recommandé :

```
class:
  classId: lbs#construction_site_process
  name: ConstructionSiteProcess
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: lgc#process
        facets:
          - memory:
              type: hypergraph
              fields:
                - ConstructionSite: lbs#construction_site
                - CurrentState: String
```

Listing 40.1 – Modèle de spécialisation d'un processus

Le préfixe `lgc#` identifie les concepts réutilisables de la plateforme. Le préfixe `lbs#` peut servir aux concepts des exemples ou du domaine applicatif. Les préfixes propres à l'application doivent rester clairement distincts de ceux de la plateforme.

40.4 Le modèle de processus comme frontière de complexité

La règle d'architecture est simple :

> Dès qu'un processus dépasse une action locale, il doit être représenté par un modèle de processus explicite.

Cette règle prévient trois formes courantes de complexité accidentelle.

Premièrement, elle évite d'enfermer la logique interobjets dans un seul objet métier. Un chantier, un utilisateur ou un message ne doit pas orchestrer à lui seul, de manière implicite, toute l'application.

Deuxièmement, elle évite d'implémenter le flux métier dans les vues. Une vue peut émettre un événement ou appeler une action de processus, mais le processus doit rester responsable de la progression métier.

Troisièmement, elle évite de répartir les transitions d'état entre des gestionnaires d'événements sans lien explicite. Ces gestionnaires doivent réagir aux événements ; c'est au modèle de processus de déterminer la signification de leurs réactions dans le déroulement global.

40.5 Exemple de processus synchrone

Le processus simplifié suivant coordonne une création d'objets de manière synchrone. Il crée un objet métier, puis un bon associé, persiste les objets et enregistre enfin l'état final.

```
class:
  classId: lbs#service_visit_process
  name: ServiceVisitProcess
  className: CustomClassItem
  currentGeneratorName: Demo.Process.ServiceVisitProcess
  type: Entity

```

```yaml
  object:
    className: CustomObjectItem
    objectName: ServiceVisitProcess

  concepts:
  - Entity:
      constraints:
      - implies: lgc#process
      facets:
      - memory:
          type: hypergraph
          fields:
          - WorkSite: lbs#work_site
          - Customer: lbs#customer
          - Voucher: lbs#service_voucher
          - CurrentState: String

  actions:
  - action:
      name: Execute
      class: ObjectItemScriptableObjectElementAction
      type: ObjectItem
      published: true
      params:
      - Result:
          dataType: Boolean
          paramType: ptReturn
      scripts:
      - clientScript: true
        serverScript: false
        language: functionalP
        classes: Desktop
        cdata: |
          Result := false;
          Voucher := Self.ObjectsContainer.CreateObjectItem('lbs#service_voucher', 1, '');
          Voucher.Customer := Self.Customer;
          Voucher.WorkSite := Self.WorkSite;
          Voucher.PostData();

          Self.Voucher := Voucher;
          Self.CurrentState := 'Completed';
          Self.PostData();
          Result := true;
```

Listing 40.2 – Modèle de processus synchrone en YAML

Dans cet exemple, l'action du processus est volontairement synchrone. Elle est simple à tester et à tracer, puisque l'objet processus porte lui-même l'orchestration.

40.6 Exemple de processus asynchrone

Le même type de processus peut être modélisé de façon asynchrone. Le processus commence par émettre un événement ; d'autres objets ou contrôleurs peuvent réagir. Lorsque la réponse attendue arrive, le processus exécute sa finalisation.

```yaml
class:
  classId: lbs#service_visit_process
  name: ServiceVisitProcess
  type: Entity

  concepts:
  - Entity:
      constraints:
      - implies: lgc#process
      facets:
      - memory:
          type: hypergraph
          fields:
          - WorkSite: lbs#work_site
```

```
          - Voucher: lbs#service_voucher
          - CurrentState: String

  events:
  - lbs.ServiceVisit.Validated:
      sources: lbs#service_visit_process/*
      actions:
      - action:
          path: ./Action[@name='Finalize']

  actions:
  - action:
      name: Start
      class: ObjectItemScriptableObjectElementAction
      type: ObjectItem
      published: true
      scripts:
      - clientScript: true
        serverScript: false
        language: functionalP
        classes: Desktop
        cdata: |
          Event := Self.ObjectsContainer.BuildMessage('lbs.ServiceVisit.Started');
          Self.ObjectsContainer.SendEvent(Event, Self.EventProxy);
          Self.CurrentState := 'Started';

  - action:
      name: Finalize
      class: ObjectItemScriptableObjectElementAction
      type: ObjectItem
      published: true
      scripts:
      - clientScript: true
        serverScript: false
        language: functionalP
        classes: Desktop
        cdata: |
          Self.CurrentState := 'Completed';
          Self.PostData();
          Event := Self.ObjectsContainer.BuildMessage('lbs.ServiceVisit.Completed');
          Self.ObjectsContainer.SendEvent(Event, Self.EventProxy);
```

Listing 40.3 – Modèle de processus asynchrone en YAML

Le processus ne suppose plus que toutes les étapes se déroulent dans une seule pile d'appels : il coordonne leur enchaînement au moyen d'événements.

40.7 Règles de conception

Les objets métier portent leurs comportements locaux. Les modèles de processus coordonnent les comportements interobjets. Les événements rendent la coordination observable ; les états rendent l'avancement intelligible ; les règles en garantissent la validité. Les vues doivent afficher et déclencher le processus, sans en devenir propriétaires.

Lorsque la coordination possède une signification métier, donnez-lui un modèle. Lorsqu'elle ne relève que d'une réaction visuelle, conservez-la dans la vue ou le contrôleur. Lorsqu'elle franchit des frontières locales ou distantes, utilisez les événements ou la messagerie.

Chapitre 41

La messagerie asynchrone comme modèle de plateforme

Dans logiCells, la coordination asynchrone est une capacité de la plateforme. La couche de messagerie fournit des concepts réutilisables pour représenter les messages, les conversations, les participants, les états des messages et la distribution des événements. Les modèles applicatifs peuvent ainsi coordonner des interactions distribuées ou différées sans incorporer la logique de transport aux objets métier ni aux vues.

41.1 Concepts de messagerie fournis par la plateforme

Ce modèle commun s'appuie notamment sur :
— des concepts de message tels que `lgc#custom_message` et `lgc#text_message` ;
— des concepts de message liés aux processus, notamment `lgc#adhoc_workflow_message` ;
— des concepts de conversation tels que `lgc#conversation` ;
— des concepts de participant permettant d'associer des acteurs aux conversations ;
— la création des messages, le suivi de leurs états et leur horodatage ;
— la construction et la distribution des événements ;
— la remise des événements en local comme à distance ;
— l'intégration de la messagerie au niveau des packages.
Ce chapitre présente ces mécanismes réutilisables comme des composantes du modèle d'exécution logiCells.

41.2 Le message en tant qu'objet conceptuel

Un message est un objet du modèle. Il peut posséder une identité propre, une identité de classe, une date de création, une date de réception, un nom d'expéditeur, une référence à une conversation et un état.

```
class:
  classId: lgc#custom_message
  name: CustomMessage
  className: CustomClassItem
  namespace: Messaging
  currentGeneratorName: Messaging.CustomMessage
  type: Entity

  object:
    className: CustomObjectItem
    objectName: CustomMessage

  concepts:
    - Entity:
```

```
        constraints:
          - implies: lgc#entity
        facets:
          - CustomMessage:
              type: database
              map: lgcCustomMessage
              fields:
                - IdCustomMessage:
                    type: String
                    rolePath: Id
                    size: 32
                - ClassIdCustomMessage:
                    type: String
                    rolePath: ClassId
                    defaultValue: lgc#custom_message
                - IdConversation:
                    type: String
                    rolePath: Conversation.Id
                    nullable: true
                - CreationDate:
                    type: DateTime
                - ReceptionDate:
                    type: DateTime
                - SenderName:
                    type: String
```

Listing 41.1 – Modèle réutilisable de message personnalisé

Il ne s'agit pas d'un paquet de transport. C'est un objet conceptuel qui peut être persisté, interrogé, acheminé et rattaché à une conversation.

41.3 États des messages

Un message peut définir plusieurs états. Leur nomenclature exacte peut évoluer, mais le principe demeure : un message ne se réduit pas à son contenu ; c'est aussi un objet conceptuel doté d'un état.

```
states:
  entityStateIndexFieldName: StateIndex
  items:
    - Created:
        stateIndex: 0
        description: Message has been created
    - Empty:
        stateIndex: 1
        description: Message has no useful content yet
    - Sent:
        stateIndex: 2
        description: Message has been sent
    - Received:
        stateIndex: 3
        description: Message has been received
```

Listing 41.2 – Déclaration des états d'un message

Les états rendent les comportements asynchrones observables. Ils facilitent également la modélisation des nouvelles tentatives, des accusés de réception, des audits et de la traçabilité.

41.4 Distribution d'un événement à partir d'un message

Le caractère asynchrone apparaît notamment lorsqu'une action sur un message construit un événement puis le distribue.

```
actions:
- action:
    name: NewMessage
```

```
    class: ObjectItemScriptableObjectElementAction
    type: ObjectItem
    published: false
    params:
    - Sender:
        dataType: Object
        paramType: ptIn
    scripts:
    - clientScript: true
      serverScript: false
      language: functionalP
      classes: Desktop
      cdata: |
        Event := Self.ObjectsContainer.BuildMessage('lbs.Message.Receive');
        Event.Sender := Sender;
        Self.ObjectsContainer.SendEvent(Event, Self.EventProxy);
```

Listing 41.3 – Distribution d'un événement déclenchée par un message

L'objet message continue d'appartenir au modèle conceptuel. L'événement traduit, quant à lui, un changement asynchrone que d'autres concepts peuvent observer.

41.5 La conversation comme contexte de coordination

La conversation fournit aux messages un contexte de coordination. Elle permet de regrouper les échanges, de leur associer des participants et d'organiser des interactions asynchrones dans la durée.

```
actions:
- action:
    name: StartConversation
    class: ObjectItemScriptableObjectElementAction
    type: ObjectItem
    published: true
    params:
    - Sender:
        dataType: Object
        paramType: ptIn
    scripts:
    - clientScript: true
      serverScript: false
      language: functionalP
      classes: Desktop
      cdata: |
        ConversationCI := Self.ObjectsContainer.ClassItemByClassId('lgc#conversation');
        Conversation := Self.ObjectsContainer.CreateObjectItem(ConversationCI, 1, '');
        Conversation.Name := Self.Name;
        Conversation.PostData();

        ParticipantsCI := Self.ObjectsContainer.ClassItemByClassId('lgc#participants');
        Participant := Self.ObjectsContainer.CreateObjectItem(ParticipantsCI, 1, '');
        Participant.Conversation := Conversation;
        Participant.Participant := Sender;
        Participant.PostData();
```

Listing 41.4 – Ouverture d'une conversation depuis un objet de type contact

L'objet métier prend l'initiative d'ouvrir la conversation, tandis que les concepts de conversation et de participant restent des composants réutilisables de la plateforme.

41.6 Messagerie locale et distante

Un modèle peut déclarer qu'un événement est local, distant ou les deux. Un même événement conceptuel peut ainsi être pris en charge par le runtime local et par une couche de messagerie distante.

```
events:
  - lbs.Conversation.Open:
      isLocal: true
      isRemote: true
      params:
        - Sender:
            dataType: Object
            paramType: ptIn
```

Listing 41.5 – Événement de conversation local et distant

La capacité à acheminer des événements localement ou à distance appartient à la plateforme logiCells. Un espace de noms applicatif, comme `lbs#`, peut en bénéficier sans redéfinir l'infrastructure de messagerie.

41.7 Organisation en packages

Une application complète peut charger plusieurs packages. Ceux de la plateforme fournissent le noyau, les services, les contrôleurs, les composants visuels et la messagerie. Les packages applicatifs apportent les modèles métier, les vues, les ressources et, si nécessaire, des extensions propres au produit.

```
packages:
  - package:
      path: '[BasePath]/packages/Common.package'
      caption: Kernel
      enabled: true
      groupId: lgc#application_group
  - package:
      path: '[BasePath]/packages/CommonServices.package'
      caption: Kernel Services
      enabled: true
      groupId: lgc#application_group
  - package:
      path: '[BasePath]/packages/messagingPlatform.package'
      caption: Messaging Platform
      enabled: true
      groupId: lgc#application_group
  - package:
      path: '[BasePath]/packages/BusinessApplication.package'
      caption: Business Application
      enabled: true
      groupId: lbs#application_group
```

Listing 41.6 – Séparation des packages de plateforme et d'application

L'essentiel est que la messagerie reste une infrastructure réutilisable. Les packages applicatifs l'utilisent au moyen de modèles déclarés de messages, de conversations et d'événements.

Chapitre 42

Rôles, autorisations et visibilité contextuelle

La sécurité d'une application conceptuelle ne se résume pas à masquer des boutons. Elle régit les concepts, vues, actions et services qui peuvent être consultés ou exécutés dans un contexte donné.

42.1 Les rôles comme positions conceptuelles

Un rôle décrit ce qu'un acteur est autorisé à faire par rapport au modèle. Certains rôles appartiennent à la plateforme ; d'autres à un espace de noms applicatif tel que `lbs#`. Un rôle peut déterminer la visibilité d'une vue, la publication d'une action, une transition de processus ou l'accès à un service.

42.2 Frontières d'autorisation

Les actions publiées demandent une attention particulière. Si une action est accessible depuis un service web, un agent ou une interface utilisateur, son autorisation doit être explicite. Sa signature peut être publique, mais son exécution doit rester soumise au contrôle du contexte.

42.3 Visibilité contextuelle

La visibilité dépend de l'utilisateur, de l'objet, de l'état du processus, du canal de publication et parfois du périphérique ou du mode d'exécution. Une action de validation peut n'être visible que pendant l'attente d'une approbation ; une action de service peut être réservée à un rôle d'intégration.

42.4 Règles de conception

Ne considérez jamais la visibilité dans l'interface comme l'unique mesure de sécurité. Protégez les actions publiées. Traitez l'accès par service web comme un canal de publication à part entière. Placez les contrôles de rôle au plus près de l'action ou du processus protégé.

42.5 Synthèse

Les rôles et les autorisations rendent les applications conceptuelles gouvernables. Ils doivent être appliqués de façon cohérente aux vues, services, processus et agents.

Chapitre 43

Règles de conception des modèles applicatifs

Une application logiCells de qualité repose sur des règles de modélisation stables. Les principes suivants récapitulent les pratiques employées tout au long de ce guide.

43.1 Séparer les responsabilités

Utilisez les objets métier pour les données et les comportements locaux. Recourez à des objets de relation lorsque la relation possède sa propre identité, son état ou ses attributs. Utilisez les objets de processus pour coordonner plusieurs objets métier, les contrôleurs pour l'orchestration applicative et les vues comme projections, non comme responsables du déroulement des processus.

43.2 Modéliser avant de coder

Déclarez de préférence les concepts, rôles, facettes, actions, événements et vues avant d'écrire le code d'implémentation. Les scripts et classes enregistrées doivent réaliser une intention préalablement déclarée, et non introduire une sémantique implicite dans le modèle.

43.3 Publier de manière délibérée

Une action publiée constitue un contrat de capacité. Elle peut être consommée par une interface utilisateur, un service web, un agent ou un processus. Ne publiez pas les actions utilitaires qui doivent rester internes.

43.4 Présenter les exemples par paires

Employez YAML pour les exemples de MODEL. Réservez XML aux cas où il faut expliquer une source visuelle canonique `*.view.xml`.

43.5 Synthèse

La rigueur de modélisation rend le système intelligible : les concepts portent le sens, les processus assurent la coordination, les événements rendent les changements observables et la publication expose des capacités stables.

Chapitre 44

Objets du runtime, conteneurs et contextes d'exécution

Un modèle logiCells prend tout son sens lorsqu'il est placé dans un contexte d'exécution. Un fichier `.model` déclare des classes, des concepts, des facettes, des actions, des collections, des événements et des vues. Le runtime transforme ces déclarations en objets que l'on peut créer, interroger, relier, exécuter, persister et observer.

La notion centrale est celle de *conteneur conceptuel*. Un conteneur ne se limite pas à un registre d'objets techniques : il définit une frontière d'exécution au sein de laquelle les identifiants conceptuels, les définitions de classes, les instances, les collections, le routage des événements, les scripts et les projections de persistance forment un ensemble cohérent.

44.1 De la déclaration à l'objet du runtime

Une déclaration de classe introduit un type conceptuel, généralement repéré par un `classId` stable. À l'exécution, cette déclaration est représentée par un objet de classe. Les instances créées à partir de cette classe deviennent des éléments d'objet (*object items*).

```yaml
class:
  classId: lgc#work_item
  name: WorkItem
  type: Entity
  object:
    className: CustomObjectItem
    objectName: WorkItem
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
        facets:
          - main:
              type: hypergraph
              fields:
                - Title: String
                - Status: String
```

Listing 44.1 – Interprétation d'un modèle déclaratif par le runtime

Cette déclaration se lit à trois niveaux. Au niveau de la modélisation, elle définit une entité nommée `WorkItem`. Au niveau de la mémoire, elle introduit une structure conceptuelle dotée de rôles tels que `Title` et `Status`. Au niveau du runtime, elle permet de créer, retrouver, modifier, persister et afficher des objets représentant des tâches.

44.2 Éléments de classe et éléments d'objet

logiCells distingue le modèle d'un objet de l'objet lui-même.

Élément de classe (*class item*) Représentation, dans le runtime, d'une classe conceptuelle. Il porte l'identifiant de classe, les générateurs, les actions, les collections, les vues et les métadonnées.

Élément d'objet (*object item*) Instance d'une classe conceptuelle au sein du runtime. Il porte les valeurs, les relations, l'état et les comportements exécutables propres à l'objet.

Dans les exemples de scripts, cette distinction se manifeste notamment lorsqu'une action recherche d'abord une classe par son identifiant, puis crée ou retrouve une instance.

```
WorkItemClass := Self.ObjectsContainer.ClassItemByClassId('lgc#work_item');
WorkItem := Self.ObjectsContainer.CreateObjectItem(WorkItemClass, 1, '');
WorkItem.Title := 'Prepare field intervention';
WorkItem.Status := 'Created';
WorkItem.PostData();
```

Listing 44.2 – Recherche d'une classe et création d'un objet dans le runtime

L'essentiel n'est pas ici la syntaxe du script, mais le principe d'architecture : les métadonnées déclaratives fournissent au runtime les informations nécessaires pour construire et exécuter dynamiquement des objets conceptuels.

44.3 Le conteneur d'objets

Le conteneur d'objets constitue l'environnement d'exécution dans lequel les opérations sur les objets sont résolues. Il peut retrouver une classe, créer un objet, rechercher une instance existante, acheminer un événement, construire un message, ouvrir un fournisseur de collections ou accéder au conteneur de métadonnées.

Ses responsabilités habituelles comprennent :

— la résolution d'identifiants de classes conceptuelles tels que `lgc#entity`, `lgc#process` ou `lbs#work_item` ;
— la création d'instances à partir des métadonnées, plutôt qu'au moyen de constructeurs codés en dur ;
— l'accès aux collections d'objets et aux jointures ;
— l'exécution de scripts dans le contexte d'un objet ;
— la distribution des événements locaux ou asynchrones ;
— la synchronisation des valeurs des objets avec les projections de persistance.

Le conteneur est également le point de rencontre entre les espaces de noms de la plateforme et ceux des applications. Un concept de plateforme peut porter un identifiant `lgc#`, tandis qu'un concept propre à une application peut utiliser un espace de noms comme `lbs#`. Le runtime n'impose pas un espace de noms unique : il exige la stabilité des identifiants et la cohérence du graphe de modèles.

44.4 Self, contexte et liaison à l'exécution

Lors de l'exécution d'une action, `Self` désigne l'objet ou la classe qui reçoit l'action. Ce mécanisme rend les actions réutilisables : un même modèle déclaratif d'action peut s'appliquer à différentes instances, puisque sa cible est liée au moment de l'exécution.

```
action:
  name: MarkAsCompleted
  type: ObjectItem
  scripts:
```

```
    - language: functionalP
      cdata: |
        Self.Status := 'Completed';
        Self.PostData();
```

Listing 44.3 – Modèle d'action attachée à un objet

Le même objet peut aussi s'appuyer sur le conteneur pour coordonner d'autres objets.

```
Voucher := Self.ObjectsContainer.CreateObjectItem('lbs#service_voucher', 1, '');
Voucher.WorkItem := Self;
Voucher.PostData();
```

Listing 44.4 – Création d'un objet associé depuis une action

C'est un premier pas vers la modélisation des processus : dès lors qu'une action commence à coordonner plusieurs objets, cette coordination mérite généralement son propre modèle de processus.

44.5 Persistance des changements conceptuels

Un objet logiCells peut exister dans la mémoire conceptuelle, dans une projection persistante, ou dans les deux. Une facette de base de données décrit la projection des rôles conceptuels vers les enregistrements et les champs. Les appels du runtime tels que `PostData` rendent les changements durables selon les règles de projection déclarées dans le modèle.

```
facets:
  - main:
      type: hypergraph
      fields:
        - Title: String
        - Status: String
  - WorkItem:
      type: database
      map: lbsWorkItem
      fields:
        - IdWorkItem:
            rolePath: Id
        - ClassIdWorkItem:
            rolePath: ClassId
            defaultValue: lbs#work_item
        - Title:
            size: 255
        - Status:
            size: 32
```

Listing 44.5 – Facettes hypergraphe et base de données d'un même concept

La facette hypergraphique exprime la structure conceptuelle ; la facette de base de données exprime sa persistance. Leur coexistence dans le même modèle donne à l'objet du runtime une signification conceptuelle et une traduction opérationnelle.

44.6 Règle de conception

La règle pratique est simple :

> Le modèle décrit ce que signifie un objet ; le conteneur fournit à ce modèle un contexte d'exécution.

Lorsque cette règle est respectée, le code applicatif n'a pas besoin de figer chaque classe, chaque relation, chaque processus ou chaque vue. Le runtime peut les découvrir à partir du modèle conceptuel.

Chapitre 45

Collections, jointures et modèles interrogeables

Une application logiCells n'est pas seulement un ensemble d'objets isolés. Elle constitue un espace conceptuel navigable. Les collections et les jointures définissent comment retrouver, projeter, filtrer et combiner les objets.

Les collections font donc partie du modèle et non de la seule interface utilisateur. Une collection peut exposer toutes les instances d'une classe, un sous-ensemble filtré, le contenu d'une relation ou une jointure entre plusieurs sources conceptuelles.

45.1 Les collections comme contrats de navigation du runtime

Une déclaration de collection nomme un mode de parcours des objets.

```yaml
collections:
  - collection:
      name: Entity
      target: ObjectItem
      providerFactory: ObjectItemInstancesCollectionProviderFactory
```

Listing 45.1 – Déclaration simple d'un fournisseur de collection

Le consommateur de cette collection n'a pas besoin de connaître la manière dont les objets sont stockés. Il lui suffit de savoir que la classe conceptuelle expose une collection nommée `Entity`. Le runtime résout le fournisseur et renvoie un ensemble navigable d'objets.

45.2 Les relations comme structures interrogeables

Un modèle de relation prend souvent la forme d'une entité ordinaire dotée de champs pointant vers les concepts associés. La facette de base de données conserve les clés de relation, tandis que la facette hypergraphique rend visibles les rôles conceptuels.

```yaml
class:
  classId: lbs#work_item_phase
  name: WorkItemPhase
  type: Entity
  concepts:
    - Entity:
        constraints:
          - implies: lgc#entity
        facets:
          - main:
              type: hypergraph
              fields:
                - WorkItem: lbs#work_item
```

```
                - Phase: lbs#phase
          - WorkItemPhase:
              type: database
              map: lbsWorkItemPhase
              fields:
                - IdWorkItemPhase:
                    rolePath: Id
                - IdWorkItem:
                    rolePath: WorkItem.Id
                - ClassIdWorkItem:
                    rolePath: WorkItem.ClassId
                    defaultValue: lbs#work_item
                - IdPhase:
                    rolePath: Phase.Id
                - ClassIdPhase:
                    rolePath: Phase.ClassId
                    defaultValue: lbs#phase
```

Listing 45.2 – Objet de relation entre une tâche et une phase

Cette approche fait des relations des éléments de premier ordre. Une relation peut posséder sa propre identité, ses champs, contraintes, actions, son historique et ses vues.

45.3 Collections issues de jointures

Une collection issue d'une jointure projette des informations provenant de plusieurs sources dans un résultat navigable unique. Elle est utile lorsqu'une vue doit présenter un objet métier accompagné de données issues d'objets associés.

```
collections:
  - join:
      name: WorkItemPhases
      target: ObjectItem
      fields:
        - Id:
            fieldName: WorkItemPhase.IdWorkItem
        - ClassId:
            fieldName: WorkItemPhase.ClassIdWorkItem
        - IdPhase:
            fieldName: Phase.Id
            rolePath: Phase.Id
        - PhaseName:
            fieldName: Phase.Name
      collections:
        - collection:
            name: WorkItemPhase
            target: ClassItem
            sourcePath: /ClassItem[@classId='lbs#work_item_phase']/Collection[@name='Entity']
        - collection:
            name: Phase
            target: ClassItem
            sourcePath: /ClassItem[@classId='lbs#phase']/Collection[@name='Entity']
      constraints:
        - constraint:
            joinKind: inner
            collection1: WorkItemPhase
            collection2: Phase
            fieldName1: IdPhase
            fieldName2: IdPhase
      filters:
        objectItemFilter: |
          Id=#Id
```

Listing 45.3 – Déclaration d'une collection par jointure

La navigation reste déclarative : la vue consomme `WorkItemPhases`. Elle n'a donc pas à rechercher directement les relations, à résoudre les objets de phase, puis à composer les lignes d'affichage.

45.4 Chemins de rôles

Un `rolePath` relie un champ projeté à un rôle conceptuel. Il constitue le pont entre la persistance tabulaire et la mémoire conceptuelle.

```yaml
fields:
  - IdWorkItem:
      rolePath: Id
  - ClassIdWorkItem:
      rolePath: ClassId
  - IdPhase:
      rolePath: Phase.Id
  - FormalNamePhase:
      rolePath: Phase.FormalName
```

Listing 45.4 – Exemples de chemins de rôles

Un chemin de rôle doit être compris comme un chemin sémantique, et non seulement comme une instruction de correspondance entre champs. Il indique au runtime où la valeur prend place dans l'objet conceptuel.

45.5 Modèles interrogeables et complexité

Avec la croissance des applications, les jointures deviennent indispensables. Faute de modèles de jointure explicites, la logique de requête est souvent dupliquée dans les écrans, scripts et rapports. Des collections déclarées explicitement font au contraire de la requête une partie réutilisable du modèle conceptuel.

> Une collection est un chemin nommé dans la mémoire conceptuelle. Une jointure est une projection nommée sur plusieurs chemins.

La règle de conception consiste à placer les schémas de navigation stables dans les déclarations du modèle. Les scripts doivent orchestrer les situations exceptionnelles, et non reproduire des requêtes structurelles qui appartiennent au modèle.

Chapitre 46

Configuration de l'application et assemblage des packages

Une application logiCells est assemblée de façon déclarative. Sa configuration relie le runtime conceptuel, le stockage en graphe ou hypergraphe, les alias de bases de données, les packages, les extensions binaires, les services, les ressources, les journaux et les bureaux.

La frontière applicative devient ainsi visible : plutôt que de dissimuler l'assemblage du runtime dans le code, logiCells l'expose au moyen d'un modèle de configuration.

46.1 Configuration de lancement du runtime : YAML `.lgc`

Un fichier de lancement du runtime portant l'extension `.lgc` est un document YAML. Le mapping `kernel` décrit l'environnement de lancement et `wallets` fournit les paramètres du portefeuille d'identifiants. Ce fichier est distinct du descripteur applicatif désigné par `kernel.config`.

```yaml
kernel:
  config: 'C:\logiCells\config\BaseConsole.Memory.application.yaml'
  modules: 'C:\logiCells\modules'
  base: 'C:\logiCells\runtime'
  libraries: 'C:\logiCells\runtime\bin'
  abi: 'C:\logiCells\abi'
  dsl: 'functionalP'
  services: 'httpServ;liveness;readiness;dbgclideamon;openapiService;mcpService'
  engine: 'C:\logiCells\runtime\bin\clgc.exe'
  type: 'Application'
wallets:
  user: 'example-user'
  password: 'REPLACE_ME'
```

Listing 46.1 – Configuration de lancement du runtime (.lgc)

Les chemins indiqués ne sont que des exemples de déploiement. `kernel.config` sélectionne le descripteur `*.application.yaml` qui assemble packages, modèles et services. `kernel.abi` pointe vers les informations ABI fournies au moteur. `kernel.services` contient une sélection d'identifiants de service séparés par des points-virgules ; `kernel.dsl` sélectionne ici `functionalP`. La valeur `wallets.password` n'est qu'un substitut : les identifiants doivent être protégés selon l'environnement cible.

Le runtime résout cette configuration de lancement avant de charger le descripteur applicatif, les chemins de packages et les manifestes de modèle nommés `_.manifest.yaml`.

46.2 Identité applicative et bureau

La configuration commence par nommer l'application et déclarer ses environnements de bureau.

```yaml
configuration:
  application:
    name: BusinessApplication
    desktopName: DefaultBusinessDesktop
    desktops:
      - DefaultBusinessDesktop:
          caption: Default Business Desktop
```

Listing 46.2 – Déclaration de l'identité applicative

Le bureau ne se réduit pas à une fenêtre : il constitue le contexte visuel racine à partir duquel sont chargés les vues, les contrôleurs, les modèles de page et les composants sous-jacents (*object-behind*).

46.3 Chargement du graphe et des métadonnées

La section dédiée au graphe ou à l'hypergraphe déclare les modalités d'hébergement de la mémoire conceptuelle.

```yaml
hypergraph:
  metaDataHypergraphKind: MemoryHypergraph
  loadApplicationMetaData: false
  inference:
    mode: auto
    execution: multithread
```

Listing 46.3 – Configuration du graphe et de l'inférence

L'implémentation précise du graphe peut varier entre développement, tests et production. Son rôle architectural reste celui du substrat de mémoire dans lequel concepts, relations, instances et métadonnées peuvent être interprétés. La distinction actuelle entre l'autorité des métadonnées NativeStore et l'autorité de l'hypergraphe sémantique est précisée plus loin dans ce chapitre.

46.4 Éléments de package, chemins de schémas, chemins de vues et bureaux

Un élément `item` de package est une unité logique activable, et non un simple nom de répertoire. Il regroupe une contribution cohérente au modèle ou au package, que l'application peut activer ou ignorer.

```yaml
# Business package
package:
  files:
    schemaDirectories:
      schemaPath:
        value: '[Modules]\\task-management\\meta\\models\\'

# Companion VIEW package
package:
  dependencies:
    - task-management
  files:
    schemaDirectories:
      schemaPath:
        value: '[Modules]\\task-management\\meta\\views\\models\\'
      schemaPath:
```

```
        value: '[Modules]\\task-management\\meta\\views\\contracts\\'
  items:
    - TaskManagement.Task:
        caption: Task
        enabled: true
```

Listing 46.4 – Racines du package métier et de son package VIEW associé

`schemaPath` indique au chargeur où trouver les manifestes et les structures porteuses de modèles. Le package VIEW associé possède deux racines aux responsabilités distinctes : `views/models` pour les redéfinitions ou surcharges de MODEL et `views/contracts` pour les contrats d'interaction et leurs conteneurs MODEL-behind colocalisés. Les éléments du package VIEW sont relatifs à ces racines : leur emplacement physique ne justifie pas de les préfixer par `contracts.` ou `models.`.

Un package VIEW doit rester limité aux interactions appartenant à son package métier. L'absence d'une ancienne déclaration `viewPaths` ne justifie pas de revendiquer l'ensemble de l'arborescence globale `meta/views`.

La section `desktops` déclare les points d'entrée ou assemblages de bureaux de l'application. Elle fait le lien entre l'application chargée et une expérience de bureau navigable. Ses attributs dépendent du runtime de bureau, mais sa fonction architecturale reste stable : elle choisit la configuration d'interface racine plutôt que de définir les concepts métier.

46.5 Alias de bases de données

Un alias associe un nom logique de base de données à une configuration concrète de pilote.

```
dataBaseAlias:
  - dBAlias:
      name: Default
      items:
        - lgc#default_group:
            type: SQLite
            driverPackage: logiCells.Data.FireDac
            value: DriverID=SQLite;database=[BasePath]/database/application.db3
```

Listing 46.5 – Déclaration d'un alias de base de données

Les détails propres à la base de données ne doivent pas se propager dans le modèle conceptuel. Celui-ci déclare les facettes et les mappings ; la configuration applicative choisit le binding de base de données du runtime.

46.6 Packages binaires et packages de modèles

Une application logiCells résulte de l'assemblage de plusieurs catégories de packages.

```
container:
  binaryPackages:
    - logiCells.Community
    - logiCells.SQLDataStructure
    - logiCells.Agent
    - logiCells.MVCCore
  packages:
    - package:
        path: '[Modules]/logicells-kernel-open/packages/Common.package'
        caption: Kernel
        enabled: true
        groupId: lgc#default_group
    - package:
        path: '[Modules]/business-application/packages/BusinessApplication.package'
        caption: Business Application
        enabled: true
        groupId: lgc#default_group
```

Listing 46.6 – Assemblage des packages du runtime

Ce modèle distingue les packages réutilisables de la plateforme des packages applicatifs. Un package de plateforme peut fournir des entités, de la messagerie, des vues, des contrôleurs ou des services. Un package applicatif apporte les modèles de domaine, processus métier, vues et ressources propres à l'application.

46.7 Espaces de noms de la plateforme et de l'application

Les exemples publics peuvent employer des espaces de noms propres à la plateforme ou à l'application. Le premier utilise généralement `lgc#`, tandis que l'application peut employer, par exemple, `lbs#`. Cette distinction permet d'identifier immédiatement la responsabilité des concepts.

```
lgc#entity
lgc#process
lgc#custom_message
lbs#work_item
lbs#work_item_process
lbs#phase
```

Listing 46.7 – Identifiants de la plateforme et de l'application

La règle n'est pas d'utiliser `lgc#` partout, mais de séparer clairement les concepts de la plateforme et ceux de l'application.

46.8 Services et fichiers

Une configuration peut également déclarer des définitions de services, répertoires temporaires, journaux, caches, dépôts de synchronisation, instantanés, dépôts de workflows et portefeuilles d'identifiants.

```
files:
  tempDirectory: '[BasePath]/temp/'
  logDirectory: '[BasePath]/log/'
  cachePath: '[BasePath]/cache/'
  synchronizationRepository: '[BasePath]/synchronization/'
  snapshotRepository: '[BasePath]/snapshots/'
  workflowBaseRepository: '[BasePath]/process/'
```

Listing 46.8 – Emplacements des fichiers du runtime

Ces emplacements sont des détails opérationnels, mais font néanmoins partie de la frontière applicative déclarative.

46.9 Capacités réutilisables de la plateforme

C'est aussi dans l'assemblage applicatif que deviennent visibles les capacités réutilisables de la plateforme. Une application logiCells n'est pas seulement constituée de concepts métier : les packages de plateforme peuvent apporter de la messagerie, des caches, des composants visuels, des services, workflows, mécanismes de synchronisation, agents, ressources, styles ou fournisseurs de runtime.

```
Packages de plateforme
  -> concepts du noyau
  -> couches MVC et de cache
  -> composants fonctionnels UXfc
  -> messagerie et conversations
```

```
  -> services et adaptateurs d'API
  -> synchronisation et instantanes
  -> agents et execution des workflows

Packages applicatifs
  -> concepts de domaine
  -> processus de l'application
  -> vues et modeles de page
  -> fournisseurs et integrations
  -> ressources et styles
```

Listing 46.9 – Capacités réutilisables dans un assemblage de packages

Cette discipline de packaging sépare les responsabilités de la plateforme de celles d'une application particulière. Une même infrastructure de messagerie, des composants visuels fonctionnels, un cache de données, une architecture de fournisseurs ou un modèle d'adaptateur de services peuvent servir à plusieurs applications sans exposer les workflows ou détails visuels propres à un client.

46.10 Ressources locales et distantes du runtime

La configuration peut déclarer des dépôts opérationnels utilisés dans les déploiements *local-first* ou distribués.

```
cache
  donnees pretes pour les vues et resultats des fournisseurs

synchronisation
  modifications en attente et etat de replication distante

snapshots
  etats enregistres coherents ou packages d'echange

process
  etat d'execution des workflows et processus

resources
  icones, styles, dispositions et ressources statiques
```

Listing 46.10 – Catégories de ressources du runtime

Ces répertoires ne sont pas de simples détails de système de fichiers : ils correspondent aux responsabilités du runtime en matière de cache local, synchronisation, packaging, récupération, exécution des processus et ressources d'expérience utilisateur.

46.11 Du package au modèle chargé

Un fichier présent sous `meta/models` n'est pas chargé du seul fait de son existence sur disque. L'intégration d'un nouveau module à l'application emprunte une chaîne explicite de chargement :

```
configuration applicative
  -> descripteur de package
  -> schemaDirectories / schemaPath
  -> _.manifest.yaml
  -> modele declare
  -> ClassItem du runtime
  -> initialisation semantique, le cas echeant
```

Listing 46.11 – Chaîne de chargement déclaratif d'un module

Lors de la création d'un module, vérifiez cette chaîne de l'extérieur vers l'intérieur. Cette précaution est particulièrement importante si deux copies du dépôt possèdent des modèles de

noms similaires ou si un ancien package publie encore une déclaration obsolète.

Pour un module autonome, l'organisation de projet public suivante est recommandée :

```
my-module/
  packages/
    MyModule.package.yaml
    MyModule.Tests.package.yaml
  meta/
    models/
      MyModule/
        _.manifest.yaml
        MyModel.model.yaml
      MyModuleTests/
        _.manifest.yaml
        MyModelTests.model.yaml
  src/
  tests/
```

Listing 46.12 – Organisation recommandée d'un module

Le jeu exact de fichiers source dépend du module, mais le package et le manifeste restent les points d'entrée déclaratifs.

46.12 Compiler et démarrer une application en ligne de commande

Les distributions qui fournissent l'hôte `sclgc` permettent de charger et démarrer une application depuis une configuration locale `.lgc`, avec `-c` :

```
sclgc -c /path/to/application.application.ACC.lgc
```

Listing 46.13 – Démarrer une application configurée avec sclgc

L'argument `-c` désigne le fichier local à charger. Celui-ci permet ensuite de résoudre l'application, les chemins de modules, l'assemblage des packages et les autres paramètres du runtime. Exécutez le programme dans un environnement où les packages binaires du runtime et leurs dépendances sont accessibles ; dans un arbre de développement, il s'agit souvent du répertoire `bin` utilisé pour le build.

Cette commande diffère de la CLI Tooling destinée à la conception. Cette dernière expose la compilation des applications par le chargeur applicatif natif, tandis que `sclgc -c ...` démarre le runtime avec la configuration locale sélectionnée. Ces deux surfaces de commande ne sont pas interchangeables.

Les valeurs de configuration locales peuvent aussi dépendre des variables d'environnement du processus. Lorsque la CLI Tooling accepte un paramètre explicite `-param`, l'ordre de priorité validé est le suivant :

```
--param explicite
  > valeur du fichier .lgc
  > environnement du processus
  > valeur non resolue
```

Listing 46.14 – Ordre de priorité des valeurs de configuration

L'autorité actuelle des paramètres applicatifs accepte aussi bien la forme longue précédée de deux tirets que la forme courte précédée d'un seul tiret. Par exemple, `-config path` et `-c path` désignent le même paramètre applicatif. Les paramètres déclarés sans valeur sont des indicateurs ; les autres consomment l'argument suivant sur la ligne de commande.

Nom long	Court	Valeur	Fonction
`config`	`c`	oui	Chemin du fichier de configuration.
`roles`	`r`	oui	Rôles utilisateur ou niveaux d'accès.

`debug`	`dbg`	non	Activer les diagnostics détaillés.
`listeners`	`ls`	oui	Nombre d'écouteurs du serveur.
`send-to`	`sto`	oui	Destination du transfert de données.
`services`	`sg`	oui	Services initialisés au démarrage.
`base`	`b`	oui	Répertoire de base.
`modules`	`m`	oui	Modules à charger.
`libraries`	`l`	oui	Bibliothèques supplémentaires.
`language`	`lg`	oui	Langue de l'application.
`test`	`t`	non	Exécuter en mode test.
`test-out`	`to`	oui	Répertoire des résultats de tests.
`test-application`	`ta`	non	Activer les tests applicatifs.
`test-hypergraph`	`th`	non	Activer l'option test-hypergraph du contrat de paramètres actuel.
`test-hypergraph-params`	`thp`	non	Activer l'option test-hypergraph-params du contrat actuel.
`script`	`f`	oui	Script à exécuter.
`db-name`	`d`	oui	Nom de la base de données.
`db-user`	`dbu`	oui	Utilisateur de la base de données.
`db-password`	`dpb`	oui	Mot de passe de la base de données.
`server`	`s`	non	Exécuter une instance serveur.
`log-level`	`ll`	oui	Niveau de journalisation.
`configuration`	`a`	non	Activer les paramètres de configuration alternatifs.
`ast-debug`	`astsd`	non	Activer le débogage des scripts/AST.
`memory-profiler`	`mp`	non	Activer le profilage mémoire.
`cache-view`	`cv`	non	Afficher la vue du cache.
`dsl`	`dsl`	oui	Fichier DSL à charger.
`minimal`	`mi`	oui	Paramètre du mode minimal.
`style`	`st`	oui	Configuration des styles.

Un build de diagnostic transitoire peut proposer `-followed-index`/`-fi` avec une valeur. Cette option dépend de la condition de compilation `TRANSIENTMLDEBUG` et ne doit donc pas être présentée comme une option portable du runtime.

Lorsque le contrôle strict des paramètres est actif, l'analyseur rejette les options inconnues et celles qui exigent une valeur mais figurent en dernière position sans argument. Les noms longs utilisent `-name`, les courts `-name`. Cette surface appartient au runtime applicatif et ne doit pas être déduite d'une autre façade CLI.

46.12.1 Reconnaître un démarrage réussi

Pour modifier un module déclaratif, il ne suffit pas de constater que le processus démarre. Suivez les étapes de chargement dans les journaux :

```
Loading application file
Loading package ...
Compiling package item ... _.manifest.yaml...
Compiling model ...
Start initializing/updating semantic Hypergraphs
End initializing/updating semantic Hypergraphs
```

Listing 46.15 – Points de contrôle d'un smoke test du runtime

Si le modèle n'apparaît pas dans cette séquence, recherchez d'abord une erreur d'assemblage du package ou du manifeste, avant d'analyser les actions, les prompts ou le comportement métier.

46.13 Règle de conception

La configuration répond à la question :

> Quels runtime, packages, graphes, bases de données, services, ressources et bureaux sont assemblés pour rendre cette application conceptuelle exécutable ?

Rendre cette réponse déclarative facilite la reproduction, l'inspection, le déploiement et l'évolution des environnements.

46.14 Configurations initiales en mémoire et persistantes

Le projet de démarrage fournit des fichiers de configuration applicative proches de ceux d'une production, et non de simples coquilles minimales. Il utilise YAML pour la configuration ; ses sources visuelles restent des fichiers canoniques `*.view.xml`.

```
yaml/configurations/BaseConsole.Memory.application.yaml
yaml/configurations/BaseConsole.Persistent.application.yaml
```

Listing 46.16 – Configurations de l'application de démarrage

La configuration en mémoire est recommandée pour la première exécution : elle permet de vérifier le chargement des métadonnées, l'assemblage des packages et la visibilité des modèles avant de traiter la persistance. La configuration persistante est plus proche d'un environnement réel, mais exige d'adapter explicitement les alias de bases, les chemins locaux, les identifiants, les ports et l'activation des services.

L'ordre de diagnostic suivant est conseillé :

```
1. Demarrer avec la configuration Memory.
2. Verifier que le package est active.
3. Verifier que le manifeste expose la classe attendue.
4. Verifier que ObjectsContainer peut resoudre la classe.
5. Ouvrir la vue et verifier les bindings DataSource.
6. Passer a la configuration Persistent.
7. Verifier les alias et mappings de bases de donnees.
8. Activer les services seulement lorsque le modele fonctionne.
```

Listing 46.17 – Ordre de diagnostic de la configuration

Il évite une erreur courante : chercher la cause d'un problème de service ou d'écriture en base alors que le package ou le manifeste n'est pas correctement chargé.

46.15 Métadonnées compilées et initialisation du graphe sémantique

La configuration applicative choisit les packages sources, leurs emplacements, les bases de métadonnées, les groupes du runtime et les services associés. Dans le pipeline à froid *native-strict*, `TMetadataNativeStore` constitue le stockage faisant autorité pour les métadonnées assemblées et produit les images `.lcmp` ; l'ancien `MetaDataHypergraph` des métadonnées ne subsiste pas comme solution de repli cachée. Le `HypergraphBundle` sémantique reste une cible indépendante et légitime pour H-Logic et les relations logiques. Distinguez le digest source et le plan de compilation de la génération des artefacts compilés, de celle du graphe sémantique et de la génération de la persistance SQL. Le chemin warm utilisant uniquement les images compilées doit valider les sources *et* la sécurité, plutôt que de forcer silencieusement l'association aux façades.

Le runtime propose des options facultatives de performance pour la compilation et le démarrage ; elles ne modifient pas le modèle applicatif. Ne définissez pas `LGC_INCREMENTAL_SIGNATURE_CACHE_REV` sans avoir établi la révision exacte et figée de l'analyseur YAML, du DOM et de la grammaire ; si la variable est absente, le calcul normalisé des signatures reprend. Le parcours développeur est détaillé au chapitre 47.

Chapitre 47

Compilation incrémentale, analyse d'impact et métadonnées natives

47.1 Identifier l'autorité avant de choisir une commande

La commande `cell` est un client du Tooling natif logiCells : elle ne constitue pas un second moteur de métadonnées. L'opération doit être choisie selon l'autorité qui lui revient :

MODEL Inspecter ou modifier les sources déclaratives XML/YAML physiques au moyen du parseur natif et du DOM source. Une opération MODEL ne charge pas implicitement une application ni un graphe sémantique.

APPLICATION Inspecter ou modifier les métadonnées assemblées pour une configuration `.lgc` résolue, notamment les façades natives. Une modification APPLICATION peut publier une nouvelle génération persistante de métadonnées même en l'absence de fichier source physique.

Plan de compilation Consulter le plan des changements de sources, sans modifier l'application.

Compile Appliquer un delta incrémental uniquement lorsque son application est sûre. Un refus de compilation incrémentale ne déclenche jamais implicitement une construction complète.

Build Lancer explicitement une construction complète ; après sa réussite, celle-ci fournit une nouvelle référence de comparaison.

47.2 Procédure de travail pour le développeur

```
cell doctor
cell workspace inspect <workspace>
cell application inspect <application.lgc>
cell application compile plan <application.lgc>
cell application compile <application.lgc>
# if the plan requires a full rebuild, decide explicitly:
cell application build <application.lgc>
```

Le plan distingue les `ChangedUnits`, les classes directement modifiées (`DirtyClasses`), la fermeture des `AffectedClasses`, les motifs de propagation typés, l'éligibilité, `fullRebuildRequired` et `fallbackReason`. La valeur `compiled=True` ne signifie pas nécessairement qu'une modification a été appliquée : une source inchangée peut être compilée avec succès tandis que `applied=False`. Les commandes correspondantes à l'échelle du package sont `cell package compile` et `cell package build`. Lorsque `package`

`compile --verbose` réussit, le plan affiché provient de la réponse d'exécution et pas nécessairement d'un appel préalable et indépendant à la planification.

Dans l'implémentation caractérisée, le planificateur piloté par les sources prend surtout en charge les modifications d'unités de classe. L'ajout ou la suppression d'autres catégories peut rester non pris en charge. Un plan affichant `changed=0`, `dirty=0` et `affected=0` établit seulement que l'algorithme de signatures testé considère la référence actuelle comme inchangée.

47.3 Distinguer source, aspect et dépendance

Le modèle R10/R13 actuel compare un condensat normalisé fondé sur le DOM et six signatures d'aspect : structure, sémantique, comportement, état, persistance et vue. Les arêtes de dépendance typées déterminent ensuite les consommateurs affectés ; des modèles voisins sans dépendance ne sont pas automatiquement concernés. Un changement d'espacement ou de commentaire peut modifier les octets du fichier tout en laissant les métadonnées normalisées identiques. Une Action dépendante peut devoir être recompilée sans modification de son propre fichier lorsque les types ou les projections de rôles qu'elle utilise évoluent.

```
source file -> native XML/YAML parser -> normalized digest
                                      -> six aspect signatures
                                      -> typed dependency closure
                                      -> eligible incremental plan
```

Le résultat `1/1/7` de la fixture de référence `lgc#folder` n'est pas une règle générale. Il faut distinguer l'optimisation `ClassChangeIsFieldAddOnly` de la propagation ordinaire des dépendances. Si la fermeture typée ou les signatures disponibles sont incertaines, le compilateur doit conserver un repli prudent.

47.4 NativeStore ne remplace pas le graphe sémantique

Le modèle de métadonnées assemblées `TXXXInfo` repose sur `TMetadataNativeStore` et sur les façades compilées `.lcmp` dans les parcours *native-strict*. L'ancien support de métadonnées `MetaDataHypergraph`/`FDataContent` ne constitue pas un repli autorisé lors d'un démarrage à froid *native-strict*. Le graphe sémantique `HypergraphBundle` demeure une autorité distincte pour le raisonnement, H-Logic, l'opérationnalisation et le contexte. Une compilation réussie des métadonnées ne prouve donc pas, à elle seule, que les artefacts sémantiques `.sem`, les types du graphe ou le schéma SQL sont à jour.

47.5 Démarrage à froid, démarrage à chaud et cache YAML physique

Le paramètre facultatif `LGC_COLD_MODEL_SINGLE_DOM=1` évite de construire un second DOM pour les chargements de modèles YAML éligibles lors d'un démarrage à froid. Une autre optimisation, facultative, utilise le fichier annexe `.units.physical-signatures` pour réemployer les condensats normalisés et signatures d'aspect de sources inchangées. Elle exige une révision du parseur renseignée et maintenue par le déploiement (`LGC_INCREMENTAL_SIGNATURE_CACHE_REV`), ainsi qu'une référence complète et validée dont l'identité source, le MD5 physique et la compatibilité du parseur correspondent. Au moindre doute, le calcul historique par parseur, DOM et signatures reprend. Ce cache n'est pas une autorité métier et ne remplace pas l'analyse des dépendances `typed-v1`.

La mesure de référence du 11 octobre 2026, réalisée en Win32 Debug, rapporte 111 correspondances de cache sur 111 unités inchangées et une réduction de 91,1% du temps d'analyse

des sources lors du second essai à froid. Ces chiffres rapportés ne démontrent ni l'équivalence sémantique de tous les fichiers générés ni une invalidation concurrente de niveau production. Le cache d'images `.lcmp` et le fichier annexe de signatures physiques sont indépendants : le premier sert à réutiliser les métadonnées compilées ; le second évite de recalculer inutilement leurs signatures source.

47.6 Analyser une défaillance sans affaiblir les invariants

Si une exécution à froid atteint `functionalP>`, le démarrage minimal a fonctionné. Il faut néanmoins examiner les diagnostics `#supertype` non résolus, la compatibilité de forme des types, les artefacts produits et les modifications éventuelles des fixtures avant de conclure à la cohérence sémantique complète. Si le chargement compilé à chaud refuse le binding natif, il faut rechercher l'origine des signatures et les conditions d'initialisation de la sécurité. Une trace contenant `applicationMD5=` puis `native-store-reparse` indique un repli à diagnostiquer, et non une autorisation de forcer le binding ou de fabriquer une signature. L'anomalie observée en v37 sur le parcours à chaud reste ouverte.

47.7 Analyse d'impact dans VS Code : respecter les responsabilités

L'intégration VS Code peut présenter les unités modifiées et affectées avec leurs emplacements sources. Consultez la version installée de l'extension pour connaître les fonctions d'analyse d'impact proposées ; l'éditeur ne doit pas déduire seul la sûreté du compilateur.

47.8 Liste de vérification du développeur

1. Consigner les révisions des sources et du compilateur, la commande exacte et la réponse du Tooling.
2. Vérifier `ChangedAspects` et les motifs typés, sans se fier à la seule taille de la fermeture.
3. Tester une modification réelle du modèle, puis sa relance immédiate en `0/0/0` ; tester aussi la restauration, et pas seulement l'ajout.
4. Vérifier la stabilité des identifiants et les tombstones pour les collections et leurs éléments structurels.
5. Comparer les sorties `.lcmp`, `.ast` et `.sem` pertinentes dans des essais contrôlés avec cache activé et désactivé.
6. Distinguer démarrage à froid vérifié, cohérence sémantique, parité des artefacts, publication concurrente et binding sécurisé à chaud.

Les algorithmes natifs et la matrice de preuves datées sont détaillés dans le livre complémentaire *Runtime RAD Studio*, dans les chapitres consacrés à NativeStore et à la compilation incrémentale des métadonnées. Le présent chapitre en expose la lecture côté développeur : il ne définit pas un compilateur alternatif.

Chapitre 48

Services déclaratifs du runtime et modèles d'exécution

Une déclaration de service n'implémente pas, à elle seule, un ordonnanceur, une file d'attente, un agent ou un journal. Elle sélectionne, configure et instancie une implémentation de runtime enregistrée par le moteur ou un package binaire. Comprendre ce lien est indispensable à la lecture des fichiers de service de production.

48.1 De la déclaration au comportement du runtime

```
metadonnees de service
-> analyseur et structure declarative
-> definition du service
-> recherche moduleClass dans le registre du runtime
-> instance du runtime configuree
-> cycle de vie : demarrage / arret
-> evenements, minuteurs, files ou comportement HTTP
```

Listing 48.1 – Chaîne de résolution d'un service déclaratif

La déclaration est donc un contrat de configuration exécutable : elle désigne une famille de runtime, fournit ses paramètres et détermine les instances actives.

48.2 Trois niveaux de service

Élément	Rôle	Contenu habituel
`serviceGroup`	Regroupement de publication et de déploiement au niveau du package	Chemin de définition et une ou plusieurs instances
`serviceGroupDefinition`	Contrat réutilisable de type de service et de configuration	`uuid`, `name`, `type`, `moduleClass`, pages, analyseurs, paramètres
`serviceInstance`	Instance concrète en cours d'exécution	Adresse, port, base de données, configuration, paramètres d'activation et de démarrage

48.3 Attributs de définition courants

Attribut	Signification

`uuid`	Identité stable de la définition. Conservez-la lorsqu'un déploiement ou des références en dépendent.
`type`	Grande famille de service, par exemple HTTP ou une autre catégorie reconnue par le moteur.
`moduleClass`	Classe ou module du runtime chargé de réaliser le comportement déclaré. C'est le lien essentiel entre métadonnées et code.
`debug`	Active les diagnostics du service lorsque cela est pris en charge ; ce n'est pas nécessairement la seule commande de journalisation.
`start`	Indique si et comment le service participe au démarrage, selon la génération du runtime. Vérifiez les valeurs admises sur le moteur cible.
`baseURL`	Adresse de base employée pour composer ou publier les routes.
`path`	Chemin ou emplacement de routage actuel.
`path_old`	Chemin historique de compatibilité rencontré dans d'anciens projets. Ne l'introduisez pas dans du nouveau code sauf exigence de migration.
`defaultPages`	Pages ou gestionnaires par défaut lorsqu'une requête ne sélectionne aucune route plus précise.

48.4 Sections complémentaires

`views` Déclare les ressources de vues ou de pages exposées par la frontière du service.

`parameters` Fournit des valeurs de configuration nommées ou typées au module du runtime.

`viewParsers` Enregistre les analyseurs ou transformations utilisés pour interpréter les vues ou contenus publiés.

`binaryPackages` Charge les implémentations du runtime requises par la définition de service.

La documentation de ces sections doit être établie à partir de la classe du runtime qui les consomme. Une section de nom similaire dans une autre famille de services peut offrir des garanties différentes.

48.5 Modèles architecturaux intégrés

Les configurations de production peuvent contenir des noms comme agents, files d'attente, journaux, opérations, règles ou minuteurs. Ces noms désignent des familles de services du runtime ; ils ne prouvent pas que toutes les propriétés classiques du patron d'architecture correspondant soient implémentées.

Famille	**Intention habituelle**	**Points à vérifier**
Agent	Exécution autonome ou déléguée	Déclencheur, droits, état, reprises et frontière des actions
Queue	Traitement asynchrone mis en tampon	Garantie de livraison, ordre, persistance, reprises et messages en échec définitif
Journal	Trace durable ou enregistrement d'événements	Conservation, ordre, rejeu et frontière transactionnelle
Operation	Opération de service exécutable et nommée	Contrat d'entrée, cible, résultat et sémantique des échecs

Rules	Évaluation ou dispatch déclaratif	Contexte d'évaluation, ordre, résolution des conflits et effets de bord
SimpleTimer	Déclenchement périodique par intervalle	Intervalle, démarrage, chevauchement et comportement en cas d'échec
calendarTimer	Déclenchement selon un calendrier	Fuseau horaire, exécutions manquées, récurrence et concurrence

Ce guide évite délibérément d'attribuer des garanties non confirmées par l'implémentation et les tests de la famille concernée.

48.6 Exemple : définition d'un service HTTP

```yaml
serviceGroupDefinitions:
  - UserManagement:
      uuid: EA588812932A434B87A5F47A795C27E4
      type: HTTP
      moduleClass: DataModuleHTTPServer
      debug: true
      virtualPages:
        - method: GET
          path: /users
          class: PathRequestWebPage
          rewrite: /ClassItem[@name='UserManagement.User']/Collection[@name='Entity']
```

Listing 48.2 – Définition d'un service HTTP en YAML

48.7 Pages HTTP virtuelles et fichiers d'extension de page

La déclaration HTTP enregistre des routes virtuelles sous `virtualPages`. Dans l'exemple, un service HTTP actif recevant `GET /users` peut sélectionner la règle `method: GET`, `path: /users`. La classe déclarée correspond au gestionnaire compatible ABI `PathRequestWebPage` ; `rewrite` identifie la collection conceptuelle cible. Une méthode ou un chemin différent ne sélectionne pas cette route.

Les fichiers d'extension de page HTTP portent le suffixe `.lgp`, par exemple `Users.lgp`. Un fichier de page et une route `virtualPages` sont deux ressources distinctes : l'existence du fichier ne suffit pas à publier une route. L'exemple ne précise aucun champ permettant de charger physiquement le fichier de page ; le rapprochement entre la ressource de page et le gestionnaire enregistré doit respecter le contrat du runtime cible.

```
service HTTP actif
-> GET /users
-> entree correspondante dans virtualPages
-> PathRequestWebPage (gestionnaire ABI)
-> rewrite vers UserManagement.User / collection Entity
-> reponse du service HTTP
```

Listing 48.3 – Sélection déclarative d'une requête pour une page HTTP virtuelle

L'analyse des requêtes, l'autorisation, l'appel du gestionnaire et la résolution physique de `.lgp` constituent des frontières d'implémentation. Les seules métadonnées déclaratives ne permettent pas d'en déduire le callback précis du runtime.

48.8 Ajouter un nouveau modèle de service du runtime

L'ajout d'une nouvelle famille déclarative exige généralement davantage qu'une nouvelle règle de métadonnées :

1. implémenter ou réutiliser une classe de runtime ;
2. définir le contrat de métadonnées qu'elle accepte ;
3. enregistrer l'implémentation dans le moteur ou un package binaire ;
4. exposer les propriétés de configuration et les points d'entrée du cycle de vie ;
5. déclarer une instance dans les métadonnées de service ;
6. tester le démarrage, l'arrêt, les erreurs, les événements et la persistance ;
7. documenter les garanties dont les appelants peuvent dépendre.

Cette frontière d'extension est l'un des atouts de logiCells : de nouvelles capacités du runtime deviennent composables de façon déclarative dès que leur contrat de métadonnées et leur enregistrement sont stabilisés.

48.9 Stratégie de référence documentaire

Un inventaire exhaustif de tous les anciens marqueurs de service deviendrait rapidement obsolète. La documentation recommandée distingue :

```
chapitre conceptuel
  explique le passage de la declaration au runtime

reference de la famille de runtime
  documente les attributs admis et les garanties de cycle de vie

exemples valides
  montrent les configurations testees avec une version identifiee du moteur

sources et tests
  font autorite pour le comportement propre a l'implementation
```

Listing 48.4 – Niveaux de documentation des services

Chapitre 49

Déploiement, conditionnement et distribution du runtime

Une fois l'application modélisée, il faut préparer sa distribution et son déploiement. Dans logiCells, cette étape repose sur le même principe que la modélisation : rendre la structure explicite.

Un package de déploiement ne se limite pas à un programme compilé. Il comprend une arborescence applicative réunissant les binaires du runtime, les packages de modèles, les vues, les services, les ressources, les bases de données, les fichiers de configuration, les journaux, les caches, les dossiers de synchronisation et les descripteurs de mise à jour.

49.1 Plateformes d'exécution prises en charge

logiCells est actuellement disponible sous **Windows**, **Linux** et **macOS**. Le runtime peut également être déployé dans un **conteneur**.

Le modèle conceptuel exécutable n'est pas lié à l'un de ces systèmes d'exploitation. Le conditionnement propre à chaque plateforme, les bibliothèques natives et les paramètres de déploiement relèvent de l'exploitation ; ils entourent le même modèle conceptuel et les mêmes contrats du runtime.

Un conteneur prépare le runtime pour le déploiement : il ne définit pas un autre modèle applicatif.

49.2 Disponibilité des plateformes et certification des sous-systèmes

Le runtime est distribué pour les trois familles de systèmes d'exploitation précitées, sur poste de travail ou serveur. Cette disponibilité ne signifie pas que toutes les voies d'intégration natives présentent le même niveau de certification sur chaque hôte. La documentation d'ingénierie du runtime décrit des références de compilation native pour Windows, macOS (y compris Apple Silicon) et Linux. En revanche, le chargeur de bibliothèques dynamiques natives utilisé par l'implémentation ABI actuellement fournie comprend un chemin de code propre à Windows.

Pour préparer une adoption, consignez le statut de chaque sous-système plutôt que de cocher une seule case globale :

Périmètre	État actuel	Portée de l'affirmation
Distribution du runtime central	Windows / Linux / macOS	Le runtime conceptuel est disponible sur ces plateformes.

Déploiement en conteneur	Disponible	Un runtime peut être conditionné et exploité dans un conteneur ; les caractéristiques de l'image et de l'hôte restent propres au déploiement.
Références de développement et de compilation natives	Documentées pour Windows / Linux / macOS	Les documents d'ingénierie recensent les cibles natives et les environnements de développement de ces hôtes.
Chargement dynamique de l'ABI native	Certification propre à la plateforme nécessaire	Ne déduisez pas la certification du chargeur natif sous Linux ou macOS de la seule disponibilité du runtime ; vérifiez le package de binding pour l'hôte retenu.
Bindings ABI Python / C#	Disponibles	La disponibilité des bindings et la couverture de leur cycle de vie sont décrites au Chapitre 64, "Modèles déclaratifs et ABI du runtime" ; leur conditionnement doit néanmoins être vérifié pour chaque hôte.

Cette distinction est délibérée. La disponibilité d'une plateforme indique où le runtime peut être utilisé ; la certification atteste qu'un chemin précis a été testé dans un sous-système donné, avec un backend et une configuration de distribution déterminés.

49.3 Arborescence de déploiement

Une arborescence de déploiement sépare généralement les ressources stables de la plateforme des ressources propres à l'application.

```
application/
  packages/
  models/
  views/
  services/
  resources/
  database/
  database/clean/
  graph/
  cache/
  log/
  temp/
  synchronization/
  snapshots/
  process/
  wallets/
```

Listing 49.1 – Exemple d'arborescence de déploiement

Cette organisation est importante, car les applications logiCells sont pilotées par les métadonnées. Les modèles et les vues sont des ressources exploitées par le runtime, et non de simples artefacts du développement.

49.4 Assembler les ressources de plateforme et d'application

Une configuration de génération statique ou de conditionnement copie généralement plusieurs catégories de fichiers :

— les packages du noyau et de la plateforme ;
— les fichiers de modèles ;
— les fichiers de vues ;
— les définitions de services ;
— les ressources et les styles ;
— les modèles de bases de données ou les bases initiales vierges ;
— les grammaires et les ressources de script ;
— les fichiers de configuration de l'application.

```
finalTreeInfos:
  - finalTreeInfo:
      targetPlugIn: common
      name: Models_kernel
      sourceFile: '#models_kernel#*.*'
      targetDirectory: '#target_data#models/'
  - finalTreeInfo:
      targetPlugIn: application
      name: Models_application
      sourceFile: '#models_application#*.*'
      targetDirectory: '#target_data#models/'
  - finalTreeInfo:
      targetPlugIn: application
      name: Views_application
      sourceFile: '#views_application#*.*'
      targetDirectory: '#target_data#views/'
```

Listing 49.2 – Exemple abstrait de génération statique

Les noms concrets des répertoires varient selon les projets, mais le principe reste le même : les ressources de plateforme et d'application sont assemblées dans une arborescence utilisable par le runtime.

49.5 Configuration selon l'environnement

Les configurations de développement et de diffusion peuvent différer. Le développement peut utiliser des graphes en mémoire, des chemins locaux, des journaux de débogage et des bases de test. La diffusion peut, quant à elle, employer des graphes persistants, des ressources conditionnées, des modèles de bases de données vierges et des dossiers de runtime à chemins fixes.

```
# Development
(graphKind: MemoryRichGraph, graphParams: [Workspace]/Graph)

# Release
(graphKind: PersistentRichGraph, graphParams: [BasePath]/Graph)
```

Listing 49.3 – Association du graphe selon l'environnement

Le modèle conceptuel devrait rester stable entre ces environnements ; la configuration le rattache au déploiement concret du runtime.

49.6 Mises à jour et évolution du runtime

Un descripteur de mise à jour peut déclarer les modules, leurs versions et leurs emplacements de téléchargement. Il permet de rendre explicite l'évolution du déploiement.

Les mécanismes de mise à jour sont opérationnels, mais ils participent aussi à la gouvernance des modèles. Dès lors que modèles, vues et services sont des ressources du runtime, leur mise à jour fait partie de l'évolution applicative.

49.7 Déploiement en conteneur

Une application logiCells peut être conditionnée et exécutée dans un conteneur. Dans une architecture orientée services, les dossiers applicatifs peuvent être montés en volumes et le runtime peut exposer les ports nécessaires.

```
services:
  business-application:
    image: business-application
    ports:
      - 9696:9696
    volumes:
      - ./deploy/database:/var/application/database
      - ./deploy/temp:/var/application/temp
```

Listing 49.4 – Exemple de déploiement en conteneur

L'essentiel est que la conteneurisation ne remplace pas le runtime conceptuel : elle le conditionne pour son exploitation.

49.8 Règle de conception

Un déploiement doit rendre visibles les frontières suivantes :
— ce qui relève de la plateforme ;
— ce qui relève de l'application ;
— ce qui constitue la configuration ;
— ce qui est généré ;
— ce qui correspond à l'état persistant du runtime ;
— ce qui peut être mis à jour indépendamment.

Le déploiement est le prolongement opérationnel de la modélisation déclarative.

Chapitre 50

Plan de référence pour un nouveau module ERP

Ce chapitre propose le point de départ recommandé pour une nouvelle capacité ERP. Il réunit les conventions de sources, les frontières MODEL, le contrat VIEW, les tests et l'assemblage des packages en un ensemble reproductible.

50.1 1. Partir d'une capacité métier délimitée

Nommez le module d'après une capacité métier stable, et non d'après un écran. Exemples : `erp-procurement`, `erp-scheduling`, `erp-training` ou `erp-expense`. Rédigez les invariants et le cycle de vie avant de créer les vues.

50.2 2. Arborescence recommandée

```
Modules/erp-example/
  meta/
    models/
      _.manifest.yaml
      BusinessObject.model.yaml
      BusinessRelation.model.yaml
      BusinessProcess.model.yaml
    views/
      BusinessObject/
        _.manifest.yaml
        BusinessObjectList.view.xml
        BusinessObjectList.models.yaml
        BusinessObjectEdit.view.xml
        BusinessObjectEdit.models.yaml
  packages/
    erp-example.package.yaml
    erp-example.view.package.yaml
Tests/
  twin-test-example/
    ...
```

Listing 50.1 – Squelette d'un module ERP

50.3 3. Définir d'abord l'identité et les invariants du modèle

Utilisez des valeurs `classId` stables. Rendez explicites l'identité et les relations obligatoires. Placez la validation au voisinage du modèle métier, plutôt que de vous reposer sur un formulaire pour rendre un champ obligatoire.

```
class:
  name: BusinessObject
  classId: erp#example_business_object
  currentGeneratorName: Uri
  type: Entity
  concepts:
    - Entity:
        facets:
          - main:
              type: hypergraph
              fields:
                - Number:
                    type: String
                    nullable: false
                - Status:
                    type: String
                    nullable: false
                    defaultValue: DRAFT
        actions:
          - action:
              name: Validate
              class: ObjectItemScriptableObjectElementAction
              type: ObjectItem
              published: false
              scripts:
                - script:
                    clientScript: false
                    serverScript: true
                    language: functionalP
                    cdata: |
                      Result := '';
                      if Self.Number = '' then
                        Result := 'Number is required';
```

Listing 50.2 – Squelette d'une entité ERP

50.4 4. Exprimer les intentions métier dans des actions réutilisables

Une action doit pouvoir être appelée depuis une vue, un service, un test ou un processus sans reproduire ses invariants. Privilégiez les opérations idempotentes lorsque des reprises sont possibles. Séparez le nettoyage destructif de la suppression métier lorsque des objets temporaires ou des instantanés existent.

50.5 5. Employer les collections comme modèles de lecture réutilisables

Ne construisez pas chaque requête d'écran directement dans VIEW. Déclarez les collections et projections une seule fois, nommez-les selon leur signification métier et réutilisez-les dans les tableaux de bord, les services et les tests. Si un rapport historique est migré, retrouvez sa projection sémantique plutôt que de recréer un objet racine de navigation au seul motif que l'ancienne application en possédait un.

50.6 6. Confier la coordination aux processus

Utilisez un processus lorsqu'une transition métier coordonne plusieurs objets ou étapes. Les scripts VIEW doivent démarrer ou observer le processus, et non implémenter eux-mêmes son enchaînement transactionnel.

50.7 7. Ajouter le package VIEW après le contrat métier

Créez les fichiers `*.view.xml` pour la hiérarchie visuelle et `*.models.yaml` pour les contrôleurs. Les contrôleurs doivent rester simples et orientés client. Pour les interactions asynchrones, utilisez des événements visuels explicites et des continuations du cycle de vie.

50.8 8. Tester le contrat par couches

Un nouveau périmètre ERP devrait posséder au minimum :

1. des contrôles de parsing MODEL, de manifestes et d'identité des classes ;
2. des tests de contrat d'actions pour la validation et les transitions d'état ;
3. des tests des collections et des modèles de lecture ;
4. des tests de processus lorsqu'une orchestration entre objets existe ;
5. des tests de liaison VIEW : chaque `source/class/classId` XML est résolu une seule fois ;
6. des tests du protocole VIEW asynchrone pour l'attente, l'achèvement et le rafraîchissement ;
7. des contrôles d'assemblage des packages et de l'application ;
8. des tests sur un environnement logiCells pris en charge avant la diffusion.

50.9 9. Définition de fin de réalisation

Un module n'est pas achevé simplement parce que ses fichiers sont analysables. Avant de considérer une capacité ERP comme terminée, vérifiez :

— la stabilité de l'identité métier et des invariants ;
— la réutilisabilité des actions et des collections ;
— l'attribution explicite de l'orchestration entre objets à un processus ;
— la séparation VIEW et MODEL-behind ;
— la maîtrise des comportements asynchrones pilotés par événements ;
— l'exhaustivité des packages et des manifestes ;
— l'existence de tests prouvant le contrat métier ;
— la validation du runtime ou, à défaut, une mention explicite du fait qu'elle reste à réaliser.

Septième partie

Sémantique du runtime

Chapitre 51

Des déclarations à l'exécution dans le runtime

Une application logiCells commence par un ensemble de déclarations, mais celles-ci ne restent pas de simples descriptions passives. Un fichier `.model.yaml` est chargé dans le runtime conceptuel, transformé en métadonnées, rattaché à un graphe, puis rendu exécutable au moyen des objets du runtime. La transition essentielle n'est donc pas celle du « modèle vers le code », mais celle de la « déclaration vers le concept exécutable ».

Ce chapitre décrit ce passage et montre comment un modèle déclaratif devient une structure d'exécution composée de `ClassItem`, de `ObjectItem`, d'actions, de collections, d'événements, de vues et de services.

51.1 Le parcours d'exécution dans le runtime

Le chemin d'exécution peut se lire comme une chaîne de responsabilités :

```
.model.yaml
  -> graphe de metadonnees
  -> ClassItem
  -> ObjectItem
  -> facettes et champs
  -> collections et fournisseurs
  -> actions
  -> evenements
  -> vues et services
```

Listing 51.1 – De la déclaration à une structure exécutable dans le runtime

La déclaration définit ce qu'est un concept. Le runtime lui donne une forme exécutable.

Un `ClassItem` représente le concept sous la forme d'un objet de type classe dans le runtime. Il connaît son identifiant de classe ainsi que les facettes, collections, actions, vues et générateurs déclarés. Un `ObjectItem` représente une instance de ce concept. Il porte des valeurs, participe aux relations, expose des actions et peut être persisté au moyen des facettes déclarées.

Le conteneur du runtime constitue l'environnement dans lequel ces objets sont résolus et exécutés. Il propose notamment des opérations pour retrouver une classe à partir de son identifiant, créer une instance, retrouver un objet existant, ouvrir un fournisseur de collection, distribuer un événement ou persister des modifications.

51.2 ClassItem et ObjectItem

Une déclaration de classe devient un `ClassItem` dans le runtime. C'est notamment le cas de `lgc#custom_message`, de `lgc#conversation` et de `lbs#work_item`. Le `ClassItem` ne se limite

pas à un schéma : il porte les éléments exécutables suivants :

— l'identité du concept ;
— les facettes déclarées ;
— les collections disponibles ;
— les actions de classe et les actions d'instance ;
— les abonnements aux événements ;
— les déclarations de vues ;
— les générateurs et les valeurs par défaut.

Un `ObjectItem` est ensuite créé à partir de ce `ClassItem`. Il correspond à l'instance conceptuelle exécutable. Une tâche peut ainsi être créée, associée à un client, persistée, affichée dans une vue, utilisée par un processus puis mentionnée dans un événement.

51.3 Les conteneurs du runtime

Le conteneur constitue la frontière opérationnelle de l'application. Il donne accès aux métadonnées, aux instances d'objets, aux collections, à la distribution des actions, au routage des événements et aux services. Dans les scripts, ce rôle apparaît notamment dans les expressions suivantes :

```
1 WorkItemClass := Self.ObjectsContainer.ClassItemByClassId('lbs#work_item');
2 WorkItem := Self.ObjectsContainer.CreateObjectItem(WorkItemClass, 1, '');
3 WorkItem.Name := 'Initial assessment';
4 WorkItem.PostData();
```

Listing 51.2 – Opérations caractéristiques du conteneur du runtime

Cet exemple illustre une séquence élémentaire :

1. résoudre le concept à partir de son identifiant de classe ;
2. créer un `ObjectItem` exécutable ;
3. lui affecter des valeurs conceptuelles ;
4. persister l'objet par sa facette de persistance déclarée.

Le même schéma se retrouve dans la coordination des processus, la messagerie, les contrôleurs de vues et les adaptateurs de services. Les applications logiCells ne s'organisent donc pas autour de classes de code isolées, mais autour de concepts que le conteneur rend exécutables.

51.4 Les facettes à l'exécution

Les facettes décrivent les projections d'un même concept dans différents espaces techniques et conceptuels. La facette hypergraphique définit ses rôles et ses relations conceptuelles. La facette base de données indique comment une partie de cette structure est persistée. Une facette de vue ou de service peut projeter ce même concept vers une interface utilisateur ou une frontière d'intégration.

Le runtime ne traite pas ces facettes comme des schémas indépendants : il les utilise comme des projections coordonnées d'un même concept. C'est pourquoi un champ peut posséder un `rolePath` tel que `Customer.Id` ou `Conversation.ClassId` : la représentation persistée conserve une projection d'une relation conceptuelle.

51.5 Les actions comme points d'entrée du runtime

Les actions sont les points d'entrée les plus visibles dans l'exécution. Une déclaration peut exposer une action d'instance, une action de classe, une action scriptable ou une action implé-

mentée par un composant enregistré dans le runtime. L'essentiel est que l'action appartienne au concept.

Le script n'est qu'un détail d'implémentation. L'action elle-même représente la capacité métier déclarée.

51.6 Événements et exécution réactive

À l'exécution, un événement associe une source, une charge utile et une ou plusieurs actions. Les événements évitent d'introduire des dépendances implicites entre objets métier, objets de processus et vues. Un processus peut émettre un événement, un contrôleur peut s'y abonner, une vue peut réagir et un composant de messagerie peut le propager de manière asynchrone.

C'est une différence fondamentale entre l'orchestration procédurale et l'exécution d'un runtime conceptuel. Dans un système procédural, l'appelant doit connaître l'appelé. Dans un runtime conceptuel, un concept émet un événement porteur de sens, puis le runtime le route vers les actions déclarées.

Chapitre 52

Mémoire conceptuelle et propagation des changements

La mémoire conceptuelle est l'espace du runtime où les concepts, objets, relations, actions et événements s'articulent. Il ne s'agit pas d'une couche passive de stockage, mais de l'endroit où les changements peuvent être interprétés, propagés et observés.

52.1 La mémoire comme graphe conceptuel

Lorsqu'un modèle est chargé, ses déclarations deviennent des métadonnées du runtime. Les `ClassItem` définissent des types de concepts exécutables ; les `ObjectItem` instancient ces concepts ; les relations relient les objets. Les actions et événements donnent un comportement à cette structure.

Un changement peut provenir d'une action utilisateur, d'un appel de service web, d'un agent, d'un script, d'un message, d'une synchronisation ou de l'évaluation d'une règle. Le runtime doit préserver la signification du changement : quel concept et quel rôle ont changé, quel processus est concerné et quels observateurs peuvent réagir.

52.2 Du changement local à sa propagation

Un changement local suit généralement le déroulement suivant :

1. modifier un rôle ou un champ d'objet ;
2. valider l'objet dans son contexte courant ;
3. enregistrer la modification ;
4. rafraîchir les collections, vues ou processus dépendants ;
5. émettre, si nécessaire, des événements vers les participants concernés ;
6. publier éventuellement la modification par synchronisation distante ou messagerie.

```
1 WorkItem.Status := 'Validated';
2 WorkItem.PostData();
3 Event := Self.ObjectsContainer.BuildMessage('lbs.WorkItem.Validated');
4 Self.ObjectsContainer.SendEvent(Event, Self.EventProxy);
```

Listing 52.1 – Modification locale suivie de la persistance

52.3 Changements observables

Un changement est plus robuste lorsqu'il peut être observé. Une affectation cachée dans un script est difficile à tracer. Une modification enregistrée, puis signalée par un événement explicite, peut être suivie, rejouée, auditée ou transmise à un processus.

52.4 Règles de conception

Gardez les modifications métier au plus près du concept responsable. Émettez des événements lorsqu'une réaction extérieure est nécessaire. Employez des modèles de processus pour coordonner plusieurs objets. Évitez les effets de bord invisibles entre objets. Rendez explicites et traçables les changements d'état importants.

52.5 Synthèse

La mémoire conceptuelle fournit au runtime un espace commun aux objets, relations, processus, vues, services et agents. La propagation reste maîtrisable lorsque les changements prennent la forme de mises à jour conceptuelles accompagnées d'événements observables.

Chapitre 53

Contextes d'évaluation

Un contexte d'évaluation fixe les conditions dans lesquelles un élément du modèle est interprété. Un même concept peut être évalué différemment selon l'objet, l'utilisateur, le processus, la vue, l'appel de service ou l'agent courants.

53.1 Pourquoi le contexte importe

Un modèle déclaratif évite les connexions codées en dur, mais ne supprime pas le contexte : il le rend explicite. Une collection affichée dans une vue peut dépendre de l'objet sélectionné ; une action publiée peut n'être autorisée que pour un rôle précis ; un appel de service peut transporter l'identifiant d'un locataire ou un espace de noms applicatif ; un agent peut disposer d'une enveloppe d'exécution restreinte.

53.2 Contextes courants

Les contextes les plus fréquents sont les suivants :
— contexte d'objet : le `ObjectItem` courant ;
— contexte de classe : le `ClassItem` courant ;
— contexte de vue : le cadre, le curseur et le jeu de données courants ;
— contexte de processus : l'état du processus et son objet courant ;
— contexte de sécurité : l'utilisateur, ses rôles et ses autorisations ;
— contexte de publication : le service, le point d'entrée et le protocole ;
— contexte d'agent : la tâche, la portion de mémoire accessible et les actions autorisées.

53.3 Propagation du contexte

Une vue peut transmettre son objet courant à un curseur imbriqué. Un processus peut se désigner comme source d'un événement. Un service peut associer les paramètres d'une requête à ceux d'une action. Un agent peut recevoir un ensemble restreint d'objets et d'actions.

53.4 Règles de conception

Ne comptez pas sur un contexte implicite lorsqu'une frontière du modèle est franchie. Rendez le contexte explicite lors des appels de services, de l'émission d'événements, de l'ouverture de vues et du démarrage des processus. Évitez les hypothèses globales qui compliquent les tests.

53.5 Synthèse

Les contextes d'évaluation permettent à un même modèle conceptuel de servir des utilisateurs, des vues, des processus, des services et des agents sans le dupliquer pour chaque mode d'exécution.

Chapitre 54

Transactions, cohérence et identité

Les applications logiCells manipulent des objets conceptuels qui doivent néanmoins posséder une identité, des règles de cohérence et une persistance fiable. Ce chapitre présente les frontières d'identité et de transaction à prendre en compte dans les modèles exécutables.

54.1 Identité conceptuelle

Un objet n'est pas défini seulement par une clé de base de données. Il possède une classe conceptuelle, une identité d'objet, un nom formel et souvent un rôle contextuel dans un autre objet. Les facettes de base de données exposent généralement cette identité par des champs comme `Id`, `ClassId` et `FormalName`.

54.2 Cohérence entre facettes

La facette hypergraphique exprime la structure conceptuelle. La facette de base de données assure la persistance et une projection interrogeable. La cohérence exige que les deux facettes décrivent le même objet sous des angles opérationnels distincts.

Un rôle tel que `Owner` peut être représenté en mémoire par une référence d'objet et, dans la projection de base de données, par `Owner.Id`, `Owner.ClassId` et `Owner.FormalName`. Le modèle doit expliciter cette correspondance.

54.3 Frontières transactionnelles

Une transaction doit correspondre à une intention métier : créer un message, valider une tâche, clôturer un processus ou publier une réponse de service constituent des frontières naturelles. Un long script qui met à jour de nombreux objets indépendants sans modèle de processus doit, en revanche, attirer l'attention.

54.4 Cohérence au niveau du processus

Lorsque plusieurs objets doivent évoluer de manière coordonnée, introduisez un objet de processus. Il peut conserver un état intermédiaire, coordonner la validation, émettre des événements et permettre les reprises après échec. Cela est particulièrement important pour les processus asynchrones et les intégrations externes.

54.5 Règles de conception

Utilisez des identifiants stables. Projetez les références conceptuelles à l'aide de chemins de rôles. N'appelez `PostData` que lorsque l'objet a atteint un état valide. Employez des objets de processus pour assurer la cohérence entre plusieurs objets et des événements pour signaler l'achèvement des changements conceptuels.

54.6 Lorsque les métadonnées, le graphe sémantique et SQL évoluent ensemble

La génération des métadonnées, la matérialisation des types sémantiques, la migration SQL et la livraison des événements appartiennent à des domaines de cohérence distincts. La publication d'une génération de métadonnées APPLICATION ne démontre pas qu'un graphe `.sem` ou qu'une forme de base de données a changé de version en même temps. Pour une modification incompatible de la facette de base de données, le cycle d'évolution envisagé comprend *EXPAND* (extension), puis *MIGRATE* (migration des données), ensuite un *SWITCH* contrôlé des métadonnées et de la sémantique, et enfin *CONTRACT* (restriction). L'abstraction `PersistenceShape` et le planificateur de migration proposés décrivent un objectif architectural ; ils ne constituent pas une transaction distribuée générique entre magasins dont le fonctionnement serait vérifié dans le code fourni.

Lorsqu'un type sémantique natif évolue, un changement de signature structurelle crée une nouvelle représentation immuable : les instances historiques ne sont pas réorientées implicitement par la compilation incrémentale. Les garanties transactionnelles WAL/MVCC de l'hypergraphe persistant n'entraînent pas, à elles seules, l'atomicité SQL. Un système de production doit donc prévoir la réconciliation, la reprise idempotente ou la compensation si des validations franchissent ces frontières.

54.7 Synthèse

Identité et cohérence sont à la fois conceptuelles et techniques. Un modèle logiCells doit rendre l'identité explicite, aligner ses facettes et choisir des frontières transactionnelles adaptées aux intentions métier.

Chapitre 55

Exécution réactive et routage des événements

Les applications logiCells ne s'exécutent pas uniquement au moyen d'appels directs : elles réagissent également à des événements. Ceux-ci rendent les changements observables, permettent aux modèles de processus de coordonner leurs participants, autorisent l'actualisation des vues sans leur confier la logique métier et offrent aux composants de messagerie un moyen de propager le travail de façon asynchrone.

Ce chapitre explique la contribution du routage des événements à la programmation conceptuelle déclarative.

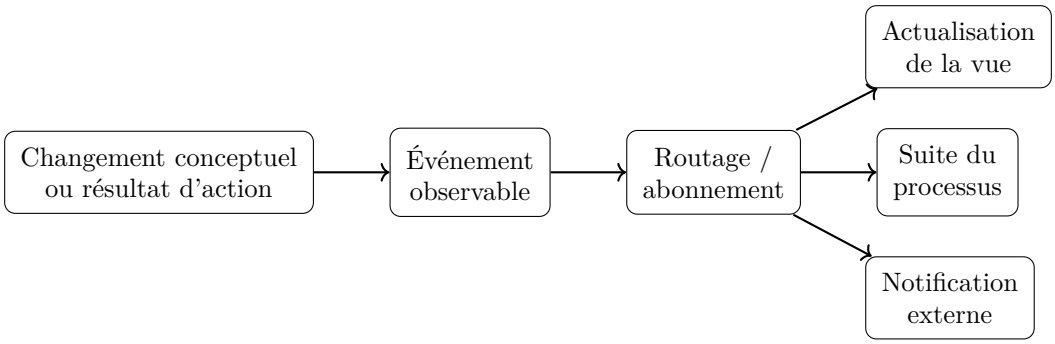


FIGURE 55.1 – Le comportement réactif repose sur des changements conceptuels observables et sur des frontières de routage explicites.

55.1 Les événements comme liens déclaratifs

Une déclaration d'événement associe une classe d'événement à une ou plusieurs actions. Elle peut également relier les paramètres de l'événement aux paramètres de ces actions.

```
events:
  - lbs.WorkItem.Started:
      actions:
        - action:
            path: './Action[@name=''OnWorkItemStarted'']'
            params:
              - Source:
                  eventParam: Source
              - Sender:
                  eventParam: Sender
```

Listing 55.1 – Acheminement d'un événement vers une action

L'émetteur n'a pas besoin d'appeler directement le destinataire. Celui-ci déclare simplement qu'il réagit à l'événement.

55.2 Les événements métier

Un événement métier signifie qu'un fait important s'est produit dans le domaine : démarrage ou fin d'un processus, réception d'un message, attribution d'un rôle, validation d'un document ou achèvement d'une tâche, par exemple.

Son nom doit exprimer le point de vue du métier, sans exposer les détails d'implémentation de l'interface utilisateur. Un événement métier bien nommé reste compréhensible sans que l'on sache quel écran y réagira.

55.3 Les événements visuels

Les événements visuels coordonnent les vues et les composants concrets. Une action sur un bouton, l'ouverture d'un cadre, un changement de sélection, la mise à jour d'un en-tête, la notification d'un jeu de données ou un événement de navigation relèvent d'abord de la couche de présentation.

Ils conservent une forme déclarative : un bouton peut déclarer l'émission d'un événement visuel, et un cadre peut déclarer sa réaction à cet événement. Cette logique reste explicite au lieu d'être enfouie dans un contrôleur d'interface codé en dur.

Le répartiteur au niveau de la page est le `PageControler`. Il reçoit les événements visuels, avertit le `PageModel` et les objets `objectBehind` des composants. Il ne transmet l'événement à un niveau supérieur que lorsque le modèle l'associe à une action, un processus, un service, un gestionnaire d'événements ou une opération du conteneur conceptuel. Cela évite que de simples interactions visuelles provoquent inutilement des appels au serveur ou des modifications du domaine.

55.4 Les événements de processus

Un modèle de processus émet fréquemment des événements à ses frontières. L'étape `Start` peut ainsi émettre `lbs.Process.Started`. De son côté, `Finalize` peut émettre `lbs.Process.Completed`. Entre ces deux étapes, d'autres événements peuvent demander une saisie, attendre une réponse distante ou informer un contrôleur.

Ce modèle favorise l'auditabilité : l'objet processus demeure responsable de la coordination, tandis que les réactions sont déclarées séparément.

55.5 Les événements de messagerie

Les événements de messagerie prolongent ce principe au-delà des frontières asynchrones. Une action locale peut construire un message, l'envoyer par le runtime et laisser un composant de messagerie l'acheminer localement, à distance ou selon les deux modalités.

Un modèle de message de la plateforme, tel que `lgc#custom_message` ou `lgc#text_message`, peut porter un état, des liens de conversation, des horodatages et des informations sur l'émetteur. Une application peut spécialiser le processus métier tout en réutilisant le mécanisme de distribution asynchrone de la plateforme.

55.6 Sources d'événements et héritage

Les déclarations peuvent restreindre la source d'un événement. C'est utile lorsqu'un contrôleur doit réagir uniquement aux événements émis par un type particulier de processus ou une famille d'objets. L'héritage intervient également lorsque le destinataire doit réagir aux événements provenant de descendants d'un type conceptuel.

La classe d'événement exprime *ce qui s'est produit*. La restriction de source précise *où cela s'est produit*.

55.7 Distribution et propagation

Les scripts du runtime construisent et distribuent souvent explicitement des événements :

```
Event := Self.ObjectsContainer.BuildMessage('lbs.WorkItem.Process.Start');
Self.ObjectsContainer.SendEvent(Event, Self.EventProxy);
```

Listing 55.2 – Construire et envoyer un événement

Les vues peuvent produire des événements visuels par l'intermédiaire d'un contrôleur de page :

```
Event := Self.PageControler.CreateVisualEvent(
  'lbs.Application.Navigation.Next',
  Source,
  Sender,
  Self.ObjectsContainer.MetaDataContainer,
  false);
Self.PageControler.DispatchVisualEvent(Event, StopNotification);
```

Listing 55.3 – Créer et distribuer un événement visuel

Les deux exemples reposent sur la même idée architecturale : l'émetteur publie un événement significatif, puis le runtime l'achemine vers les destinataires déclarés.

Chapitre 56

Services, actions et publication

Dans logiCells, un service ne constitue pas une couche technique isolée du modèle. Il prend généralement appui sur une action déclarée par un concept. Le nom de cette action désigne une capacité, ses paramètres définissent un contrat d'appel et son statut de publication indique qu'elle peut être exposée au-delà du script local qui l'implémente.

Ce chapitre décrit le chemin qui mène d'une action exécutable à un point d'accès de service. Il clarifie une ambiguïté fréquente : une action publiée n'est pas simplement une action visible dans l'interface utilisateur. C'est une capacité définie au niveau du modèle, accessible aux autres objets du runtime, aux agents, aux adaptateurs de services et aux services Web lorsque la configuration de l'application autorise cet accès.

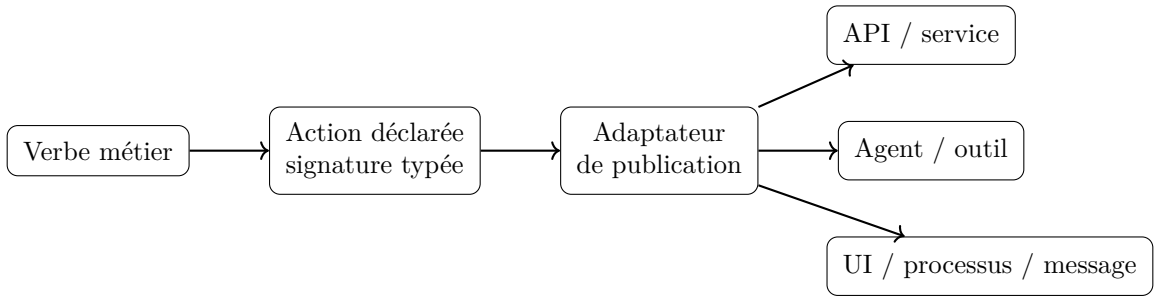


FIGURE 56.1 – Une capacité déclarée peut être exposée par plusieurs mécanismes d'accès sans dupliquer l'opération métier.

56.1 L'action comme frontière de service

L'action est la dimension opérationnelle d'un concept. Un concept peut définir des données, des relations, des collections, des états, des événements et des vues ; l'action, elle, déclare ce que ce concept peut faire. Lorsqu'elle est marquée comme publiée, le modèle indique que cette capacité fait partie du contrat public du concept.

Il importe de distinguer les situations suivantes :

Action interne Action utilisée par un script, un modèle de vue, un gestionnaire d'événements ou une implémentation de processus. Elle peut rester privée au modèle.

Action publiée Action volontairement rendue accessible comme capacité invocable. Elle peut être appelée par d'autres éléments du modèle et sélectionnée pour être publiée sous forme de service.

Action exposée Action publiée et associée à un mécanisme d'accès concret : service Web, groupe de services, commande d'agent ou passerelle de messagerie.

Cette séparation permet au modèle de rester stable lorsque la stratégie de publication externe évolue. Une même action conceptuelle peut être utilisée depuis un formulaire local, un processus serveur, une tâche planifiée, un service Web ou un agent distant, sans dupliquer la capacité métier.

56.2 La signature déclarative

La signature d'une action publiée est définie dans le modèle. Elle comprend le nom de l'action, son porteur, son mode d'exécution et ses paramètres. Les paramètres ne sont pas de simples détails d'implémentation : ils établissent le contrat que les appelants externes doivent respecter.

Le porteur détermine la façon dont le service est adressé. Une action `ObjectItem` s'exécute sur un objet conceptuel concret. Une action `ClassItem` s'exécute au niveau du concept ; elle convient notamment aux capacités de création, de recherche, d'import, de publication ou d'administration.

56.3 Accès par service Web

Une action publiée peut être appelée par un service Web lorsqu'un adaptateur est configuré pour l'exposer. Cet adaptateur n'invente pas une seconde API métier : il associe l'appel externe à la signature d'action déjà déclarée dans le modèle.

Le mappage s'organise généralement autour de quatre éléments :

1. Le concept cible, identifié par son identifiant de classe ou son chemin dans le modèle.
2. L'action cible, identifiée par son nom publié.
3. Les paramètres d'entrée, construits à partir de la charge utile de la requête.
4. Le paramètre de retour ou l'objet de réponse, reconverti en réponse du service.

La publication est ainsi pilotée par le modèle. L'API n'est pas une copie du domaine entretenue manuellement : c'est une projection contrôlée des capacités conceptuelles sélectionnées.

56.4 Appels de service synchrones

Le mode de publication le plus simple est synchrone. Un client appelle un service Web ; l'adaptateur résout le concept ou l'objet cible, invoque l'action publiée, attend son résultat et renvoie la réponse.

Ce modèle convient lorsque l'appelant attend une réponse immédiate et que l'action est suffisamment courte pour s'exécuter dans le cadre d'un échange requête-réponse. Les cas habituels comprennent la validation, la recherche d'informations, la création d'un objet léger, le calcul d'un état ou la consultation d'une projection synthétique.

56.5 Passerelles de service asynchrones

Toutes les actions publiées ne doivent pas s'exécuter de façon synchrone. Les opérations longues, distantes ou faisant intervenir plusieurs participants se modélisent plus naturellement au moyen d'un objet processus ou d'une passerelle de messagerie. Le service Web peut alors se limiter à démarrer le processus et à renvoyer un accusé de réception ; la suite se déroule par événements et messages.

Un enchaînement fréquent est le suivant :

```
Requête externe
  -> action publiée StartProcess
  -> création de l'objet processus
  -> émission d'un événement de démarrage
  -> retour d'un accusé de réception
  -> poursuite du processus par événements/messages
```

Listing 56.1 – Point d'entrée synchrone et poursuite asynchrone

L'action demeure la frontière de service, mais le processus porte la logique de coordination.

56.6 Sécurité et périmètre de publication

La publication ne vaut pas autorisation. Marquer une action comme publiée signifie qu'elle peut, en principe, être appelée. L'accès effectif dépend encore de la configuration des services, des rôles, de la visibilité contextuelle, de l'authentification, de l'autorisation et des règles de validation.

Un modèle robuste traite donc la publication en plusieurs étapes :

1. L'action est déclarée.
2. L'action est marquée comme publiée.
3. Un adaptateur de service l'expose.
4. Les règles de sécurité déterminent qui peut l'appeler.
5. Les règles de validation déterminent si l'appel est recevable dans le contexte courant.

56.7 Règles de conception

Le nom d'une action publiée doit exprimer une capacité métier et non un mécanisme de transport. On privilégiera `Verify`, `Enroll`, `Start`, `Finalize`, `Import`, `Synchronize` ou `Search`, plutôt que des noms liés à HTTP ou à l'interface utilisateur.

Une action publiée doit avoir une signature stable, des types de paramètres explicites et une convention de retour claire. Lorsqu'elle coordonne plusieurs objets, mieux vaut introduire un modèle de processus que dissimuler l'orchestration dans un script de service.

Chapitre 57

Frontières du runtime et points d'extension

Une application logiCells est extensible parce que son runtime sépare les déclarations des modèles de leurs implémentations concrètes. Ce chapitre présente les principales frontières où interviennent les extensions.

57.1 Frontière du modèle

C'est la frontière où sont déclarés concepts, rôles, facettes, collections, actions, événements et vues. Elle doit rester stable et lisible : elle constitue le contrat public du modèle applicatif.

57.2 Frontière d'implémentation

Certains éléments du modèle sont exécutés par des classes d'implémentation concrètes. Une action peut être scriptable ou confiée à une classe enregistrée. Un composant fonctionnel peut être abstrait dans le modèle, puis rendu par un composant concret enregistré pour une plate-forme. Un adaptateur de service peut exposer une action via HTTP ou un autre protocole.

57.3 Frontière de publication

Les actions publiées forment une frontière contrôlée entre le runtime et les accès externes. Une même action publiée peut être appelée par une interface, un service, un agent ou un adaptateur de service web ; le modèle reste la source de sa signature.

57.4 Frontière de messagerie

La messagerie crée une frontière asynchrone : un événement local peut devenir un message, et un message distant peut être converti en événement local. Lorsque la distinction est pertinente, le modèle doit préciser quels événements sont locaux, distants ou les deux.

57.5 Règles de conception

Conservez la signification métier dans le modèle. Placez les comportements remplaçables derrière des implémentations enregistrées. N'exposez pas comme services des actions utilitaires internes. Traitez la publication web, la messagerie et l'activation de l'interface comme des adaptateurs des capacités du modèle.

57.6 Les liaisons d'implémentation ne sont pas des concepts métier

Certaines déclarations de métadonnées lient un artefact conceptuel à une implémentation du runtime enregistrée, par exemple un générateur ou une implémentation d'objet sélectionnée par un identifiant de liaison. Cet identifiant sert à choisir une implémentation ; il ne doit pas imposer le nom du concept métier, le verbe de l'ontologie, la capacité publiée ou la représentation que l'utilisateur se fait du domaine.

Lorsque des alias déclaratifs stables existent, préférez des noms orientés capacités et laissez l'enregistrement les résoudre en implémentations concrètes. Si la grammaire exige encore un nom de classe concret, maintenez cette liaison à la frontière du runtime en l'expliquant comme telle. Le sens de l'abstraction est ainsi préservé : les concepts métier sélectionnent des capacités, puis ces capacités sont associées à des implémentations.

57.7 Synthèse

Les frontières du runtime rendent les applications logiCells évolutives. Le modèle définit le sens ; les implémentations, services et composants réalisent l'exécution concrète.

Chapitre 58

Cycle de vie du runtime, propriété des ressources et arrêt

La durée de vie fait partie du contrat du runtime. Des défaillances qui semblent relever des données, de la messagerie, de l'interface ou de l'IA proviennent souvent d'un objet, d'une fonction de rappel, d'un worker ou d'un résultat natif qui survit à la ressource dont il dépend.

Ce chapitre présente les règles publiques. Les détails propres aux langages d'implémentation sont traités dans les guides d'ingénierie du runtime.

58.1 Durée de vie mémoire et durée de vie opérationnelle

Conserver une référence ne signifie pas que la ressource soit encore opérationnelle. Un canal distant peut exister après sa fermeture ; un objet de service, après son arrêt ; un runtime d'inférence résident, après la réinitialisation de son cache contextuel.

Distinguez systématiquement :

— **la durée de vie mémoire** : l'objet ou la valeur peuvent-ils encore être référencés ?
— **la durée de vie opérationnelle** : la ressource est-elle démarrée, connectée, abonnée ou encore prête à accepter du travail ?

58.2 Catégories de propriété des ressources

Au niveau applicatif, quatre catégories sont importantes :

Contrats managés ou comptés par références Conserver la référence publique maintient en vie l'enveloppe ou le contrat ; cela ne remplace pas les opérations explicites d'arrêt ou de fermeture.

Objets explicitement détenus par le runtime Le contrat du constructeur, de la fabrique ou de la méthode indique à qui revient la libération de l'objet.

Propriété exercée par le framework client Les composants de client riche peuvent également relever du cycle de propriété des composants d'interface et des abonnements aux événements.

Valeurs de bas niveau du runtime Certaines valeurs hypergraphiques ou natives avancées obéissent à un contrat de conservation propre au runtime et ne doivent pas être traitées comme de simples objets du langage hôte.

58.3 Ordre d'arrêt des services et des workers

Un arrêt déterministe procède généralement des producteurs vers l'état partagé :

1. cesser d'accepter de nouvelles commandes et actions utilisateur ;
2. désabonner les fonctions de rappel et les récepteurs d'événements ;
3. arrêter les minuteries, ordonnanceurs et producteurs en arrière-plan ;
4. annuler ou terminer les travaux en cours selon la politique applicative ;
5. fermer les canaux et transports distants ;
6. arrêter les services et synchroniseurs ;
7. attendre les workers qui peuvent encore accéder à l'état applicatif ;
8. libérer les jeux de données, fournisseurs, modèles et conteneurs du runtime ;
9. décharger les modules et désenregistrer les fabriques globales en dernier.

La destruction ne remplace pas une transition explicite d'arrêt ou de fermeture lorsque des opérations peuvent encore être en cours.

58.4 Propriété au niveau de l'ABI

L'ABI ajoute un contrat de propriété lisible par machine. Un binding doit respecter les politiques de détention et de libération inscrites dans le descripteur, sans les déduire du nom d'une méthode ou du type de retour. Le vocabulaire détaillé des modes de propriété et de libération figure dans Chapitre 64, “Modèles déclaratifs et ABI du runtime”.

58.5 Cycle de vie des ressources d'IA résidentes

Les poids du modèle résident et l'état propre à un contexte ont des durées de vie différentes. Un nouveau contexte d'inférence peut réinitialiser son cache sans décharger les poids immuables. Recréer le runtime résident pour chaque jeton ou requête annule l'intérêt de cette résidence et peut masquer des défauts de cycle de vie derrière des initialisations répétées.

La même règle s'applique à l'inférence locale gérée par un fournisseur : l'arrêt du traitement des requêtes et le déchargement du modèle constituent deux transitions distinctes.

58.6 Cycle de vie du client riche

Une vue, un DataSet/DataCursor, une continuation asynchrone et un objet de page ou contrôleur peuvent avoir des durées de vie différentes. Une fonction de rappel du client ne doit pas mettre à jour une page déjà détruite. Un worker de synchronisation ne doit pas conserver un objet client au seul motif qu'une opération demeure en attente. Le guide *Rich Client & View Architecture* présente les modèles propres aux clients.

58.7 Liste de contrôle du développeur

Avant de considérer l'arrêt comme terminé, posez-vous les questions suivantes :

— Un thread ou une continuation peut-il encore appeler le code applicatif ?
— Une source d'événements peut-elle notifier un objet en cours de destruction ?
— Une réponse distante peut-elle arriver après la disparition du contexte de requête ?
— Une file, un fournisseur ou un cache peut-il être libéré alors qu'un producteur l'utilise encore ?
— Chaque service démarré a-t-il reçu sa transition correspondante d'arrêt ou de fermeture ?
— Les résultats ABI détenus ont-ils été libérés exactement une fois ?
— Les registres globaux et les modules natifs ont-ils été conservés jusqu'à ce que leurs derniers utilisateurs aient terminé ?

Un chemin d'arrêt déterministe permet de vérifier concrètement que la propriété des ressources est correctement modélisée.

Huitième partie

CLI, API et intégration externe

Chapitre 59

Utiliser l'outil en ligne de commande `cell`

Les commandes présentées dans ce chapitre proviennent du code source de l'exécutable CLI disponible. Le dépôt contient également des textes qui désignent cette façade sous le nom `lgc`. Tant que le conditionnement logiciel n'a pas harmonisé ces appellations, ce guide suit le contrat d'aide de l'exécutable, qui se présente sous le nom `cell`.

59.1 Architecture

L'interface en ligne de commande (CLI) sert de façade au Tooling Host. Les règles métier relatives à l'espace de travail, aux applications, aux packages et aux métadonnées restent dans les services d'outillage : la CLI ne doit pas les réimplémenter implicitement. Le chemin d'exécution habituel passe par JSON-RPC sur l'entrée/sortie standard (STDIO) vers `logicells-tooling`. Les journaux sont séparés du document de résultat final.

59.2 Options globales

```
cell [--json] [--quiet] [--verbose]
     [--tooling <path>] [--tooling-endpoint <url>]
     [--timeout <ms>] [--lgc <file.lgc>]
     [--param name=value] [--workspace-folder <path>] <command>
```

L'option `-json` réserve la sortie standard à une enveloppe stable et lisible par machine. Les messages de diagnostic sont dirigés hors de ce flux de résultats.

59.3 Commencer par `doctor`

```
cell doctor
cell --json doctor
```

Le diagnostic présente la matrice réelle des capacités exposées par le Tooling Host sélectionné. L'absence d'une capacité doit rester visible : la CLI ne doit pas basculer silencieusement vers une seconde implémentation.

59.4 Explorer l'espace de travail, les applications et les packages

```
cell workspace inspect [path]
cell application inspect <application>
cell application validate <application>
cell application packages <application>
cell package inspect <package>
cell package validate <package>
```

La construction et la compilation d'une application sont deux commandes distinctes :

```
cell application build <file.lgc>
cell application compile <file.lgc>
cell application compile plan <file.lgc>
```

La planification de compilation n'est exposée publiquement que si le Tooling Host annonce la capacité correspondante.

59.5 Écrire et inspecter des modèles MODEL

```
cell declarative inspect <file>
cell declarative validate <file>
cell model file inspect <file>
cell model source inspect <file>
cell model metadata inspect <file>
cell model source convert <file> --format xml|yaml [--output <file>]
```

La CLI expose également des commandes structurées d'ajout, de modification et de suppression de champs, de relations et d'actions, avec l'option `-dry-run`. Il convient de les privilégier pour les modifications réalisées par un agent ou par un automatisme : les services d'outillage appliquent alors les mêmes règles de résolution et de validation des modèles que dans les parcours interactifs.

59.6 Explorer la sémantique

L'exécutable disponible permet d'inspecter la sémantique d'une application assemblée, d'un fichier H-Logic ou d'un projet :

```
cell semantic inspect <file.lgc>
cell semantic source inspect <file.hlg>
cell semantic project inspect <path>
cell semantic nodes <application|source|project> <path>
cell semantic node inspect <mode> <path> <node>
cell semantic neighbors <mode> <path> <node>
cell semantic relations <mode> <path> <node>
cell semantic stats <mode> <path>
cell semantic path <mode> <path> <from> <to>
cell semantic export <mode> <path> <node>
cell semantic query <mode> <path> <node>
cell semantic explain <mode> <path> <node>
```

59.7 Scripts, services, runtime et débogueur

```
cell script languages
cell script validate <source> [--language <name>]
cell script compile <source> --output <ast> [--language <name>]
cell script inspect <ast>
cell script run <source|ast> [--language <name>]

cell services inspect <service-file>
cell services status <file.lgc>
cell server inspect <file.lgc>
cell server start <file.lgc> [--foreground]
```

```
cell server list
cell server status [pid]
cell server stop [pid]
```

Les commandes du débogueur utilisent les identifiants des processus du runtime. Elles couvrent notamment la pile, les variables locales, l'évaluation d'expressions, la poursuite de l'exécution, les différents pas d'exécution, l'arrêt et l'ajout ou la suppression de points d'arrêt.

59.8 Publication d'applications : MCP et OpenAPI

L'exécutable expose des compilateurs de publication symétriques pour les applications et les packages :

```
cell mcp compile application <file.lgc> --output <dir> [--generate-doc]
cell mcp compile package <package> --output <dir> [--merge] [--generate-doc]
cell openapi compile application <file.lgc> --output <dir> [--generate-doc]
cell openapi compile package <package> --output <dir> [--merge] [--generate-doc]
```

Ces commandes projettent le contrat de l'application ou du package sélectionné vers des artefacts propres au protocole. Les artefacts MCP d'une application peuvent être hébergés en déclarant le serveur MCP dans la configuration Agents de son fichier `.lgc`. Voir Chapitre 61, "Publier des applications avec OpenAPI et MCP".

59.9 Exécution des agents et sessions persistantes

La commande d'exécution ponctuelle est la suivante :

```
cell agent run <task> [--agent-id <id>] [--subject-id <id>]
               [--purpose <name>] [--profile <name>]
               [--knowledge-source <path>] [--knowledge-mode source|project]
               [--knowledge-answer deterministic|verbalized]
               [--knowledge-query <hlogic>]
               [--knowledge-nested-query <hlogic>]
               [--max-turns <n>] [--max-provider-iterations <n>]
               [--max-iterations <n>] [--interactive]
               [--on-confirmation fail|reject]
```

Pour une exécution persistante, la CLI propose :

```
cell agent session open [--agent-id <id>]
cell agent session goal <session-id> <goal> [options]
cell agent session pending <session-id>
cell agent session confirm <session-id> <request-id>
cell agent session reject <session-id> <request-id>
cell agent session snapshot <session-id>
cell agent session evaluate <session-id>
cell agent session close <session-id>
```

Ces commandes nécessitent un `-tooling-endpoint` persistant, qui conserve l'état entre deux invocations de la CLI. Voir Chapitre 69, "Runtime Agent, objectifs et sessions persistantes" pour le cycle de vie des sessions et les options de requête de connaissances.

Les commandes disponibles dépendent de la version de la CLI et du Tooling Host installés. `cell application compile plan` inspecte un plan sans l'appliquer ; `compile` demande une compilation incrémentale admissible ; `build` déclenche une construction complète. Les modifications MODEL pilotées par les fichiers sources diffèrent des opérations sur les métadonnées compilées APPLICATION, qui ne réécrivent pas nécessairement les fichiers `.model.yaml`. Consultez l'aide de la commande avant d'automatiser son usage.

La CLI expose aussi des parcours de découverte et de publication qui complètent les commandes de script et d'API :

```
cell transform list
cell transform capabilities
cell mcp compile application <file.lgc> --output <dir>
cell openapi compile application <file.lgc> --output <dir>
cell doc compile application <file.lgc> --output <dir> --format xml|html
```

La production documentaire XML est accessible par Tooling. Contrôlez l'option `--format xml|html` et les fichiers générés avec la version déployée. Publier un descripteur OpenAPI ou MCP n'applique pas automatiquement la confirmation Agent aux appels externes ; vérifiez séparément l'autorisation de ces appels.

La façade de débogage runtime prise en charge se connecte à `sdlgc` avec `--ast-debug`. Il faut utiliser le `processpid` logique retourné par le runtime de débogage, et non l'identifiant du processus du système d'exploitation. Les formes canoniques des nouvelles commandes comprennent `cell debug break add`, `break clear`, `step-in` et `step-out`. Le débogage interactif d'un script intégré à YAML exige en outre le mappage des sources décrit au Chapitre 75, "Débogage source de FunctionalP dans YAML".

Ce guide décrit `cell agent run` et `cell agent session`. Les commandes de développement de l'espace de travail et les profils d'agents supplémentaires, lorsqu'ils sont proposés, doivent être vérifiés dans la CLI et le registre Tooling installés avant toute automatisation.

59.10 Règle de conformité au contrat

L'aide de l'exécutable et la négociation des capacités du Tooling Host font autorité pour la disponibilité opérationnelle des commandes. Une commande prévue mais non disponible peut être décrite uniquement si son statut « prévu » est indiqué clairement ; aucune capacité non annoncée par les services d'outillage ne doit être présentée comme implémentée.

Chapitre 60

Publication des API et des services

Pour les mécanismes qui relient les métadonnées au runtime et pour les familles de services avancés, voir Chapitre 48, “Services déclaratifs du runtime et modèles d’exécution”.

Dans logiCells, une API doit être comprise comme une projection publiée de capacités conceptuelles. La surface de service ne constitue pas un modèle parallèle : c’est une couche d’accès contrôlé aux concepts, objets, actions, événements et processus.

60.1 Objectif

Ce chapitre explique comment concevoir des API et des services à partir des capacités publiées par les modèles. Il examine le lien entre les signatures d’actions, les points de terminaison de service, le mappage des requêtes et réponses et la conception de services orientés processus. Le parcours concret de compilation et d’hébergement OpenAPI et MCP est présenté dans Chapitre 61, “Publier des applications avec OpenAPI et MCP”.

60.2 De la signature d’action à l’opération API

Une action publiée fournit une description naturelle de l’opération. Son nom devient le nom de l’opération ; l’objet ou la classe qui la porte détermine la ressource cible ou le contexte de service ; ses paramètres deviennent les champs de la requête ; et son paramètre de retour constitue la réponse.

```
ClassItem: lbs#account_service
Action:    SearchAccount
Params:    Query:String, MaxCount:Integer, Result:List

Possible endpoint:
POST /services/lbs/account-service/SearchAccount
```

Listing 60.1 – D’une action conceptuelle à une opération de service web

La forme exacte du point de terminaison relève de l’adaptateur. L’opération conceptuelle, elle, reste inchangée.

60.3 Mappage des requêtes et des réponses

Les champs d’une requête doivent correspondre aux paramètres déclarés. Les types primitifs sont mappés directement. Les types conceptuels sont résolus au moyen d’identifiants, d’identifiants de classe, de références d’objets ou de projections sérialisées. Les valeurs de retour peuvent être des valeurs primitives, des références d’objets, des listes, des objets d’état ou des accusés de réception de message.

60.4 Groupes de services

Les groupes de services organisent les points de terminaison et les responsabilités d'intégration. Un groupe peut exposer des services web synchrones, des passerelles de messagerie, des agents distants ou des adaptateurs spécialisés. Il ne doit pas redéfinir la sémantique métier : son rôle est d'orienter les appels externes vers les actions ou les messages conceptuels.

60.5 Publication d'actions par des services web

Un point de terminaison de service web peut exposer directement une action publiée. Il s'agit du modèle le plus transparent, car l'API externe reflète alors la capacité conceptuelle. L'adaptateur valide la requête, contrôle l'accès, résout la cible, appelle l'action, puis renvoie le résultat.

60.6 Pages HTTP virtuelles et sélection des routes

Une publication HTTP peut également définir des pages virtuelles. Une définition `serviceGroupDefinition` de `type: HTTP` porte les règles `virtualPages`. Dans l'exemple documenté, `GET /users` correspond à une entrée `method/path`, sélectionne le gestionnaire `PathRequestWebPage` enregistré sous son nom ABI, puis applique la cible conceptuelle `rewrite` déclarée. Les ressources d'extension de page HTTP utilisent l'extension `.lgp`. La déclaration de route et le nom du fichier de ressource de page relèvent de deux préoccupations distinctes. Voir Chapitre 48, "Services déclaratifs du runtime et modèles d'exécution" pour la forme YAML et la chaîne de sélection des requêtes.

60.7 API orientées processus

Les opérations de longue durée ne doivent pas être modélisées comme un unique appel API bloquant. L'API doit plutôt exposer une action publiée, par exemple `Start`, `Submit`, `Request` ou `Finalize`. Celle-ci crée ou met à jour un objet de processus et renvoie un accusé de réception ou une référence à ce processus.

60.8 Contrats lisibles par machine : OpenAPI et MCP

OpenAPI et MCP sont deux surfaces de publication d'un même runtime conceptuel. Elles ne doivent pas inventer un second système de types. Un schéma OpenAPI généré et le `inputSchema` d'un outil MCP doivent conserver les distinctions présentes dans le modèle : valeurs primitives, références d'objets typées, rôles relationnels, paramètres obligatoires et formats pris en charge.

On en déduit une règle pratique de conformité :

```
concept/action metadata
-> runtime parameter and relation contract
-> OpenAPI request/response schema
-> MCP tool inputSchema/output contract
-> adapter validation
-> runtime execution
```

Listing 60.2 – Un contrat projeté vers plusieurs surfaces de publication

Si un chemin de rôle est nécessaire pour écrire ou résoudre une relation, le contrat lisible par machine doit exposer une structure suffisante pour permettre au client de le découvrir. Si le développeur doit déduire des chemins de rôle cachés en examinant des projets existants, le contrat généré est incomplet.

MCP exige la même rigueur que HTTP. Un appel d'outil doit être validé par rapport au schéma déclaré avant toute distribution. Les arguments inconnus, les paramètres obligatoires manquants, les dates et heures mal formées, les identifiants de classe non résolus et les références d'objets de forme invalide doivent être rejetés avant l'exécution de l'action sous-jacente. Un manifeste MCP n'est donc pas seulement un catalogue d'outils : c'est un contrat de frontière exécutable.

Pour `ClassId`, les adaptateurs de publication doivent privilégier une résolution exacte. Si une valeur possède la syntaxe d'un identifiant de classe, la convertir silencieusement en recherche par nom peut masquer des erreurs de casse ou d'espace de noms, et entraîner des comportements différents entre OpenAPI, MCP et les appels natifs.

60.9 Sémantique des erreurs et de la validation

Une erreur doit être exprimée à la frontière où la valeur invalide est reconnue pour la première fois. Les erreurs techniques relèvent de l'adaptateur de service ; les erreurs de validation, de l'action ou du processus. Les exceptions métier doivent être représentées par des états, des événements ou des objets de réponse structurés lorsque l'appelant doit pouvoir les traiter.

Un adaptateur de publication doit renvoyer des diagnostics assez précis pour identifier le contrat en défaut : opération ou outil, paramètre ou rôle, valeur reçue, type ou format attendu et, le cas échéant, `ClassId` non résolu. Il ne faut ni transformer une référence typée invalide en relation vide, ni accepter une valeur `DateTime` incorrecte pour n'échouer que plus tard, avant la persistance.

60.10 Règles de conception

Concevez l'API à partir du modèle, vers l'extérieur. Publiez des actions stables. Gardez les adaptateurs de service aussi simples que possible. Utilisez des processus pour les opérations de longue durée et des messages pour leur continuation asynchrone. Employez des types explicites et des conventions de retour cohérentes. Évitez d'exposer les noms des tables internes ou des détails propres aux vues.

Chapitre 61

Publier des applications avec OpenAPI et MCP

logiCells peut projeter une application ou un package vers des artefacts de publication lisibles par machine, pour OpenAPI comme pour MCP. Il s'agit de deux projections protocolaires du même modèle applicatif, et non de deux modèles métier distincts.

Le parcours développeur est volontairement explicite :

```
application / package
        |
        +--> cell openapi compile ... --> artefacts OpenAPI
        |
        +--> cell mcp compile ...     --> artefacts MCP
```

La projection part de l'application et de ses capacités publiées. Les artefacts propres au protocole sont générés dans un répertoire de sortie et peuvent ensuite être hébergés par la configuration de service correspondante du runtime.

61.1 La publication part du contrat applicatif

Un protocole externe doit exposer des capacités possédant déjà un sens dans le modèle conceptuel. Les actions publiées constituent la principale frontière opérationnelle. Leurs noms déclarés, paramètres, types et états de publication forment le contrat dont dérivent les surfaces protocolaires externes.

Par exemple, la publication des actions peut être pilotée depuis la CLI :

```
cell application action add <file.lgc> <classId> <name> --published
cell application action update <file.lgc> <classId> <name> --published
cell application action update <file.lgc> <classId> <name> --unpublished
```

La même distinction s'applique aux fichiers MODEL avec `cell model action add/update`. La publication reste donc une décision du modèle avant d'être une décision de protocole.

61.2 Compiler une application pour MCP

La CLI actuelle expose la compilation MCP pour les applications complètes comme pour les packages :

```
cell mcp compile application <file.lgc> --output <dir> [--generate-doc]
cell mcp compile package <package> --output <dir> [--merge] [--generate-doc]
```

Voici une compilation concrète vers un chemin local au projet :

```
cell mcp compile application /path/to/app.lgc \
  --output /path/to/build/mcp
```

La commande écrit la projection MCP dans le répertoire indiqué. Utilisez `-generate-doc` si la documentation générée doit accompagner les artefacts de publication. Pour les packages, `-merge` contrôle le comportement de fusion exposé par l'exécutable actuel.

61.2.1 Héberger la surface MCP générée

La surface MCP générée est mise à disposition en déclarant le serveur MCP dans la configuration `.lgc` de l'application utilisée par Agents. Au démarrage de l'application configurée, le runtime Agents lance le serveur MCP répertorié. Un client MCP externe obtient ainsi une frontière protocolaire vers les capacités applicatives publiées qui ont été sélectionnées.

Le cycle de vie est le suivant :

```
application .lgc
   -> cell mcp compile application
   -> artefacts MCP generes
   -> serveur MCP reference dans la configuration Agents .lgc
   -> demarrage du serveur par le runtime Agents
   -> connexion des clients MCP externes
```

La configuration serveur dans un fichier `.lgc` dépend de la version distribuée. Reprenez le schéma de configuration livré avec votre runtime et vérifiez l'activation du serveur, le transport et les droits d'accès avant d'ouvrir le point d'accès à des clients tiers.

61.3 Compiler une application pour OpenAPI

OpenAPI suit un parcours équivalent pour la publication des applications et des packages :

```
cell openapi compile application <file.lgc> --output <dir> [--generate-doc]
cell openapi compile package <package> --output <dir> [--merge] [--generate-doc]
```

Par exemple :

```
cell openapi compile application /path/to/app.lgc \
  --output /path/to/build/openapi
```

OpenAPI et MCP doivent être considérés comme deux projections parallèles. Un développeur ne doit pas réécrire une règle métier pour OpenAPI, puis une seconde fois pour MCP. Le modèle et les actions publiées restent l'autorité ; les compilateurs de protocole en dérivent les contrats externes.

61.4 Publication au niveau de l'application ou du package

Les deux familles de protocoles acceptent des entrées applicatives ou des packages. Le choix dépend de la frontière à publier :

Compilation d'application À employer lorsque le contrat externe appartient à l'application assemblée décrite par une configuration `.lgc` concrète.

Compilation de package À employer lorsque le contrat publié provient de la frontière d'un package réutilisable. Cette commande expose aussi l'option facultative `-merge`.

Avant de publier une application, vérifiez-la utilement avec les commandes :

```
cell application validate <file.lgc>
cell application inspect <file.lgc>
cell application packages <file.lgc>
cell application models <file.lgc>
```

Pour un package :

```
cell package validate <package>
cell package inspect <package>
cell package item inspect <package> <name>
```

61.5 Les deux directions MCP

La prise en charge de MCP doit être considérée dans deux directions distinctes.

logiCells comme serveur MCP Une application ou un package est compilé vers une surface MCP publiée, puis hébergé selon la configuration du runtime. Les clients MCP externes et les orchestrateurs d'agents peuvent accéder à la frontière applicative exposée.

logiCells comme client MCP Le runtime Agent peut découvrir et invoquer des capacités de serveurs MCP externes. Ce chemin possède sa propre gouvernance et peut distinguer les opérations en lecture seule de celles qui produisent des effets.

Ces directions répondent à deux besoins différents. Le serveur publie les capacités logiCells vers l'extérieur ; l'opérationnalisation cliente permet aux agents logiCells d'utiliser des outils externes.

61.6 Discipline des types et de la validation

Les schémas de requêtes/réponses OpenAPI et d'outils MCP représentent, au niveau des protocoles, un contrat typé existant. Les adaptateurs de publication doivent préserver l'information de type du modèle, sans l'affaiblir.

En particulier :

— les paramètres obligatoires doivent rester obligatoires à la frontière protocolaire ;
— les valeurs typées mal formées doivent être rejetées avant le dispatch de l'action sous-jacente ;
— les identifiants conceptuels doivent préserver exactement leur sémantique d'identité ;
— une action publiée ne doit pas devenir appelable sous un contrat de protocole plus permissif que son contrat natif ;
— la documentation générée doit être régénérée lorsque le contrat applicatif publié évolue.

61.7 Parcours de publication recommandé

Un parcours reproductible est le suivant :

1. Valider l'application ou le package.
2. Inspecter les actions à publier et vérifier leur statut `published`.
3. Compiler la projection OpenAPI et/ou MCP dans un répertoire de sortie propre.
4. Générer la documentation du protocole au besoin avec `-generate-doc`.
5. Configurer la frontière de service du runtime dans le fichier `.lgc` de l'application.
6. Démarrer l'application et le runtime, puis vérifier la surface publiée avec un client externe du protocole.
7. Recompiler après une modification de signature d'action publiée ou de métadonnées du modèle pertinentes.

61.8 Projeter un protocole n'autorise pas l'exécution

La génération d'une surface protocolaire ne donne pas automatiquement des droits à tous les appelants. La publication, la sécurité du runtime, les politiques et les contrôles des opérations à effets sont des sujets distincts. Une capacité peut figurer dans le contrat généré tout en exigeant une autorisation ou une confirmation à l'exécution.

Cette distinction est particulièrement importante avec MCP, puisque l'appelant peut lui-même être un agent. Découverte, autorisation et exécution constituent trois étapes séparées.

61.9 Appels MCP entrants et autorité du runtime

Pour les appels MCP entrants, distinguez trois questions :

1. L'action est-elle publiée et donc visible dans la surface MCP générée ?
2. L'appelant est-il autorisé à demander cette opération dans le contexte d'exécution courant ?
3. Si l'opération produit des effets, une confirmation ou des preuves complémentaires sont-elles exigées avant le dispatch ?

La publication rend une action accessible par un protocole, mais ne remplace pas l'autorisation métier. Pour toute action modifiant l'état, vérifiez l'authentification du client, l'application des permissions et, lorsque le scénario l'exige, la confirmation avant exécution. Ces conditions doivent être testées sur le point d'accès entrant effectivement déployé. Le contrat général figure dans Chapitre 70, "Contrat d'exécution gouvernée des agents".

61.10 Configurer et tester le serveur publié

Utilisez un projet initial ou un exemple de configuration officiel correspondant à la version de logiCells installée. Vérifiez l'artefact MCP produit, les paramètres de transport et le composant Agents chargé d'activer le serveur. Testez ensuite un appel autorisé, un appel refusé et une opération soumise à confirmation avant toute ouverture à des clients externes.

Chapitre 62

Adaptateurs déclaratifs d'API externes

De nombreuses applications logiCells doivent consommer des API externes. L'essentiel est de ne pas confondre l'API extérieure avec le modèle métier : elle doit être encapsulée dans un adaptateur déclaratif qui expose les requêtes, réponses, règles de pagination, erreurs et services sous forme d'objets conceptuels.

Ce modèle d'intégration permet de se connecter à des systèmes externes tout en préservant une surface conceptuelle stable pour les fournisseurs, les jeux de données, les actions et les processus.

62.1 Les API externes comme frontières conceptuelles

Un adaptateur d'API externe doit être modélisé comme une frontière explicite entre le runtime logiCells et le système distant.

```
logiCells action or provider
  -> API service model
  -> request concept
  -> remote endpoint
  -> response concept
  -> collection provider or process update
```

Listing 62.1 – Chaîne d'adaptation d'une API externe

L'adaptateur doit décrire le contrat. Les URL distantes, les paramètres de requête, les en-têtes et les règles de pagination ne doivent pas être dispersés dans les composants visuels ou les processus métier.

62.2 Concepts de requête, de réponse et de service

Un adaptateur robuste distingue généralement trois concepts.

Concept	Responsabilité
Request	Déclare l'URI, la méthode, les en-têtes, les paramètres et le contenu transmis.
Response	Déclare l'état de réussite, le statut technique, les erreurs, les données retournées et les métadonnées de pagination.
Service	Relie un type de requête à un type de réponse et fournit la frontière d'intégration exécutable.

Cette séparation rend l'intégration vérifiable et permet de tester les requêtes et les réponses indépendamment de la couche de présentation.

62.3 Modèle générique d'API paginée

L'exemple suivant s'appuie sur une ressource métier générique. Il illustre uniquement le modèle public, sans présupposer une API propre à un client.

```yaml
models:
  concepts:
    - RemoteWorkItemRequest:
        type: Entity
        id: lbs#remote_work_item_request
        facets:
          - uri:
              type: hypergraph
              fields:
                - Endpoint:
                    type: String
                    kind: attribute
                - Page:
                    type: Integer
                    kind: attribute
                - ItemsPerPage:
                    type: Integer
                    kind: attribute
                - Filter:
                    type: String
                    kind: attribute
    - RemoteWorkItemResponse:
        type: Entity
        id: lbs#remote_work_item_response
        facets:
          - main:
              type: hypergraph
              fields:
                - Success:
                    type: Boolean
                    kind: attribute
                - CurrentPage:
                    type: Integer
                    kind: attribute
                - TotalPages:
                    type: Integer
                    kind: attribute
                - TotalItems:
                    type: Integer
                    kind: attribute
                - Items:
                    type: lgc#list
                    kind: slot
    - RemoteWorkItemService:
        type: Service
        id: lbs#remote_work_item_service
        facets:
          - main:
              type: hypergraph
              fields:
                - Request:
                    type: lbs#remote_work_item_request
                    kind: slot
                - Response:
                    type: lbs#remote_work_item_response
                    kind: slot
```

Listing 62.2 – Adaptateur d'API paginée dans un modèle YAML

Ici, `kind` ne s'emploie que sur les champs du modèle déclarés à l'intérieur des facettes. Il classe les rôles conceptuels. Il ne s'applique ni aux jeux de données, ni aux sources de données, ni aux curseurs.

62.4 De l'adaptateur d'API au fournisseur de collection

L'adaptateur devient particulièrement intéressant lorsqu'il est relié à un fournisseur de collection. Celui-ci masque à la vue les détails de pagination, de reprise, d'authentification, de rafraîchissement et de synchronisation.

```
UXfc table or lookup component
  -> DataSource
  -> DataSet
  -> RemoteWorkItemProvider
  -> RemoteWorkItemService
  -> RemoteWorkItemRequest / RemoteWorkItemResponse
  -> remote API
```

Listing 62.3 – API distante derrière un fournisseur de collection

Le composant visuel ne change pas. Le fournisseur décide si les données proviennent du cache local, s'il faut les rafraîchir depuis l'API distante, les paginer à la demande ou les synchroniser en arrière-plan.

62.5 Erreurs, statuts et traçabilité

Un concept de réponse doit généralement exposer davantage que les seules données retournées : il doit aussi indiquer la réussite, le statut technique, les messages d'erreur métier, les horodatages, les identifiants de trace et, lorsqu'elles existent, les informations de pagination.

Ces informations alimentent le diagnostic, les politiques de nouvelle tentative, les tableaux de bord, les processus de validation humaine et les assistants d'IA.

62.6 Règles de conception

Modélisez les API externes comme des adaptateurs explicites. Séparez les concepts de requête, de réponse et de service. Conservez les métadonnées de pagination visibles. Reliez les API distantes par des fournisseurs et non directement par des composants visuels. Lorsque le traitement exige une intervention humaine ou asynchrone, faites remonter les échecs sous forme d'événements, de messages ou d'états de processus.

Chapitre 63

Intégration avec les systèmes externes

Les applications logiCells fonctionnent rarement de manière isolée. Elles accèdent à des bases de données, échangent des messages, appellent des services, reçoivent des événements externes, synchronisent des états locaux et distants et peuvent déléguer des traitements à des agents. L'intégration fait donc partie de la conception de la plateforme, mais elle ne doit pas compromettre la structure conceptuelle du modèle.

La règle centrale est simple : les systèmes externes doivent se raccorder à des capacités conceptuelles explicites, et non à des détails d'implémentation contingents.

63.1 L'intégration comme gestion des frontières

Un modèle logiCells contient des concepts, des objets, des relations, des actions, des événements, des vues et des processus. Un système externe doit être connecté à l'une de ces frontières explicites. Le couplage direct d'un code externe aux tables internes ou aux composants visuels fragilise l'intégration.

Les principales frontières sont les suivantes :

Facettes de base de données Assurent la persistance des champs conceptuels et leur projection dans un stockage relationnel ou structuré.

Actions publiées Exposent des capacités métier au moyen de services ou d'adaptateurs.

Événements Signalent les changements d'état et coordonnent des participants faiblement couplés.

Messages Franchissent de manière asynchrone les frontières entre processus, appareils ou réseaux.

Agents Exécutent des tâches, observent des changements ou coordonnent des traitements distants.

63.2 Intégration synchrone

L'intégration synchrone convient lorsqu'un appelant a besoin d'un résultat immédiat. Le cas le plus courant est l'appel externe d'une action publiée et exposée par un adaptateur de service. L'action s'exécute dans le runtime conceptuel et renvoie un résultat typé.

Cette approche préserve le modèle comme source de vérité. Le système externe connaît un contrat de service, tandis que le runtime exécute une capacité conceptuelle.

63.3 Intégration asynchrone

L'intégration asynchrone convient lorsque la latence, une dépendance externe, une intervention humaine ou une coordination en plusieurs étapes rendent une réponse immédiate inadaptée. L'appel externe doit alors créer ou mettre à jour un objet de processus, envoyer un message ou émettre un événement.

Cette approche est particulièrement utile pour :

— la messagerie distante ;
— la synchronisation hors ligne ;
— les processus de longue durée ;
— les interactions avec les agents ;
— les validations humaines ;
— les mises à jour d'interface pilotées par des événements.

La couche de messagerie de la plateforme appartient à logiCells/logiCells. Les processus métier peuvent spécialiser son utilisation, mais la déclaration des messages, leur distribution, les conversations et la coordination asynchrone constituent des mécanismes réutilisables.

63.4 Intégration aux bases de données

Une facette de base de données est une projection de champs conceptuels : elle ne doit pas être confondue avec le concept lui-même. Un même champ peut apparaître dans la facette hypergraphique sous la forme d'un rôle conceptuel, et dans la facette de base de données sous celle d'une ou de plusieurs colonnes persistées. Les objets liés sont généralement aplatis au moyen de chemins de rôle tels que `Owner.Id`, `Owner.ClassId` ou `Owner.FormalName`.

La persistance devient ainsi explicite, tandis que le modèle conceptuel reste indépendant et se situe au-dessus du schéma de base de données.

63.5 Intégration des messages et des événements

Les événements expriment les changements observables dans le runtime. Les messages permettent de transmettre ces changements au-delà des frontières techniques. Un message peut être persisté, routé, délivré plus tard, associé à une conversation ou rattaché à un état de processus.

Une bonne architecture d'intégration distingue ces niveaux :

```
Evenement conceptuel = fait survenu dans le modele
Message              = representation transportable d'un changement
                       ou d'une demande
Conversation         = groupe de messages et de participants lies
Processus            = objet de coordination reagissant aux
                       evenements et aux messages
Adaptateur de service = point d'entree externe dans le modele
```

Listing 63.1 – Niveaux d'intégration : correspondances conceptuelles

63.6 Agents et exécution externe

Les agents peuvent observer le modèle, exécuter des actions, consommer des messages, publier des événements ou appeler des services externes. Ils ne doivent pas contourner la sémantique conceptuelle. Un agent qui modifie le système doit utiliser les actions, la création d'objets, la persistance de données ou l'émission d'événements prévues par le runtime.

63.7 Règles de conception

Privilégiez les actions publiées pour les intégrations synchrones, les événements et les messages pour les échanges asynchrones, et les objets de processus lorsque plusieurs objets métier interviennent. Utilisez les facettes de base de données pour persister les données, et non comme API sémantique principale. Faites appel aux agents lorsque l'intégration nécessite de l'observation, de l'automatisation ou une exécution différée.

Chapitre 64

Modèles déclaratifs et ABI du runtime

L'ABI du runtime constitue la frontière d'intégration typée entre logiCells et le code hôte externe. Elle se situe volontairement sous le modèle conceptuel : concepts, relations, actions et permissions définissent le contrat métier ; l'ABI détermine comment le code externe invoque les capacités du runtime en toute sécurité et comment les valeurs et les durées de vie des objets franchissent cette frontière.

64.1 Disponibilité actuelle

La distribution actuelle de logiCells fournit des bindings ABI pour **Python** et **C#**. Ils exposent le même contrat ABI tout en transposant les règles de durée de vie et de représentation des valeurs dans les conventions de chaque langage.

Il ne faut pas présumer de la disponibilité de bindings pour d'autres langages sans déclaration explicite de leur prise en charge dans la distribution utilisée. Le présent guide ne revendique notamment aucune prise en charge de Java/JNI/JNA.

64.2 Identité stable des invocations

Un identifiant d'invocation comprend un identifiant de module, de type, de méthode et un genre d'appel. Celui-ci distingue les constructeurs, les appels statiques, les appels d'instance et les autres formes d'invocation représentées par l'ABI. Les manifestes ABI du code source actuel utilisent la version 2 du schéma, la version 1 de l'ABI et un contrat d'identification explicite.

64.3 Modèle de valeurs

Le modèle de valeurs de l'ABI couvre les valeurs nulles, booléennes, entières et flottantes, les chaînes, handles, tableaux, enregistrements, pointeurs, callbacks, valeurs numériques de largeur fixe et tampons d'octets. Un descripteur de valeur fournit aussi les informations nécessaires à sa bonne conversion : identité du type, taille, nullabilité, propriété et politique de libération.

Un descripteur de méthode regroupe :

— l'identité de l'invocation ;
— la signature qualifiée et sa forme canonique ;
— l'exigence éventuelle d'un objet récepteur ;
— la taille et l'alignement de l'enregistrement d'arguments ;
— les descripteurs de paramètres ordonnés ;
— le descripteur de résultat.

64.4 La propriété fait partie du contrat ABI

La propriété d'une ressource est une métadonnée explicite. Elle ne se déduit ni du nom d'une méthode, ni de son type de retour, ni d'une convention de nommage.

Pour les utilisateurs habituels des bindings Python et C#, deux cas reviennent particulièrement souvent :

`borrowed` L'objet renvoyé reste la propriété du runtime ou d'un autre objet du runtime. L'appelant peut utiliser cette référence dans les limites de durée de vie documentées, mais il ne doit pas libérer l'objet natif comme s'il en était propriétaire.

`owned_by_caller` La propriété est transférée à l'appelant. Le binding du langage doit libérer l'objet natif sous-jacent lors de la fermeture ou de la destruction contrôlée de son enveloppe.

Cette distinction prévient deux erreurs symétriques : une référence empruntée ne doit pas être libérée par l'appelant, tandis qu'un résultat dont il est propriétaire ne doit pas être traité comme une simple référence empruntée et provoquer une fuite.

> La propriété est définie par les métadonnées ABI. Ne la déduisez pas de noms tels que `Build`, `Create`, `Clone` ou `Snapshot`.

64.5 Vocabulaire complet de la propriété

Les auteurs de bindings et d'extensions du runtime doivent connaître tout le vocabulaire de l'ABI, et non uniquement les deux cas courants exposés par les enveloppes. Le contrat actuel définit six valeurs de propriété :

Propriété ABI	Signification à la frontière
`value`	La valeur ABI est transmise par valeur ; aucun transfert de durée de vie d'objet n'en découle.
`caller`	Le stockage ou la responsabilité de durée de vie est associé à un état fourni par l'appelant, selon le descripteur et l'agencement de l'invocation.
`owned_by_caller`	La ressource ou le résultat est transféré à l'appelant, qui doit le libérer selon la politique prévue.
`owned_by_module`	Le module conserve la propriété ; l'appelant ne doit pas détruire cette ressource comme un état lui appartenant.
`borrowed`	L'appelant reçoit une référence non propriétaire, dont la validité dépend de la durée de vie du propriétaire documenté par l'API.
`retained`	La frontière enregistre une relation de conservation de durée de vie ; le binding et le runtime doivent respecter les opérations correspondantes de conservation/libération au lieu de traiter la référence comme un simple emprunt.

Les noms publics des énumérations dans le code constituent des identifiants d'implémentation. Les manifestes et les bindings générés exposent la sémantique stable correspondante. Un générateur de bindings doit préserver ces distinctions, même si le langage hôte les présente au moyen d'une API d'enveloppe simplifiée.

64.6 Politiques de libération

Le contrat ABI actuel définit quatre modes de libération :

`none` Aucune opération de libération n'est associée à la valeur ;

`moduleRelease` La libération s'effectue par la fonction de libération du module ;

moduleFree La libération de la valeur ou de sa mémoire s'effectue par la fonction correspondante du module, suivant le contrat applicable à sa représentation ;

method La libération s'effectue par un contrat explicite au niveau d'une méthode.

Le type de propriété et le mode de libération sont deux champs distincts : un binding doit lire les deux. Il ne faut pas présumer que tous les résultats appartenant à l'appelant utilisent la même opération de libération.

64.7 Binding Python

Le binding Python préserve les règles de propriété de l'ABI dans ses enveloppes. Un résultat natif transféré à l'appelant est enveloppé dans une référence propriétaire ; un résultat emprunté dans une référence non propriétaire.

Pour une enveloppe propriétaire, l'appel de `close()` libère l'objet natif par le chemin de libération choisi dans le descripteur. Les appels répétés à `close()` doivent rester sûrs et ne jamais provoquer de double libération.

La validation récente vérifie spécifiquement que les résultats `owned_by_caller` produisent des enveloppes propriétaires, que les résultats `borrowed` demeurent non propriétaires et que la fermeture d'une enveloppe propriétaire atteint l'opération de libération du module exactement une fois.

64.8 Binding C#

Le binding C# suit le même contrat par l'intermédiaire de la propriété des handles. Les enveloppes de résultats transférés à l'appelant possèdent leur handle natif ; celles des résultats empruntés ne le possèdent pas.

Dans une enveloppe propriétaire, `Dispose()` libère l'objet natif sous-jacent. Plusieurs appels à `Dispose()` doivent rester sûrs et ne pas libérer deux fois le même objet natif.

La campagne de validation couvre la propagation de la propriété durant la génération des manifestes et des enveloppes C#, notamment la propriété correcte des résultats assimilables à des instances de classe et la préservation des retours réellement empruntés.

64.9 Paramètres et réécriture des valeurs

Chaque descripteur de paramètre indique son nom, son type source, son descripteur de valeur ABI, son décalage en octets et sa direction (`in`, `out` ou `in/out`). Le contrat d'exécution prend en charge la réécriture des valeurs `out` et `in/out` mises à jour.

Les validations actuelles comprennent les écritures en retour des scalaires, chaînes et tableaux, les cas de handles d'interface, la réaffectation d'interfaces et les chemins de propriété des résultats objets.

64.10 Placement natif et RPC

Le système de types ABI distingue les modes d'accès natif et RPC. Le placement relève de l'exécution : déplacer une capacité au-delà d'une frontière de processus ne doit pas imposer une seconde définition de son sens métier conceptuel.

Il ne faut pas déduire de cette abstraction la couverture effective des plateformes hôtes. Disponibilité du runtime et certification du chargeur ABI sont deux questions distinctes.

64.11 Manifestes ABI et informations de cible

Les manifestes `*.abi.json` générés décrivent les modules, types, méthodes, paramètres, résultats et caractéristiques de la cible. Ces dernières comprennent notamment la plateforme et l'architecture, la taille des pointeurs, l'endianness et les règles d'empaquetage des enregistrements.

Ces propriétés ont une portée opérationnelle. Deux artefacts ABI contenant les mêmes noms logiques de méthodes ne sont pas nécessairement compatibles au niveau binaire si les tailles de pointeurs, les agencements d'enregistrements ou les conventions d'empaquetage diffèrent.

64.12 Portée des plateformes pour le chargement ABI natif

Le runtime logiCells est disponible sous Windows, Linux et macOS. Toutefois, dans la version des sources étudiée, le chemin de chargement des modules natifs utilise l'API Windows des bibliothèques dynamiques. Ce guide *n'assimile donc pas* la disponibilité multiplateforme du runtime à une certification du chargement de modules ABI natifs sur toutes ces plateformes.

Python et C# sont les bindings actuellement disponibles. Pour déployer un binding donné sous Linux ou macOS, il faut vérifier le chargeur et le package de runtime effectivement fournis pour cette plateforme, sans présumer d'une parité à partir de la disponibilité globale du runtime.

Cette distinction est volontaire et doit rester visible tant que le chargement ABI propre à chaque plateforme n'a pas été couvert par les mêmes critères de validation exécutable.

64.13 Comportements ABI validés

Les validations ABI actuelles portent sur les fonctionnalités nécessaires aux bindings Python et C# dans les périmètres Interfaces, Common et Kernel inclus dans la version de référence du framework. Elles comprennent :

- l'initialisation et l'arrêt des modules ;
- les handles d'objets, l'identité des types et la propriété ;
- les valeurs scalaires et les chaînes UTF-8 ;
- les tableaux primitifs, imbriqués et d'interfaces ;
- les enregistrements gérés, non gérés et imbriqués ;
- les valeurs `out` et `in/out` de type scalaire, chaîne ou tableau ;
- les handles et la réaffectation d'interfaces ;
- les résultats objets empruntés ou appartenant à l'appelant ;
- la libération des ressources et l'idempotence de `close()`/`Dispose()` pour les résultats propriétaires.

Cette liste décrit des comportements ABI validés, et non une promesse de prise en charge automatique de toute construction imaginable dans un langage hôte. Lorsqu'une nouvelle surface ABI est ajoutée, il convient de vérifier le contrat de manifeste et d'exécuter les tests correspondants des bindings et du runtime.

64.14 Règle pour les développeurs

Modélisez d'abord le contrat métier, exposez ensuite une capacité stable et réservez les descripteurs ABI à la frontière opérationnelle. Les enveloppes Python et C# constituent des projections d'intégration du contrat du runtime ; elles ne doivent pas devenir le vocabulaire du modèle conceptuel.

Chapitre 65

Règles de conception de la publication

La publication consiste à rendre une capacité du modèle accessible au-delà de son contexte interne immédiat. Une capacité peut être publiée vers une interface utilisateur, un service web, un agent, un processus ou un canal de messagerie.

65.1 Publier des capacités, non des détails internes

Une action publiée doit correspondre à une capacité significative : valider un justificatif, créer un message, démarrer un processus, finaliser une tâche ou récupérer une projection autorisée. Évitez de publier les actions utilitaires de bas niveau qui exposeraient des détails d'implémentation.

65.2 Utiliser des signatures typées

La liste des paramètres définit le contrat du service. Choisissez des noms explicites, des types déclarés et, si possible, un seul rôle de retour. Préférez les types conceptuels lorsqu'un paramètre représente un concept.

65.3 Séparer les canaux de publication

Une même action peut être accessible depuis une vue, un agent ou un service web, mais les politiques associées diffèrent. Un service web exige une authentification et un mappage des requêtes et des erreurs. Une vue impose des règles de visibilité et d'interaction. Un agent requiert une enveloppe d'exécution.

65.4 Versionner les contrats

Les actions publiées sont des contrats. Renommer des paramètres, changer les types de retour ou modifier les effets de bord doit être considéré comme une évolution versionnée. Lorsqu'un contrat doit changer, préférez créer une nouvelle action ou un nouveau profil de publication.

65.5 Synthèse

La publication doit exposer des capacités conceptuelles stables. Le modèle demeure la source du contrat ; les canaux sont des adaptateurs.

Neuvième partie

logiCells et intelligence artificielle

Chapitre 66

Mémoire conceptuelle pour les applications d'IA

Les applications d'IA ont besoin de mémoire, mais toutes les formes de mémoire ne se valent pas. Une archive textuelle peut être interrogée, sans pour autant exposer directement les objets, rôles, contraintes, relations, actions ou états de processus. logiCells fournit une mémoire conceptuelle où l'information est structurée en concepts exécutables.

66.1 Objectif

Ce chapitre explique pourquoi la mémoire conceptuelle constitue un fondement solide pour les applications d'IA. Il montre aussi comment les modèles de langage, agents et composants d'automatisation peuvent exploiter les modèles logiCells sans se substituer à la sémantique du runtime.

66.2 Au-delà de la recherche documentaire

Un système de recherche peut retrouver des textes pertinents. Une mémoire conceptuelle peut en outre préciser quel objet est concerné, quelles relations le relient aux autres, quelles actions sont autorisées, quel est l'état du processus et quels événements sont survenus.

Pour l'IA, cette différence est essentielle. Le modèle fournit un ancrage structuré. L'IA peut raisonner sur des concepts et proposer des opérations, tandis que leur exécution reste soumise aux actions publiées et à la validation du runtime.

66.3 Un contexte structuré pour l'IA

La mémoire conceptuelle peut fournir à une IA un contexte structuré comprenant :
— les définitions de concepts ;
— l'état des objets ;
— les relations et les chemins de rôles ;
— les états des processus ;
— les traces d'événements ;
— les actions publiées disponibles ;
— les projections propres aux vues ;
— le contexte de sécurité et de visibilité.

66.4 Les actions comme points d'exécution contrôlés

Un assistant d'IA ne doit pas modifier directement le stockage. Il doit demander ou invoquer des capacités exposées par le modèle. Les actions publiées constituent des points d'exécution contrôlés : elles définissent ce qui peut être accompli, avec quels paramètres et sous quelles règles de validation.

66.5 Mémoire conceptuelle et explications

Comme le modèle explicite les concepts, actions et événements, les explications générées par l'IA peuvent s'appuyer sur des éléments identifiables. Plutôt que d'affirmer que « le système a décidé », l'assistant peut désigner l'état du processus, la règle, l'événement ou l'action à l'origine du résultat.

66.6 Règles de conception

Exposez à l'IA un contexte structuré plutôt qu'un accès illimité aux données internes. Laissez-la proposer des actions, mais confiez l'exécution au runtime. Utilisez les états des processus pour les décisions de longue durée. Appuyez-vous sur les journaux et les événements pour rendre auditables les actions assistées par l'IA.

Chapitre 67

Positionnement de logiCells dans les architectures neuro-symboliques

L'expression *neuro-symbolique* recouvre plusieurs architectures associant des représentations numériques apprises à des structures symboliques explicites. Elles ne résolvent pas toutes les mêmes problèmes et ne donnent pas la même autorité à leurs composants neuronaux et symboliques. Une plateforme doit donc préciser son positionnement, plutôt que d'employer le terme comme un simple synonyme de « IA avec règles ».

logiCells est conçu autour d'un hypergraphe conceptuel exécutable. Types, rôles, relations polyadiques, règles, observations, événements, provenance et traces d'exécution y coexistent. Le calcul neuronal intervient comme second mode de représentation et de recherche au-dessus de cette même mémoire conceptuelle. logiCells occupe ainsi deux positions neuro-symboliques complémentaires :

1. **l'intégration neuro-symbolique compositionnelle gouvernée**, à la frontière applicative, où les structures symboliques produisent les entrées neuronales et gouvernent la matérialisation de leurs résultats ;
2. **le raisonnement Symbolic[Neuro]**, au sein du moteur de déduction, où un raisonneur formel consulte un service d'intuition neuronale pour ordonner sa recherche sans lui déléguer la validation.

La notation `Symbolic[Neuro]` reprend le vocabulaire architectural popularisé par Henry Kautz dans sa taxonomie neuro-symbolique. Celle-ci sert ici de carte de conception, et non d'échelle de maturité. Un couplage mathématique plus étroit n'est pas automatiquement préférable à une composition qui préserve l'auditabilité, l'interchangeabilité des composants et les garanties formelles.

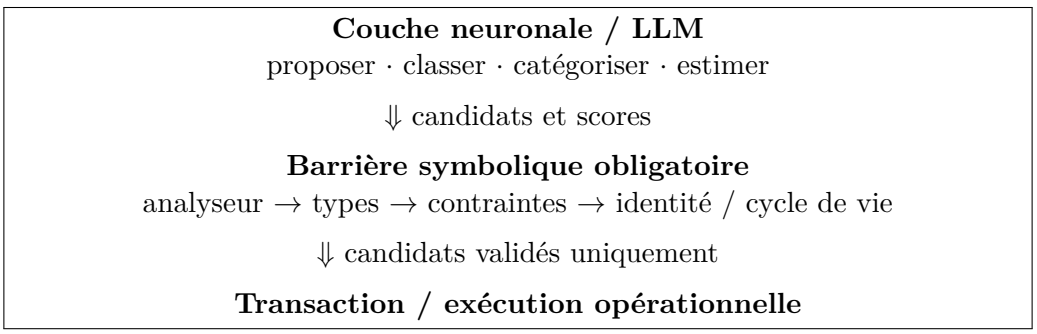


FIGURE 67.1 – Composition neuro-symbolique gouvernée : les composants neuronaux influencent les préférences, mais la validation symbolique conserve l'autorité sur la validité.

67.1 La taxonomie de Kautz comme carte architecturale

La taxonomie de Kautz distingue plusieurs modes récurrents d'association des systèmes symboliques et neuronaux. Ce chapitre suit le vocabulaire architectural présenté dans la conférence commémorative Robert S. Engelmore de l'AAAI et sa version publiée ; il reprend également le vocabulaire compositionnel de van Harmelen et ten Teije lorsque celui-ci décrit plus précisément les frontières entre modules. Les principales questions architecturales sont :

— Quel composant reçoit l'entrée initiale ?
— Lequel réalise le raisonnement ou la recherche ?
— Lequel produit le résultat porteur d'un sens externe ?
— Les connaissances symboliques servent-elles à produire des données d'apprentissage ?
— Le composant neuronal guide-t-il seulement une procédure symbolique ou participe-t-il à la sémantique de la conclusion ?
— Les structures symboliques sont-elles compilées dans une architecture neuronale, ou les composants restent-ils distincts à l'exécution ?

Cette taxonomie évite le faux dilemme entre moteur purement symbolique et architecture totalement différentiable. Plusieurs intégrations valides existent entre ces extrêmes.

67.1.1 Pipelines symbolique–neuronal–symbolique

Une entrée symbolique peut être encodée numériquement, traitée par un modèle neuronal, puis décodée en sortie symbolique. Beaucoup de pipelines de langage et d'embeddings suivent ce schéma. La présence de symboles aux extrémités ne signifie pas que le calcul interne réalise un raisonnement symbolique explicite.

logiCells prend en charge ce modèle lorsqu'une projection conceptuelle est transformée en tenseurs denses et qu'un modèle retourne des catégories, des scores ou des embeddings. Ce schéma, pris isolément, n'est cependant pas la principale revendication architecturale de la plateforme.

67.1.2 Perception neuronale suivie d'un raisonnement symbolique

Un composant neuronal peut transformer des données non structurées ou perceptuelles en symboles destinés au raisonnement formel. Un classificateur de documents, un extracteur d'entités ou un modèle de vision peut proposer des observations et relations typées. H-Logic et les transformations hypergraphiques peuvent ensuite les valider, les combiner et les propager.

logiCells peut appliquer ce schéma aux modèles externes, aux modèles de langage locaux et à l'extraction structurée. La frontière fondamentale est la suivante : un résultat neuronal entre dans l'hypergraphe comme observation qualifiée ou proposition, jamais comme fait incontestable.

67.1.3 Systèmes symboliques produisant des données d'apprentissage neuronal

Un moteur symbolique peut générer des exemples, des étiquettes, des contraintes ou des cas synthétiques employés pour entraîner un modèle neuronal. Les modèles conceptuels et les règles H-Logic de logiCells pourront prendre en charge ce parcours en produisant des jeux d'apprentissage et des traces de déduction.

Cette direction importe tout particulièrement pour le guidage de la déduction. Les traces de preuves réussies ou échouées peuvent constituer des exemples auto-supervisés permettant d'apprendre quelles prémisses, règles ou branches sont utiles.

67.1.4 Symbolic[Neuro]

Dans le modèle `Symbolic[Neuro]`, la procédure symbolique demeure l'algorithme principal et appelle un composant neuronal comme sous-routine. AlphaGo en fournit une illustration connue : les règles du jeu et la recherche arborescente restent explicites, tandis que les évaluations apprises orientent les positions à examiner en priorité.

C'est la cible principale de l'intuition neuronale dans logiCells :

```
objectif formel et ensemble complet de candidats
    -> evaluation ou classement neuronal
    -> recherche symbolique reordonnee
    -> preuve H-Logic ou echec formel
```

Listing 67.1 – Raisonneur symbolique assisté par une intuition neuronale

Le modèle neuronal influence *où le raisonneur regarde en premier*, mais pas *ce qu'il a le droit de conclure.*

67.2 Première position de logiCells : la composition gouvernée

La première frontière neuro-symbolique implémentée est compositionnelle et gouvernée. Le framework possède également un cycle de vie exécutable des candidats H-Logic : chaque candidat porte des informations d'origine et de provenance, est validé indépendamment de son activation dans le graphe et ne devient connaissance opérationnelle que par un chemin contrôlé de transaction et de commit. Analyser syntaxiquement une proposition n'équivaut donc pas à activer une connaissance.

```
hypergraphe conceptuel
    -> projection tensorielle conceptuelle ou dense
    -> calcul neuronal ou tensoriel
    -> observation typee ou proposition
    -> H-Logic, transformations et gouvernance
    -> materialisation acceptee, rejetee ou en attente
```

Listing 67.2 – Cycle de composition gouverné

Les détails d'implémentation des programmes tensoriels figurent dans un chapitre de sources avancé du dépôt et non dans le parcours de lecture métier. Le contrat architectural nécessaire ici se résume aux propriétés suivantes :

Origine symbolique de l'identité Lignes et axes des tenseurs ainsi que positions topologiques restent traçables jusqu'aux concepts et aux rôles.

Dérivation numérique explicite Le tenseur est une projection d'un état conceptuel sélectionné ; il ne remplace pas cet état.

Sortie qualifiée Un score ou une prédiction est représenté comme observation, hypothèse ou relation proposée.

Matérialisation gouvernée Règles, transformations, politiques et validations humaines décident si la proposition affecte la connaissance opérationnelle.

Exécution interchangeable Le runtime neuronal peut évoluer sans modifier le contrat conceptuel de ses frontières.

Cette intégration est volontairement souple pour l'optimisation et exigeante pour la sémantique et la gouvernance. Elle convient particulièrement aux systèmes métier de longue durée, où traçabilité, remplacement des modèles et supervision humaine sont indispensables.

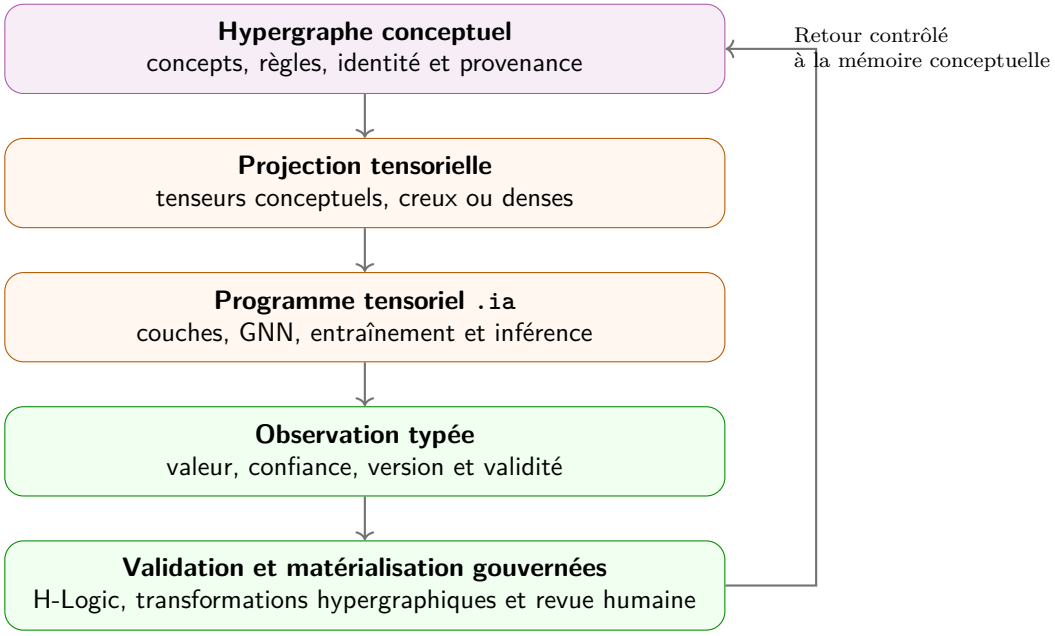


FIGURE 67.2 – Intégration compositionnelle neuro-symbolique gouvernée. Le programme tensoriel produit une observation typée ; H-Logic, les transformations hypergraphiques et la revue humaine déterminent si elle rejoint la mémoire conceptuelle, et de quelle manière.

67.3 Deuxième position : déduction Symbolic[Neuro]

La deuxième position fait entrer le calcul neuronal dans la boucle de raisonnement formel. H-Logic conserve l'autorité sur la dérivation, mais peut consulter un modèle latent pour choisir ce qu'il doit explorer.

Parmi les services de guidage possibles figurent :

— la sélection des prémisses ;
— la sélection des règles ;
— le classement des branches ;
— la planification des jointures et parcours ;
— la génération de conjectures ;
— la proposition de relations ;
— la priorisation des contradictions ou anomalies.

La solution sûre par défaut est une *heuristique de réordonnancement*. Le raisonneur construit d'abord le même ensemble complet de candidats qu'en l'absence de modèle neuronal. Les scores modifient ensuite l'ordre, et non l'appartenance à cet ensemble.

```
Candidates := BuildCompleteCandidateSet(Goal)
Scores     := Intuition.Rank(Goal, Candidates, Context)
Ordered    := ReorderWithoutRemoving(Candidates, Scores)
Result     := ExploreFormally(Ordered)
```

Listing 67.3 – Guidage préservant la complétude

La garantie critique concerne la validité, pas l'achèvement assuré dans un budget opérationnel fini :

> Une intuition neuronale ne doit pas modifier ce qui constitue une preuve valide. Avec un délai maximal, une profondeur ou un nombre de nœuds limités, elle peut cependant modifier les preuves valides atteintes avant épuisement du budget.

Cette distinction a une importance opérationnelle. En mode sûr, un candidat mal noté est reporté plutôt qu'éliminé ; un ordre défavorable peut néanmoins faire expirer une preuve

auparavant atteinte dans le budget imparti. L'épuisement des ressources doit donc être rapporté explicitement, jamais interprété comme une fausseté formelle. La garantie sûre est que chaque preuve produite reste valide et que les conclusions formelles des deux recherches complètes sont équivalentes lorsque toutes deux se terminent.

Cette garantie suppose également que l'opération réordonnée soit sémantiquement insensible à l'ordre. Avant d'activer le guidage, il faut identifier les fonctions H-Logic comme la négation par échec, les opérateurs de type *cut*, la sémantique du premier résultat ou les limites portant sur les N premiers résultats. Lorsque le sens observable dépend de l'ordre, le guidage doit être désactivé, limité à une région sémantiquement neutre ou déclaré comme politique opérationnelle approximative.

L'élagage approximatif pourra être pris en charge ultérieurement, mais devra être explicitement déclaré et ses résultats marqués comme issus d'une recherche approximative.

67.4 Pourquoi l'hypergraphe transforme-t-il le problème d'intégration ?

Beaucoup de systèmes neuro-symboliques doivent relier deux univers incompatibles : un prouveur manipule des termes et prédicats, tandis qu'un modèle consomme des vecteurs anonymes. logiCells part d'une situation différente : le raisonnement formel et la projection latente sont dérivés du même hypergraphe conceptuel.

Un tenseur conceptuel peut préserver :

— l'identité stable de chaque concept ;
— le type formel de chaque relation ;
— les rôles ordonnés d'une relation polyadique ;
— l'analyse associée à chaque axe tensoriel ;
— la version de la topologie et de la correspondance des indices ;
— la provenance des valeurs projetées.

Le pont n'est donc pas une simple sérialisation. Il est un mapping explicite entre identités symboliques et positions numériques. Ce mapping fonde à la fois la matérialisation gouvernée et le guidage neuronal de la recherche formelle.

67.5 Des hypergraphes, pas seulement des graphes binaires

Un réseau neuronal de graphes classique suppose souvent des arêtes binaires. Une relation logiCells peut être polyadique et sensible aux rôles. La transformer en clique ou en étoile peut effacer son arité, ses noms de rôles et son identité propre.

Considérons une relation de transaction avec les rôles acheteur, vendeur, produit, canal et date. Une représentation conceptuelle tenant compte des rôles doit préserver la structure d'incidence :

```
participant
    <-> role du participant
    <-> instance d'hyperrelation
    <-> role du participant
    <-> autre participant
```

Listing 67.4 – Incidence hypergraphique tenant compte des rôles

Un futur tenseur conceptuel peut donc comporter au moins trois axes :

```yaml
inputs:
  - name: transactionIncidence
    kind: conceptual
    formalType: Transaction

```

```yaml
    axes:
      - name: relation
        analysis: relationIdentity

      - name: role
        analysis: participantRole

      - name: participant
        analysis: participantIdentity
```

Listing 67.5 – Projection conceptuelle de rang trois tenant compte des rôles

Il s'agit d'un domaine naturel pour les opérations natives logiCells. Les runtimes génériques de calcul dense savent exécuter l'algèbre linéaire après projection, mais ignorent la sémantique des hyperarités, des rôles formels et de l'identité conceptuelle incrémentale.

67.6 Engagements de conception

Les engagements suivants précisent le positionnement recherché par la plateforme.

67.6.1 Ne pas confondre performance neuronale et correction formelle

En mode de guidage sûr, la correction et la performance sont évaluées séparément. Pour des recherches non bornées qui se terminent toutes deux, les conclusions formelles doivent être équivalentes. Sous des ressources bornées, le résultat peut être *ressources épuisées* ; ce statut n'est ni une preuve ni une réfutation.

```
si les deux recherches completes se terminent :
    conclusions guidees = conclusions non guidees

sous un budget fini :
    toute preuve produite demeure formellement valide
    l'epuisement des ressources reste explicite

sur une charge de reference :
    cout guide agrege < cout non guide agrege
```

Listing 67.6 – Correction et performance évaluées séparément

La performance est une propriété agrégée de benchmark, pas une affirmation sémantique sur chaque preuve. Certains objectifs sont déjà triviaux, d'autres ne tirent aucun bénéfice du guidage, et certains peuvent même ralentir. Un backend ou un modèle n'est accepté que s'il respecte le contrat sémantique et améliore une charge de référence déclarée, sans masquer les régressions par des statuts de délai expiré ou d'échec.

67.6.2 Toute intuition est versionnée

Un score n'a de sens qu'en relation avec :
— la version de l'artefact de modèle ;
— un état ou instantané de l'hypergraphe ;
— une définition de projection ;
— un mapping d'indices ;
— une version de l'espace latent ;
— un instant d'évaluation.

Une intuition périmée ne doit jamais être utilisée silencieusement. Le runtime doit pouvoir attendre une vue actualisée, revenir au raisonnement purement symbolique ou autoriser explicitement une obsolescence bornée.

67.6.3 Les propositions ne sont pas des conclusions

Prédictions de liens, conjectures et suggestions de prémisses manquantes sont représentées comme des structures symboliques proposées, accompagnées de confiance et de provenance. Elles peuvent ensuite être prouvées, réfutées, conservées pour une revue humaine ou laissées expirer.

67.6.4 Un runtime natif aux responsabilités ciblées

logiCells doit conserver la responsabilité native des opérations dépendant de l'identité conceptuelle et de la sémantique hypergraphique :

— les correspondances d'indices stables et locales ;
— les projections conceptuelles creuses ;
— l'invalidation incrémentale après mutation de l'hypergraphe ;
— la propagation de messages hypergraphiques tenant compte des rôles ;
— la conversion des propositions en observations conceptuelles gouvernées.

Le calcul dense générique et l'inférence de grands modèles de langage peuvent emprunter plusieurs chemins de runtime. La documentation d'ingénierie décrit une frontière de service LLM indépendante du fournisseur et un chemin d'inférence résidente lisant des modèles GGUF, qui peut bénéficier d'une accélération comme MLX sur les déploiements Apple Silicon pris en charge. Le chemin résident comprend un état de modèle persistant et une infrastructure de génération jeton par jeton, avec cache KV. Les moteurs externes comme llama.cpp restent utiles en tant que références ou fournisseurs, mais ne sont pas l'unique modèle d'implémentation.

Le contrat public porte sur la capacité et la frontière de son cycle de vie, pas sur le type d'implémentation interne. L'application doit sélectionner le backend d'inférence pris en charge par son déploiement, sans présumer que tous les backends existent sur chaque hôte.

67.7 Périmètre architectural et objectifs explicitement exclus

Aujourd'hui, logiCells ne compile pas les règles H-Logic en poids neuronaux et n'intègre pas toute la sémantique formelle dans une architecture différentiable. Cette orientation plus étroite `Neuro[Symbolic]` constitue un autre choix de conception. Elle peut être intéressante pour l'apprentissage sous contraintes, mais lie davantage la validité, le comportement d'entraînement et le remplacement des modèles que l'architecture gouvernée retenue ici.

Le périmètre actuel est volontaire :

— les structures symboliques restent inspectables et exécutables indépendamment ;
— les composants neuronaux fournissent scores, représentations et propositions ;
— les obligations de preuve formelle restent du ressort de H-Logic et du runtime conceptuel ;
— les politiques approximatives sont explicites, et non cachées dans un optimiseur ;
— remplacer un modèle ne redéfinit pas le sens d'une preuve acceptée.

Ce choix favorise l'auditabilité, l'interchangeabilité opérationnelle et la supervision humaine. Des contraintes différentiables pourront être ajoutées ultérieurement comme autre modèle de composition, mais elles ne remplaceront pas silencieusement le contrat gouverné.

67.8 Terminologie et références principales

Ce guide emploie *intégration neuro-symbolique compositionnelle gouvernée* pour le modèle situé à la frontière applicative et *guidage de la déduction Symbolic[Neuro]* pour celui situé dans le raisonneur. Ces termes restent cohérents dans les chapitres suivants.

Le positionnement s'appuie sur :

— Henry A. Kautz, « The Third AI Summer : AAAI Robert S. Engelmore Memorial Lecture », *AI Magazine*, volume 43, numéro 1, pages 93–104, 2022, issu de la conférence Engelmore de l'AAAI 2020 ;
— Frank van Harmelen et Annette ten Teije, « A Boxology of Design Patterns for Hybrid Learning and Reasoning Systems », *Journal of Web Engineering*, 2019 ;
— Michael van Bekkum, Maaike de Boer, Frank van Harmelen, André Meyer-Vitali et Annette ten Teije, « Modular Design Patterns for Hybrid Learning and Reasoning Systems : A Taxonomy, Patterns and Use Cases », *Applied Intelligence*, volume 51, numéro 9, pages 6528–6546, 2021.

La taxonomie est une carte des compositions architecturales, et non une affirmation selon laquelle une catégorie serait universellement plus mature qu'une autre.

67.9 État actuel et évolution envisagée

L'implémentation actuelle présente deux niveaux de maturité distincts qu'il ne faut pas confondre. La *barrière symbolique autour des propositions d'IA* est implémentée et couverte par des tests exécutables. L'architecture de guidage interne de recherche `Symbolic[Neuro]` est encore développée par étapes.

Les statuts doivent rester explicites dans tout le guide :

Implémenté et testé – barrière des propositions gouvernées Les candidats H-Logic portent origine et provenance ; la validation syntaxique est indépendante de l'activation ; les candidats rejetés ne peuvent pas être committés ; les candidats acceptés suivent un cycle explicite transaction/commit. Les tests actuels comprennent des candidats issus de LLM.

Implémenté et testé – socle neuronal et tensoriel Chargement déclaratif `.ia`, entrées denses et conceptuelles, couches sélectionnées, entraînement supervisé, convolution de graphe et GraphSAGE sur topologie conceptuelle.

En cours de conception Traces de déduction, interfaces d'intuition, vues latentes versionnées et réordonnancement des candidats préservant la complétude.

Prévu Sélection des prémisses et règles, planification des requêtes, propagation de messages hypergraphiques sensible aux rôles, index latents incrémentaux et génération gouvernée de conjectures.

La distinction est fondamentale : « le LLM propose et le runtime symbolique décide » constitue déjà un contrat de gouvernance exécutable. Le guidage neuronal de la recherche interne de preuves est une capacité distincte, moins mature.

67.10 Synthèse du positionnement

logiCells ne soutient pas que toute intégration neuro-symbolique doive être différentiable. La plateforme privilégie délibérément deux formes complémentaires :

Composition gouvernée Le calcul neuronal produit des observations et des propositions dont les effets sémantiques restent contrôlés par le runtime conceptuel.

Symbolic[Neuro] Le raisonnement formel consulte l'intuition neuronale pour prioriser la recherche, tout en conservant l'autorité symbolique sur les conclusions.

La promesse essentielle se résume ainsi :

> Le composant neuronal peut choisir où le raisonneur cherche d'abord, mais jamais ce qui constitue une preuve valide.

Lorsque les ressources sont bornées, le guidage peut néanmoins déterminer si la preuve est atteinte à temps. L'épuisement des ressources reste donc un résultat explicite, jamais assimilé à un échec formel.

Chapitre 68

Agents opérant sur les modèles conceptuels

Dans logiCells, un agent doit agir sur le modèle conceptuel plutôt que sur des artefacts techniques sans lien entre eux. Il peut consulter les concepts, parcourir leurs relations, appeler des actions publiées, consommer des messages, produire des événements et se coordonner avec des objets de processus.

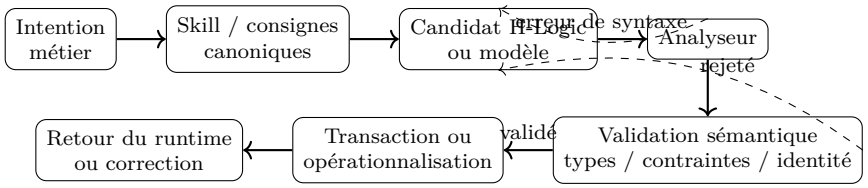


FIGURE 68.1 – La génération agentique suit une boucle de correction gouvernée : le Skill oriente la génération, mais l'analyseur et la validation sémantique décident de l'acceptation.

68.1 Objet du chapitre

Ce chapitre explique comment des agents accèdent à une application logiCells par ses capacités conceptuelles publiées. L'agent peut être mis en œuvre par un orchestrateur tiers et traverser une frontière de publication, par exemple MCP. Le protocole modifie le mode d'accès, mais ne redéfinit pas la capacité conceptuelle.

L'exécution des agents résidents est disponible par le runtime Agent et la CLI. Les principes conceptuels présentés ici sont complétés par le parcours de session exécutable décrit dans Chapitre 69, "Runtime Agent, objectifs et sessions persistantes".

68.2 Principes fondamentaux

Les agents utilisent des éléments explicites du modèle :

Concepts Définissent les objets et capacités accessibles à l'agent.

Actions Offrent les opérations appelables.

Actions publiées Fournissent des points d'entrée d'exécution stables.

Événements Permettent de réagir aux changements.

Messages Permettent la coordination asynchrone.

Processus Structurent les travaux de longue durée.

68.3 Frontière de publication et MCP

Un agent externe doit accéder à une projection maîtrisée de l'application, et non à une API technique autonome dupliquant la sémantique métier. Une couche de publication peut exposer des ressources conceptuelles sélectionnées et des actions publiées à un orchestrateur d'agents. MCP est l'un des protocoles possibles pour cette frontière.

La plateforme actuelle prend en charge MCP dans les deux sens. Le composant Tooling comprend un chemin de *client/opérationnalisation* MCP gouverné : découverte, listes explicites d'autorisations pour les opérations en lecture seule ou à effets, vérifications des politiques et des empreintes de schémas, et tests d'exécution des outils. La découverte ne constitue expressément pas une autorisation.

La CLI expose également la compilation MCP pour les applications et les packages. Une surface MCP compilée peut être déclarée dans la configuration Agents du fichier `.lgc` de l'application. Le runtime Agents démarre alors le serveur MCP configuré et rend la frontière publiée accessible aux clients MCP externes. La procédure développeur figure dans Chapitre 61, "Publier des applications avec OpenAPI et MCP".

La publication d'un serveur et la gouvernance des appels entrants sont deux garanties distinctes. Les sources examinées établissent l'existence de la surface de publication et d'hébergement ; elles ne démontrent pas, à elles seules, qu'un appel MCP entrant à effets traverse `PolicyContext`, `EvidenceRequirements` et `PendingConfirmation` avant son dispatch. L'autorité requise est définie dans Chapitre 70, "Contrat d'exécution gouvernée des agents" et ne doit être considérée comme vérifiée qu'après réussite des tests de régression du chemin serveur.

Le contrat conceptuel reste piloté par le modèle :

```
Modèle logiCells
    -> capacités conceptuelles publiées
    -> adaptateur de publication / frontière MCP
    -> agent externe
```

La couche de publication doit préserver les noms déclarés, les paramètres typés, le périmètre d'autorisation et les règles de validation. Elle ne doit pas inventer une ontologie parallèle pour les mêmes opérations métier.

68.4 Utiliser les contrats d'interaction VIEW sans afficher de formulaire

Un agent peut consommer le même contrat d'interaction VIEW qu'un client natif ou web sans matérialiser de formulaire graphique. Ce contrat décrit de façon structurée le contexte de données courant, les fonctions d'interaction, le comportement associé au MODEL, les actions disponibles et les relations avec les événements.

L'agent peut ainsi suivre le parcours suivant :

```
1 charger le contrat d'interaction
2 -> consulter les valeurs disponibles et courantes
3 -> determiner les valeurs encore necessaires a l'action visee
4 -> presenter a l'utilisateur l'etat utile
5 -> demander uniquement les informations manquantes
6 -> appeler l'action publiee ou l'action du modele declaree
7 -> observer le resultat ou l'evenement et poursuivre l'interaction
```

Listing 68.1 – Exécution agentique d'un contrat d'interaction

L'agent reste ainsi aligné sur la même sémantique d'interaction applicative que les interfaces destinées aux personnes. Le contrat n'ordonne pas de fabriquer un faux formulaire : il décrit

un plan d'interaction indépendant du canal. Les exigences d'autorisation, de politique et de confirmation proviennent toujours de la frontière d'exécution gouvernée présentée plus loin.

68.5 Invocation des actions publiées par un agent

Un agent peut appeler une action publiée de la même manière qu'elle peut être exposée par un service web. Seul le mécanisme d'accès diffère, et non la capacité conceptuelle. Une même action peut donc être réutilisée par une interface utilisateur, une API, un processus ou un agent d'automatisation.

68.6 Coordination entre agents et processus

Lorsqu'un agent participe à un enchaînement de plusieurs étapes, l'objet de processus doit rester l'autorité de coordination. L'agent peut exécuter une étape, valider des données, appeler un service externe ou émettre un événement de résultat ; l'état du processus conserve néanmoins la trace de l'avancement.

Un orchestrateur externe doit donc éviter de reconstruire un workflow applicatif déjà modélisé dans logiCells. Il peut déclencher ou faire avancer ce processus par ses capacités publiées, puis observer l'état ou les événements déclarés.

68.7 Agents et messagerie

Pour les travaux asynchrones, l'agent peut écouter des messages ou des événements et répondre ultérieurement. Ce fonctionnement convient aux traitements longs ou dépendants d'un système externe. La réponse peut prendre la forme d'un message, d'un événement ou d'une invocation d'action.

68.8 Automatisation avec intervention humaine

Lorsqu'une automatisation exige une décision humaine, la pause doit être représentée dans le processus conceptuel, et non dissimulée dans le code de l'agent. Le processus expose l'état en attente, la vue ou l'action publiée pertinente, puis l'événement qui permet la reprise. La décision reste ainsi observable et auditable.

68.9 Agents d'IA et barrière symbolique

Un agent d'IA peut examiner l'état conceptuel, proposer des actions, résumer des événements, classer des candidats ou suggérer des changements de modèle. Il reste une *source de propositions*, et non une autorité. Le H-Logic généré doit être considéré comme un candidat et passer par l'analyseur officiel et la validation sémantique avant de pouvoir intervenir dans une transaction ou une exécution opérationnelle.

Lorsqu'un Skill d'IA dispose d'un validateur exécutable, la boucle recommandée est *proposer* $\rightarrow$ *valider* $\rightarrow$ *corriger*. Il faut préférer un nombre limité de tentatives à une autocorrection sans borne : après quelques échecs, les derniers diagnostics et la question métier non résolue doivent être renvoyés au développeur.

68.10 Règles de conception

Les agents doivent utiliser les mêmes capacités publiées, états de processus, événements et messages que les autres participants au runtime. Il ne faut pas dupliquer une capacité métier

sous la forme d'une fonction propre à l'agent lorsqu'une action publiée existe déjà. MCP, ou tout autre protocole d'agent, constitue un adaptateur de publication et non un second modèle métier. La coordination de longue durée appartient aux modèles de processus. Les opérations critiques doivent rester auditables par les changements d'état, les événements et les journaux.

Chapitre 69

Runtime Agent, objectifs et sessions persistantes

La CLI `cell` expose directement le runtime Agent. Les développeurs peuvent exécuter une tâche en une seule commande ou travailler dans une session persistante au cycle de vie explicite : ouvrir la session, soumettre un objectif, consulter les confirmations en attente, confirmer ou rejeter une opération, examiner un instantané, évaluer la session et la fermer.

Ce chapitre décrit cette surface exécutable, sans dépendre d'une interface utilisateur ni d'un workflow commercial particulier.

69.1 Exécution ponctuelle d'un agent

Le point d'entrée direct est :

```
cell agent run <task> [--agent-id <id>] [--subject-id <id>]
               [--purpose <name>] [--profile <name>]
               [--knowledge-source <path>]
               [--knowledge-mode source|project]
               [--knowledge-answer deterministic|verbalized]
               [--knowledge-query <hlogic>]
               [--knowledge-nested-query <hlogic>]
               [--max-turns <n>]
               [--max-provider-iterations <n>]
               [--max-iterations <n>]
               [--interactive]
               [--on-confirmation fail|reject]
```

Choisissez cette forme lorsque la tâche peut être exécutée en une seule interaction CLI, sans gestion explicite d'une session persistante par l'appelant.

Les options d'identité et de contexte sont distinctes :

`-agent-id` Sélectionne l'identité de l'agent lorsqu'il existe plusieurs définitions.

`-subject-id` Fournit l'identité du sujet utilisée par le contexte d'exécution.

`-purpose` Transmet au contexte du runtime une finalité nommée.

`-profile` Sélectionne une configuration ou un profil d'exécution.

Les limites d'itérations permettent au développeur de borner les boucles du fournisseur et du runtime. Utilisez-les dans les tests et automatisations afin qu'un comportement inattendu du fournisseur ne déclenche pas une exécution sans limite.

69.2 Sources de connaissances et requêtes H-Logic

L'exécution d'un agent peut recevoir une source de connaissances explicite et l'interroger avec H-Logic :

```
--knowledge-source <path>
--knowledge-mode source|project
--knowledge-answer deterministic|verbalized
--knowledge-query <hlogic>
--knowledge-nested-query <hlogic>
```

Le mode `source` cible directement la source de connaissances fournie ; le mode `project` utilise une frontière de connaissances liée au projet. Le mode de réponse distingue les résultats déterministes des formulations en langage naturel. Les arguments de requête H-Logic rendent la demande de connaissances explicite au lieu de laisser au fournisseur toute la sélection du contexte par un prompt libre.

Pour le diagnostic, validez d'abord, si possible, les mêmes structures sémantiques et requêtes H-Logic avec les commandes sémantiques dédiées. Le comportement de l'agent est plus simple à interpréter si la validité de la requête sous-jacente est établie indépendamment du fournisseur.

69.3 Termes canoniques du runtime Agent

La documentation développeur emploie les noms de capacités suivants de manière cohérente entre la CLI, la spécification du runtime et la matrice de capacités :

Agent Session / Goal Session persistante et objectif soumis, exposés par `cell agent session open` et `cell agent session goal`.

KnowledgeQuery Capacité de requête de connaissances. Ses options CLI comprennent `-knowledge-query` et `-knowledge-nested-query`. Le nom de la capacité reste `KnowledgeQuery`.

PendingConfirmation État du runtime dans lequel une opération à effets proposée n'a pas encore été déclenchée et peut être examinée avec `cell agent session pending`.

Confirmation Broker Responsabilité du runtime consistant à résoudre une demande par confirmation ou rejet, puis à reprendre la session propriétaire. La CLI publique en expose le cycle de vie, même si l'implémentation réside derrière un service du runtime.

PolicyContext Contexte d'autorisation permettant de décider si une opération est autorisée, refusée ou soumise à confirmation. La syntaxe précise de configuration dépend du runtime et de sa version ; on ne peut la déduire de la CLI seule.

EvidenceRequirements Conditions de preuve éventuellement imposées par la politique avant d'autoriser une opération. Leur syntaxe de déclaration doit provenir du schéma faisant autorité avant publication d'un exemple prêt à copier.

Ces noms décrivent des capacités et non l'orthographe des commandes. Ils sont repris dans le contrat d'exécution gouvernée de Chapitre 70, “Contrat d'exécution gouvernée des agents”.

69.4 Sessions persistantes des agents

Les sessions persistantes exposent directement la machine à états du runtime :

```
cell agent session open [--agent-id <id>]
cell agent session goal <session-id> <goal> [options]
cell agent session pending <session-id>
cell agent session confirm <session-id> <request-id>
cell agent session reject <session-id> <request-id>
cell agent session snapshot <session-id>
cell agent session evaluate <session-id>
cell agent session close <session-id>
```

Les commandes de session nécessitent un `-tooling-endpoint` persistant. Un processus Tooling STDIO transitoire ne peut conserver l'état d'une session entre plusieurs commandes CLI indépendantes.

Voici une séquence de développement habituelle :

```
cell --tooling-endpoint <url> agent session open
# record the returned session id

cell --tooling-endpoint <url> agent session goal <session-id> \
  "<goal>" --max-iterations 8

cell --tooling-endpoint <url> agent session pending <session-id>
# inspect a request id when confirmation is required

cell --tooling-endpoint <url> agent session confirm <session-id> <request-id>
# or:
cell --tooling-endpoint <url> agent session reject <session-id> <request-id>

cell --tooling-endpoint <url> agent session snapshot <session-id>
cell --tooling-endpoint <url> agent session evaluate <session-id>
cell --tooling-endpoint <url> agent session close <session-id>
```

Utilisez `-json` lorsqu'un script nécessite des résultats stables et lisibles par une machine plutôt qu'une sortie destinée à l'utilisateur.

69.5 Goals : les objectifs

Un Goal est la tâche transmise à une session d'agent déjà ouverte :

```
cell agent session goal <session-id> <goal>
     [--agent-id <id>] [--subject-id <id>]
     [--purpose <name>] [--profile <name>]
     [--max-iterations <n>]
     [--knowledge-source <path>]
     [--knowledge-mode source|project]
```

Un objectif ne constitue pas une autorisation. Il donne une intention à l'agent dans le contexte du runtime. Les capacités disponibles, vérifications de politiques, contraintes de connaissances et exigences de confirmation continuent de gouverner ce qui peut réellement être exécuté.

69.6 Confirmations en attente

Les traitements à effets peuvent se suspendre en attente de confirmation au lieu de s'exécuter immédiatement. La CLI permet d'examiner cet état :

```
cell agent session pending <session-id>
```

L'identifiant de requête retourné est ensuite utilisé explicitement :

```
cell agent session confirm <session-id> <request-id>
cell agent session reject <session-id> <request-id>
```

Le principe essentiel est que la proposition et l'autorité restent distinctes. Un fournisseur peut proposer une opération ; le runtime peut la maintenir en attente ; un autre participant peut l'accepter ou la rejeter ; et la session reste le conteneur de l'exécution et de sa suite.

Pour une exécution ponctuelle, `-on-confirmation fail|reject` définit le comportement de la CLI lorsqu'une confirmation est exigée sans qu'un parcours interactif de confirmation soit maintenu.

69.7 Instantanés et évaluation

`snapshot` et `evaluate` sont deux surfaces développeur distinctes :

`snapshot` Examine l'état courant de la session pour le diagnostic, les tests ou l'orchestration.

`evaluate` Demande l'évaluation de la session par l'opération correspondante du runtime.

Ne remplacez pas l'inspection de la session par le récit du fournisseur. Pour diagnostiquer une exécution à plusieurs tours, appuyez-vous sur les surfaces de session explicites afin d'établir ce que le runtime a enregistré.

69.8 MCP dans l'exécution des agents

Le runtime Agent peut utiliser MCP dans les deux directions décrites dans Chapitre 61, "Publier des applications avec OpenAPI et MCP".

Un agent externe peut se connecter à une surface MCP publiée par logiCells. Inversement, un agent logiCells peut consommer les outils de serveurs MCP externes configurés. Sur le chemin client, la découverte n'emporte pas autorisation d'exécuter. Le classement lecture seule/opérations à effets, la politique et la confirmation peuvent rester des étapes indépendantes de l'opérationnalisation des outils.

Cette distinction est importante lorsqu'un outil apparaît dans la découverte mais reste inexécutable dans le contexte courant : le contrat de publication, la décision du catalogue ou de la politique et l'état de confirmation en attente sont des éléments différents du runtime.

69.9 Fournisseurs locaux

Le choix du fournisseur est configuré par le profil d'application ou de runtime sélectionné, et non codé en dur dans la syntaxe CLI. La documentation du runtime mentionne notamment des fournisseurs locaux comme Ollama et llama.cpp. Les sources documentaires fournies ne contiennent pas de section publique canonique `.lgc` pour ces deux fournisseurs ; ce guide n'en invente donc aucune. Une configuration prête à copier doit provenir d'un projet initial versionné ou du schéma officiel de configuration pour la distribution cible. Une fois configurée, elle s'exécute par les mêmes commandes `agent run` ou de session persistante.

L'interchangeabilité des fournisseurs ne modifie pas le contrat d'exécution conceptuel : Goal, connaissances, politique, confirmation, exécution d'action ou d'outil et état de session enregistré restent des responsabilités du runtime.

69.10 Tester un scénario d'agent

Un scénario de régression destiné aux développeurs doit rendre observable chaque frontière :

1. Démarrer le point d'accès Tooling persistant ou s'y connecter.
2. Ouvrir une session.
3. Soumettre un objectif avec un nombre limité d'itérations.
4. Fournir une source de connaissances connue si le scénario en dépend.
5. Examiner les demandes en attente au lieu de supposer qu'un outil à effets a été exécuté.
6. Exercer les deux chemins de confirmation et de rejet lorsque cela s'applique.
7. Capturer un instantané de la session.
8. Évaluer et fermer la session.

9. Répéter avec `-json` dans les automatisations afin que les assertions portent sur l'enveloppe de résultat stable.

Le test doit également vérifier les comportements négatifs : un rejet ne doit pas être assimilé à une exécution réussie, et la seule découverte d'une capacité ne suffit pas à la considérer comme autorisée.

69.11 Effets externes typés et preuves associées

Une opérationnalisation MCP sortante peut transformer la réponse d'un fournisseur en liaisons de connaissances typées et en preuves associées. Distinguez la réponse brute de l'outil, le reçu métier vérifié et le contexte transmis à l'agent. Lorsque le runtime fournit ces types, examinez le résultat et sa provenance avant toute étape suivante. Une réponse obtenue ne constitue pas, à elle seule, une autorisation de conservation comme connaissance persistante ou partagée.

Pour toute opération à effets, vérifiez les permissions requises, les arguments soumis et la demande de confirmation dans la configuration qui exécutera réellement l'action.

Chapitre 70

Contrat d'exécution gouvernée des agents

Ce chapitre spécifie, pour les développeurs, la frontière d'autorité commune aux agents résidents, aux appels de protocoles entrants et aux outils produisant des effets. Il distingue les règles normatives des affirmations de disponibilité. Une exigence formulée ici ne prouve pas que tous les chemins de protocole l'implémentent déjà : la matrice des capacités enregistre ce qui est établi par les éléments de validation fournis avec la version.

70.1 Modèle d'autorité

Le runtime doit distinguer publication, découverte, proposition, autorisation, confirmation et exécution. Les règles suivantes sont normatives :

— Publier ne signifie pas autoriser.
— Découvrir ne signifie pas autoriser.
— Le résultat d'un fournisseur est une proposition, pas un droit d'exécution.
— Une opération à effets ne doit pas être déclenchée avant satisfaction de toutes les conditions requises de politique et de confirmation.
— L'autorisation doit être évaluée par rapport au contexte d'exécution effectivement utilisé lors du dispatch.

Voici une séquence d'états conceptuelle utile :

```
existant -> publie -> decouvrable -> propose
         -> evaluation de la politique
         -> autorise | refuse | confirmation en attente
         -> confirme (si necessaire)
         -> revalide
         -> execute une seule fois
         -> resultat enregistre / audit
```

70.2 Contexte d'exécution indépendant du protocole

La gouvernance doit être attachée à l'action et à son contexte d'exécution, et non à un protocole de publication particulier. Le contexte logique d'une opération gouvernée inclut, lorsqu'ils sont disponibles :

— l'identité de l'appelant ou de l'agent ;
— l'identité du sujet ;
— la finalité ;
— la session propriétaire ;

— l'action ou l'outil demandé, avec ses arguments typés ;
— le résultat et le contexte de `KnowledgeQuery` ;
— les éléments de preuve satisfaisant les `EvidenceRequirements` ;
— le `PolicyContext` ;
— l'état de confirmation ;
— le résultat du dispatch et l'identifiant de trace ou d'audit.

La CLI expose directement plusieurs de ces éléments via `-agent-id`, `-subject-id`, `-purpose`, les options de connaissance, les identifiants de session et ceux des demandes en attente.

70.3 Métadonnées d'action gouvernée

Le contrat logique d'une action doit pouvoir représenter les dimensions suivantes, même si le schéma concret YAML/XML emploie d'autres noms :

Publication L'action est-elle exposée à l'extérieur ?

Classement des effets L'exécution peut-elle modifier l'état ou produire un effet externe ?

Rattachement à une politique Quelle politique ou quel `PolicyContext` gouverne l'appel ?

Exigence de confirmation Une confirmation est-elle inutile, dépendante de la politique ou obligatoire ?

Exigences de preuve Quels éléments doivent être présents avant l'autorisation ?

Identité d'audit Quel identifiant stable permet d'enregistrer l'opération demandée et son issue ?

Cette spécification n'invente volontairement aucune syntaxe concrète de déclaration. Celle-ci doit provenir du schéma modèle/configuration faisant autorité et d'un projet initial validé, avant d'être publiée dans un exemple de code pour les développeurs.

70.4 Sémantique de PendingConfirmation

Lorsqu'une confirmation est nécessaire, le runtime entre dans l'état `PendingConfirmation` au lieu de lancer l'opération.

70.4.1 Rejet

Une opération de rejet :
— NE DOIT PAS déclencher l'opération en attente ;
— DOIT résoudre la demande en attente identifiée ;
— DOIT conserver la session propriétaire, sauf si un échec distinct de session survient ;
— DOIT rendre le rejet observable par les surfaces de diagnostic et d'audit.

70.4.2 Confirmation

Une opération de confirmation :
— DOIT identifier une demande en attente précise ;
— NE DOIT PAS autoriser silencieusement des arguments matériellement différents ;
— DOIT revalider les conditions du runtime susceptibles d'évoluer entre proposition et exécution, ou garantir autrement leur stabilité ;
— NE DOIT autoriser qu'un seul déclenchement pour cette confirmation ;
— DOIT enregistrer le résultat réel de l'exécution afin que la session reprenne sur ce résultat et non sur le récit du fournisseur.

Le rejeu ou la duplication d'un identifiant de confirmation NE DOIT PAS provoquer un second déclenchement de la même opération à usage unique.

70.5 Exécution d'outils MCP sortants

Lorsque logiCells agit comme client MCP, la séquence gouvernée est conceptuellement la suivante :

```
Goal de l'agent
 -> decouverte MCP externe
 -> selection dans le catalogue des capacites
 -> evaluation de PolicyContext
 -> lecture seule OU PendingConfirmation
 -> confirmation/rejet si necessaire
 -> appel MCP externe au plus une fois
 -> resultat renvoye a la meme session
 -> instantane / audit
```

Un outil découvert qui ne satisfait pas les contrôles de politique, de preuve ou de confirmation doit rester inexécutable dans ce contexte.

70.6 Exécution entrante d'actions publiées par MCP

Lorsqu'un client MCP externe appelle une action publiée par logiCells, la séquence gouvernée cible est :

```
client MCP externe
 -> serveur MCP
 -> resolution de l'action publiee
 -> validation des arguments types
 -> construction du contexte d'execution
 -> PolicyContext / EvidenceRequirements
 -> autoriser | refuser | PendingConfirmation
 -> confirmation/rejet si necessaire
 -> revalidation
 -> execution au plus une fois
 -> resultat + audit
```

La publication d'une application ou d'un package crée un point d'accès protocolaire, pas une politique d'autorisation. Vérifiez les appels entrants à effets selon les exigences ci-dessus, avec l'adaptateur et le runtime réellement déployés. N'attribuez pas à la seule publication des garanties de confirmation ou d'exécution unique.

70.7 OpenAPI et autres adaptateurs de publication

Le même principe s'applique à OpenAPI et aux futurs adaptateurs. Si la gouvernance fait partie du contrat de l'action et du runtime, sa projection dans un protocole ne doit pas l'affaiblir. Un adaptateur peut proposer d'autres mécanismes de transport, mais il ne doit pas contourner les exigences de politique, de preuve ou de confirmation propres à l'action.

70.8 Scénarios de régression requis

Toute version revendiquant une exécution gouvernée devrait posséder des tests versionnés couvrant le chemin concerné :

1. une opération en lecture seule autorisée s'exécute sans confirmation superflue ;
2. une opération refusée ne provoque aucun dispatch ;
3. une opération à effets exigeant une confirmation passe en `PendingConfirmation` ;
4. un rejet ne déclenche rien et laisse la session utilisable ;
5. une confirmation provoque exactement un dispatch ;
6. le rejeu de la confirmation ne déclenche aucune seconde exécution ;
7. une modification des arguments protégés entre proposition et dispatch est rejetée ou revalidée ;
8. `KnowledgeQuery` et les exigences de preuve se ferment en mode sûr si les preuves obligatoires manquent ;
9. les résultats et les rejets restent observables dans l'instantané et les traces d'audit de la session ;
10. les chemins MCP entrant et Agent résident respectent les mêmes règles d'autorité au niveau de l'action lorsqu'ils ciblent une même action gouvernée.

70.9 Critères d'acceptation du projet de démarrage

Le projet canonique illustrant ce contrat doit permettre à un développeur de reproduire le parcours suivant sans dépendre de chemins internes au dépôt :

```
action publiee
 -> regle de politique / preuve / confirmation configuree
 -> fournisseur + endpoint Tooling persistant
 -> ouverture de session
 -> Goal + KnowledgeQuery
 -> demande en attente
 -> rejet => demontrer l'absence d'effet
 -> nouvelle demande
 -> confirmation => un seul effet
 -> reprise de la session
 -> inspection de l'instantane et de l'audit
```

Un second exercice doit exposer cette même action gouvernée par MCP et établir si le chemin entrant traverse la même frontière d'autorité. La configuration du projet de démarrage fait elle-même partie du contrat développeur et doit être versionnée avec la distribution.

70.10 Vérifier les actions publiées avant déploiement

Avant d'exposer une action par MCP, OpenAPI ou un autre protocole, vérifiez que l'adaptateur déployé applique bien l'authentification, les autorisations et les confirmations exigées sur le chemin réel des appels entrants. Ne publiez que les opérations nécessaires. En préproduction, testez les demandes refusées, répétées ou annulées autant que les appels réussis. La disponibilité d'un adaptateur de publication ne garantit pas, à elle seule, la gouvernance complète de l'exécution.

70.11 Séparer assistance au développement et connaissances partagées

Les opérations applicatives doivent rester distinctes des droits de modification du projet. Un agent autorisé à exécuter une action métier ne doit pas obtenir automatiquement l'accès en écriture aux modèles sources. De même, une preuve contextuelle renvoyée par un outil n'est pas nécessairement destinée à une mémoire persistante et partagée. Définissez explicitement les portées, les validations et la conservation des informations avant d'activer ces usages.

Chapitre 71

Processus conceptuels avec intervention humaine

De nombreux processus ne peuvent pas être entièrement automatisés. Ils exigent une validation, une saisie, une correction, une approbation ou une interprétation humaine. Dans logiCells, cette participation doit être modélisée comme une étape du processus, et non dissimulée dans les écrans.

71.1 Les tâches humaines comme étapes de processus

Une tâche humaine peut être représentée par une action, une vue, un état ou un objet d'activité. Le processus conserve le point où l'intervention est attendue ainsi que le résultat requis.

71.2 Les vues comme interfaces de tâche

La vue utilisée pour une tâche humaine est une projection du processus ou de l'objet métier. Elle ne doit pas porter elle-même le processus ; elle recueille ou affiche les informations utiles à l'état courant.

71.3 Assistance par l'IA

Un assistant d'IA peut résumer la situation, proposer une décision, repérer des informations manquantes ou préparer une réponse. La validation reste sous la responsabilité de la personne lorsque le processus l'exige.

71.4 Règles de conception

Rendez les tâches humaines explicites. Enregistrez l'état du processus avant et après l'intervention. Utilisez des événements pour signaler la fin de la tâche. Conservez à la vue son rôle d'interface, sans lui confier l'orchestration. Autorisez les agents à assister les utilisateurs sans contourner les règles de validation.

71.5 Synthèse

Les processus avec intervention humaine sont plus faciles à maintenir lorsque la participation des personnes fait partie du modèle de processus, plutôt que de dépendre d'un effet de bord implicite de l'interface utilisateur.

Chapitre 72

Explicabilité, traçabilité et gouvernance

Les modèles conceptuels exécutables doivent pouvoir être expliqués. Un utilisateur, un auditeur, un développeur ou un agent doit pouvoir comprendre pourquoi un objet existe, quel processus l'a modifié, quelle action a été appelée et quel événement a été émis.

72.1 Une explicabilité fondée sur la structure

logiCells facilite l'explication parce que le modèle nomme les concepts, les rôles, les actions et les processus. Une explication peut ainsi citer les éléments du modèle, plutôt que des chemins d'exécution de code anonymes.

72.2 La traçabilité par les événements et les journaux

Événements et journaux d'exécution se complètent. Les événements décrivent les changements significatifs ; les journaux consignent les détails de l'exécution. Une bonne trace relie l'objet, l'action, l'événement, l'état du processus et le contexte de l'utilisateur ou de l'agent.

72.3 Gouvernance des capacités publiées

Les actions publiées doivent faire l'objet d'une gouvernance. Le modèle doit indiquer clairement quelles capacités sont exposées, quels paramètres sont acceptés, quels rôles peuvent les appeler et quels effets elles produisent.

72.4 Gouvernance de l'IA

Lorsque des assistants d'IA exploitent des modèles conceptuels, la gouvernance devient encore plus importante. L'assistant doit passer par des actions explicites et des contextes autorisés ; il ne doit ni contourner les contraintes du modèle ni inventer des capacités qui n'y figurent pas.

72.5 Règles de conception

Nommez clairement les actions et les événements importants. Représentez explicitement les états des processus. Journalisez les transitions d'état et les appels de services. Évitez les effets de bord cachés. Traitez les actions publiées comme des contrats gouvernés.

72.6 Synthèse

L'explicabilité ne doit pas être ajoutée après coup : elle découle de la modélisation explicite des concepts, actions, événements, états et frontières de publication.

Dixième partie

Pratiques d'ingénierie conceptuelle

Chapitre 73

Tester les modèles conceptuels

Tester une application logiCells ne revient pas à tester seulement du code. Il faut également vérifier les concepts, relations, contraintes, actions, événements, processus, publications de services et projections de vues. Un modèle conceptuel étant exécutable, il doit être validé en tant qu'artefact exécutable.

73.1 Objectif

Ce chapitre introduit les bonnes pratiques de test des modèles conceptuels déclaratifs. Il s'agit de sécuriser leur évolution et de détecter les incohérences avant qu'elles ne se manifestent au runtime. Les tests applicatifs doivent s'exprimer aussi près que possible de la surface conceptuelle utilisée par l'application elle-même.

73.2 Que faut-il tester ?

Une stratégie complète devrait couvrir :
— le chargement des fichiers de modèle ;
— l'héritage des concepts et leurs implications ;
— les facettes hypergraphiques et de base de données ;
— les chemins de rôles et les projections d'objets liés ;
— les fournisseurs de collections et les jointures ;
— les signatures et l'exécution des actions ;
— l'accès aux actions publiées ;
— le routage des événements ;
— les transitions d'état des processus ;
— les bindings de vues et les datasets ;
— l'exposition des services et le mapping des requêtes.

73.3 Structurer un package de tests déclaratifs

Conservez les tests applicatifs dans un package dédié lorsqu'ils ne sont pas requis par l'application de production. Un module typique peut ainsi séparer son modèle de production et ses tests :

```
my-module/
  packages/
    MyModule.package.yaml
    MyModule.Tests.package.yaml
  meta/
    models/
      MyModule/
```

```
        _.manifest.yaml
        Customer.model.yaml
      MyModuleTests/
        _.manifest.yaml
        CustomerModelTests.model.yaml
        CustomerActionTests.model.yaml
  src/
  tests/
```

Listing 73.1 – Organisation recommandée du module et des tests

Le package de tests suit les règles normales de chargement des packages. Son répertoire de schéma doit rendre le manifeste de tests accessible, et l'application utilisée pour les tests doit charger à la fois le package applicatif et le package de tests. La simple présence d'un modèle de test dans l'arborescence source ne suffit pas à l'inclure dans l'exécution.

Un descripteur de package de tests adopte la même structure qu'un package applicatif :

```yaml
package:
  items:
    - MyModuleTests:
        caption: '#MyModuleTests'
        enabled: true
  files:
    schemaDirectories:
      schemaPath:
        value: '[Modules]\my-module\meta\models'
```

Listing 73.2 – Descripteur du package de tests

Son manifeste déclare normalement les modèles de suites :

```yaml
package:
  objectsContainer: CustomObjectsContainer
  plugginCaption: My Module Tests
  uri: http://www.logicells.net/schema/MyModuleTests
  classes:
    - CustomerModelTests:
        classId: myapp#customer_model_tests
        type: Entity
    - CustomerActionTests:
        classId: myapp#customer_action_tests
        type: Entity
```

Listing 73.3 – Manifeste du package de tests

73.4 Déclarer une suite de tests sous forme de modèle

Une suite déclarative reste un modèle ordinaire. Son concept `Entity` implique `lgc#unit_test`. Le runtime reconnaît ainsi une suite de tests unitaires intégrée au modèle applicatif déclaratif.

```yaml
class:
  name: CustomerModelTests
  classId: myapp#customer_model_tests
  className: CustomClassItem
  currentGeneratorName: MyModuleTests.CustomerModel
  type: Entity

  generators:
    - MyModuleTests.CustomerModel:
        class: GUIDStringGeneratorInfo
        params: ''

  object:
    className: CustomObjectItem
    objectName: CustomerModelTests

  concepts:
```

```
    - Entity:
        constraints:
          - implies: lgc#unit_test
```

Listing 73.4 – Suite de tests déclarative minimale

Ne remplacez pas le concept `Entity` par le seul attribut `type: Entity`. La suite doit participer au modèle conceptuel en tant qu'Entity et impliquer explicitement `lgc#unit_test`.

73.5 Un cas de test par action déclarée

Chaque Action déclarée directement par le modèle de suite constitue un cas de test. Pour un test applicatif scripté, utilisez `UnitTestScriptableObjectElementAction`. Nommez l'action de manière descriptive, en précisant le contrat qu'elle protège.

```
actions:
  - action:
      name: Customer_Class_Is_Loaded
      class: UnitTestScriptableObjectElementAction
      type: ClassItem
      caption: '#Customer_Class_Is_Loaded'
      published: true
      scripts:
        - script:
            clientScript: true
            serverScript: false
            language: functionalP
            classes: Desktop
            cdata: |
              TargetClassItem := Self.ObjectsContainer.ClassItemByClassId('myapp#customer');
              Self.Assert(
                GUnitTest,
                (TargetClassItem <> nil),
                'Customer ClassItem is not loaded'
              );
```

Listing 73.5 – Cas de test du chargement d'un modèle

L'Action de test peut inspecter la même surface conceptuelle du runtime que les autres actions applicatives. Elle convient donc pour vérifier le chargement des classes, la présence des métadonnées, la disponibilité des actions, l'existence des états de processus et la conformité du résultat produit par l'application.

Un cas de test peut comprendre plusieurs assertions si elles portent sur un même contrat cohérent. Privilégiez plusieurs petites Actions plutôt qu'une action de test unique vérifiant des aspects sans rapport de l'application.

73.6 Assertions en functionalP

Dans une Action de test unitaire, utilisez `Self.Assert` avec `GUnitTest` :

```
Self.Assert(
  GUnitTest,
  condition,
  'Message explaining the expected contract'
);
```

Listing 73.6 – Modèle d'assertion functionalP

Le message doit identifier le contrat applicatif absent ou rompu. Préférez des messages précis comme `PromptDefinition.Version metadata missing` ou `Start action is missing` plutôt qu'un message générique tel que `Test failed`.

Les tests devraient généralement utiliser les identités publiques du modèle, ses métadonnées et ses Actions déclarées. Évitez de reproduire les algorithmes d'implémentation dans un script de test pour comparer deux copies de la même logique.

73.7 Organiser les suites par contrat

Pour un module non trivial, répartissez les tests selon la frontière conceptuelle protégée. Une organisation utile est :

```
MyModuleTests
  CustomerModelTests
    chargement de la classe
    proprietes obligatoires
    relations et implications

  CustomerActionTests
    presence des actions
    contrats des parametres
    valeurs retournees

  CustomerProcessTests
    etats declares
    Start / Continue / Retry / Cancel
    contrats de transition et de retour
```

Listing 73.7 – Exemple d'organisation des suites de tests

Pour `logiCells.Agent`, cette structure conduit naturellement à `AgentModelTests`, `AgentActionTests` et `AgentProcessTests`. L'objectif est la lisibilité et la facilité du diagnostic, et non la reproduction des couches internes du framework.

73.8 Exécuter les tests déclaratifs avec sclgc

La configuration applicative utilisée pour les tests doit charger le module testé et son package de tests. Lancez l'application avec `-t` :

```
sclgc -c /path/to/application.application.ACC.lgc -t
```

Listing 73.8 – Exécuter tous les tests applicatifs enregistrés

La forme longue est équivalente :

```
sclgc --config /path/to/MyApplication.ACC.lgc --test
```

Listing 73.9 – Lancer les tests avec des noms de paramètres longs

Pour enregistrer les résultats dans un répertoire précis, ajoutez `-to` ou `-test-out` :

```
sclgc -c /path/to/MyApplication.ACC.lgc -t -to /path/to/test-results
```

Listing 73.10 – Lancer les tests avec un répertoire de résultats explicite

Le modèle de test déclaratif est indépendant de la plateforme, mais l'exécutable hôte des tests et son packaging dépendent de la distribution. Avant de placer cette commande dans une intégration continue, vérifiez que le package du runtime cible expose les mêmes options d'hôte de tests sur la plateforme choisie.

73.9 Tester les actions publiées

Les actions publiées méritent une attention particulière, car elles peuvent être invoquées par des interfaces utilisateur, processus, agents ou services web. Un test doit vérifier la signature de

l'action, le mapping des paramètres, le comportement de validation, le résultat retourné et les effets produits.

Les tests de contrat sont particulièrement utiles ici, puisque la déclaration est partagée entre plusieurs appelants. Ils doivent protéger la stabilité des noms et directions des paramètres, des types de données pertinents et des effets métier attendus.

73.10 Tester les processus

Les tests de processus doivent préserver les contrats observables du workflow plutôt que reproduire son implémentation. Vérifiez notamment que :

— les états obligatoires sont déclarés ;
— les actions d'entrée et de continuation existent ;
— les contrats de nouvelle tentative et d'annulation restent disponibles lorsqu'ils sont nécessaires ;
— les informations de résultat et d'erreur restent observables ;
— la validation métier intervient avant l'état final ou la matérialisation, lorsqu'elle est exigée.

Lorsqu'un processus appelle un agent ou un service externe, les assertions sur les états du processus doivent rester déterministes. Les tests de qualité des sorties du modèle et de disponibilité d'un fournisseur externe appartiennent à des suites d'intégration distinctes.

73.11 Tester les comportements asynchrones

Les comportements asynchrones doivent être testés par leurs événements et messages. Vérifiez qu'un événement est émis, que le bon gestionnaire est déclenché, que l'état du processus évolue correctement et qu'aucune hypothèse cachée propre au client n'est nécessaire.

73.12 Tester les vues

Un test de vue ne doit pas commencer par vérifier l'apparence. Il doit d'abord contrôler les bindings : curseur d'entrée, datasets, sources de données, chemins de collections et liaisons d'actions. Le rendu concret relève de l'implémentation du composant enregistré.

73.13 Régression et évolution du modèle

Lorsqu'un modèle évolue, les tests doivent protéger ses contrats publics. Les signatures d'actions publiées, identifiants de classes, chemins de rôles, états de processus et mappings de services ne doivent pas changer accidentellement. Tout changement volontaire doit apparaître dans les tests et la documentation.

Une bonne règle consiste à ajouter ou mettre à jour un test déclaratif dès qu'une évolution introduit un nouveau contrat applicatif stable. Cela maintient la cohérence entre exemples, comportement du runtime et attentes applicatives.

73.14 Synthèse

Une suite de tests déclaratifs est un modèle qui implique `lgc#unit_test`. Chaque Action déclarée par la suite constitue un cas de test ; les tests scriptés peuvent appeler `Self.Assert(GUnitTest, ...)`. Le package de tests est chargé par les mêmes mécanismes de packages et de manifestes que les autres modèles applicatifs. Lorsque ce package figure dans la configuration de l'application, `sclgc -c <application.lgc> -t` exécute les tests applicatifs.

Chapitre 74

Déboguer les applications déclaratives

Déboguer une application déclarative consiste à suivre l'exécution du modèle, et pas seulement à avancer pas à pas dans un code procédural.

74.1 Parcours de diagnostic

Commencez par le concept. Vérifiez son identifiant de classe, ses facettes, chemins de rôles, collections, actions et événements. Examinez ensuite l'objet du runtime et le contexte d'évaluation courant.

74.2 Défaillances fréquentes

Les erreurs courantes comprennent les chemins de rôles manquants, les identifiants de classes incorrects, les modifications non enregistrées, les chemins de collections erronés, les jeux de données non liés, les actions non publiées et les événements sans abonnés.

74.3 Journalisation

Les journaux doivent contenir les noms des éléments du modèle. Une ligne telle que `[RiskReviewProcess.Finalize]` renseigne davantage qu'un message générique.

74.4 Déboguer le modèle avant le script

Si un comportement est inattendu, contrôlez d'abord la déclaration : identifiant de classe, propriétaire et type de l'action, liaison événementielle, chemin de collection et cible de la vue. De nombreux problèmes du runtime proviennent d'un chemin déclaratif incorrect plutôt que du script lui-même.

74.5 Ordre de vérification

Un ordre efficace est le suivant : chargement du modèle, résolution du concept, création de l'objet, fournisseur de collection, liaison de l'action, acheminement des événements, curseur de la vue, puis persistance. Cet ordre respecte la chaîne de dépendances du runtime.

74.6 Utiliser des noms explicites dans les traces

Incluez les noms conceptuels dans les journaux : identifiant de classe, identifiant d'objet, nom d'action et d'événement, état du processus. Les traces deviennent ainsi utiles aux développeurs, aux équipes de support et aux outils de diagnostic assistés par l'IA.

74.7 Diagnostic du chargement et de la sémantique

Plusieurs messages de diagnostic correspondent à des étapes distinctes de la chaîne déclarative. Interprétez-les comme des indices sur la couche en échec, plutôt que comme des erreurs sémantiques indifférenciées.

Le modèle n'apparaît jamais dans le journal de compilation. Vérifiez la configuration de l'application, l'activation du package, le chemin de schéma et l'appartenance au manifeste.

Enregistrement Entity absent. La classe peut déclarer `type: Entity` sans enregistrement conceptuel `Entity` explicite. Examinez `class/concepts`.

Aucun mappage polyadique entre deux concepts. Vérifiez l'enregistrement sémantique des deux concepts et la possibilité d'aligner leurs rôles positifs. Pour les entités ordinaires, vérifiez la sémantique implicite de `Me` avant d'ajouter un mappage explicite.

Identifiant classId dupliqué. Recherchez tous les packages et manifestes actifs déclarant cet identifiant. Utilisez un autre identifiant pour un autre sens ; ne recourez à la redéfinition explicite que pour un remplacement intentionnel.

Source différente de celle attendue. Examinez le chemin physique du package ou modèle effectivement résolu. Un deuxième répertoire de travail ou un package historique peut être actif même lorsque le fichier modifié est correct.

Pour un module nouvellement créé, un premier test de démarrage en ligne de commande, lorsqu'une distribution fournit l'hôte de runtime `sclgc`, est le suivant :

```
sclgc -c <path-to-application.ACC.lgc>
```

Listing 74.1 – Test de démarrage du runtime déclaratif

Lisez ensuite le journal dans cet ordre : application, package, manifeste, modèle, collections, initialisation sémantique, démarrage du runtime. Arrêtez l'analyse à la première couche où le module attendu disparaît ou où une erreur structurelle est signalée.

Pour le diagnostic en ligne de commande, n'oubliez pas que le runtime accepte des noms de paramètres longs et courts, par exemple `-debug`/`-dbg`, `-log-level`/`-ll`, `-test`/`-t`, `-test-out`/`-to`, `-modules`/`-m` et `-services`/`-sg`. Un paramètre prenant une valeur consomme l'argument suivant ; un indicateur simple n'en consomme pas. Avec une validation stricte, les paramètres inconnus et les valeurs obligatoires manquantes sont des erreurs, non des entrées ignorées silencieusement.

74.8 Déboguer une action FunctionalP intégrée à YAML

Un bloc FunctionalP intégré introduit deux systèmes de coordonnées : la ligne du fichier YAML physique et la ligne au sein de l'AST du script compilé. Le débogueur interactif utilise `ScriptIdentity` et les lignes de l'AST ; l'éditeur doit maintenir une correspondance dans les deux sens à partir des informations de provenance produites par le parseur, sans renuméroter les lignes de l'AST. La spécification VS Code v1.0 exige que points d'arrêt et cadres de pile soient projetés vers les lignes YAML physiques, avec des états en attente, vérifié, non résolu et périmé, ainsi qu'une nouvelle résolution tenant compte de la génération. Il s'agit d'une intégration ciblée,

qui ne constitue pas encore une fonctionnalité entièrement certifiée dans le snapshot fourni de l'extension et du runtime. Pour les détails, le Chapitre 75, "Débogage source de FunctionalP dans YAML" présente le mappage source, le socle CLI et les critères de test.

74.9 Synthèse

Le parcours de débogage le plus rapide suit la structure conceptuelle : déclaration, objet du runtime, action, événement, vue, puis service.

Chapitre 75

Déboguer FunctionalP au niveau source dans les modèles YAML

75.1 Fonctionnalités disponibles et cible de développement

Le débogueur natif exécute des programmes FunctionalP/AST compilés. La CLI `cell debug` se connecte au module de débogage d'un processus `sdlgc --ast-debug` actif. Pour placer un point d'arrêt sur une ligne physique d'un fichier `.model.yaml` ou `.models.yaml`, il faut en outre un adaptateur d'éditeur assurant la correspondance entre la source YAML et le script compilé. Vérifiez que l'adaptateur installé propose cette fonction.

75.2 Séparer conception et exécution

```
VS Code editor
  |-- ToolingClient -> logicells-tooling JSON-RPC
  |     MODEL / APPLICATION / SEMANTIC / SCRIPT / compile-plan
  |
  `-- Debug Adapter -> RuntimeDebugClient -> sdlgc --ast-debug
        processes / breakpoints / stack / locals / steps / REPL
```

Le Tooling détient l'autorité sur l'analyse des sources, la résolution sémantique et la compilation. Le moteur de débogage du runtime détient celle des arrêts, de l'état d'exécution et de l'avancement pas à pas. VS Code est client de ces deux autorités : cela ne justifie pas de dupliquer les parseurs ou le moteur FPL en TypeScript.

75.3 La provenance source est un produit de compilation à part entière

Tout script embarqué qui doit être débogué à partir de ses sources nécessite une correspondance stable entre son identité à l'exécution et son document physique :

Identité de la source `SourcePath`, `SourceDocumentId`, `SourceUnitId` et, éventuellement, une identité métier telle que `Package::Class@Action`.

Emplacement `ContentStartLine`, `ContentEndLine`, éventuellement la colonne, et la nature du contenu : `yaml-literal`, `fp`, `xml` ou code généré.

Identité d'exécution `ScriptIdentity`, ligne AST, lignes exécutables et génération de compilation concernée.

Pour un scalaire littéral YAML `|`, lorsque la relation entre lignes physiques et contenu est déterministe, la conversion élémentaire est :

$$\text{AstLine} = \text{SourceLine} - \text{ContentStartLine} + 1, \qquad \text{SourceLine} = \text{ContentStartLine} + \text{AstLine} - 1.$$

Les lignes vides internes comptent. L'AST conserve sa numérotation locale : modifier le compilateur FunctionalP pour simuler les numéros de ligne du document romprait la séparation des responsabilités. Un scalaire YAML replié `>` ne permet pas nécessairement cette conversion simple ; il faut alors refuser son débogage au niveau source ou utiliser une table de correspondance explicitement produite par le parseur.

75.4 Identité et état des points d'arrêt

La clé canonique d'un point d'arrêt associe `ScriptIdentity` à `AstLine` : elle n'est ni le chemin du fichier seul ni un simple numéro de ligne. Deux Actions d'un même fichier YAML peuvent chacune posséder une instruction AST à la ligne 1. L'interface du débogueur `logiCells.Debugger.WebPage.pas` accepte `scriptIdentity` sur la route examinée pour les points d'arrêt, et le moteur client expose cette identité dans les cadres de pile. C'est un prérequis utile côté runtime, mais non la preuve que le registre de correspondance YAML (*SourceMapRegistry*) est terminé.

PENDING Emplacement source connu, mais script compilé ou ligne exécutable encore indisponible.

VERIFIED Identité du script et ligne AST exécutable résolues pour la génération actuelle.

UNRESOLVED Ligne physique ne correspondant à aucune instruction exécutable ni à aucun script embarqué connu.

STALE Identité associée à une génération de compilation antérieure : la liaison doit être rétablie.

Il ne faut pas renvoyer un point d'arrêt DAP marqué *verified* au seul motif que la requête a été acceptée. Après une recompilation incrémentale, les anciennes liaisons aux scripts doivent être invalidées, puis résolues par rapport à la nouvelle génération. Un échec de résolution doit comporter un motif explicite. L'enregistrement des correspondances source doit intervenir en dehors de la vérification interne des arrêts du runtime : il ne faut pas analyser le YAML à chaque test de point d'arrêt.

75.5 Projection des sources dans DAP et commandes du runtime

Une requête `setBreakpoints` portant sur une ligne physique YAML identifie l'Action ou le script qui la contient, calcule la ligne AST, obtient l'identité courante du script, installe le point d'arrêt dans le runtime, puis transmet à VS Code le résultat réel de la résolution. Lors d'un arrêt, le runtime renvoie l'instruction et les identités de pile ; l'adaptateur les projette sur le fichier YAML et les lignes physiques. Les commandes `step`, `step-in` et `step-out` suivent le cadre d'exécution courant et doivent changer de document source lorsque l'exécution traverse plusieurs scripts embarqués.

Les commandes CLI existantes constituent une référence de débogage indépendante et utile :

```
cell server start <app.lgc> --foreground
cell server status
cell script compile <script.fp> --output <script.ast>
cell debug break add <script.ast> <astLine>
cell debug run <script.ast>
```

```
cell debug processes
cell debug stack <processpid>
cell debug locals <processpid>
cell debug step <processpid>
cell debug step-in <processpid>
cell debug step-out <processpid>
cell debug continue <processpid>
```

Le `processpid` renvoyé par `debug run` identifie un processus logique du runtime, et non le PID système de `sdlgc`. Les opérations de pas à pas, d'évaluation et d'arrêt disponibles dépendent de l'adaptateur installé. Une réponse valide comportant une liste vide de variables locales ne garantit pas que leur inspection soit configurée.

75.6 La correspondance source et la compilation incrémentale se rejoignent sur l'identité

Le condensat normalisé de la source, les signatures sémantiques et comportementales, l'index AST produit et la provenance source remplissent des fonctions différentes. La réutilisation d'une signature physique n'autorise pas à conserver un point d'arrêt rattaché à un ancien AST exécutable. Quand un script ou une dépendance pertinente évolue, les correspondances source et les liaisons des points d'arrêt doivent être invalidées selon leur version. À l'avenir, le rechargement à chaud pourra s'appuyer sur ce même modèle de provenance, mais il ne constitue pas un prérequis pour la première fonctionnalité de débogage dans YAML.

75.7 Scénarios d'acceptation

1. Deux scripts embarqués dans un même document, possédant les mêmes numéros de ligne AST, s'arrêtent indépendamment grâce à `ScriptIdentity`.
2. Un point d'arrêt placé dans un bloc littéral YAML correspond à la bonne instruction locale, y compris en présence de lignes vides.
3. Les scalaires repliés non pris en charge produisent un diagnostic explicite de correspondance impossible.
4. La pile, les pas à pas et les appels imbriqués reviennent au bon fichier YAML et à la bonne ligne physique.
5. Une recompilation rend les anciens points d'arrêt obsolètes ; ceux-ci sont ensuite correctement résolus ou signalés comme non exécutables.
6. Les erreurs du Tooling et du débogueur runtime restent distinguables ; aucune modification automatique des sources ni aucun rechargement à chaud n'est annoncé.

Ces scénarios décrivent le comportement attendu d'un adaptateur capable de cartographier les scripts intégrés au YAML. Avec la version installée, vérifiez la résolution des lignes physiques, la relocalisation des points d'arrêt après recompilation et les états affichés. À défaut de cet adaptateur, utilisez le débogueur de scripts compilés et les commandes du runtime.

Chapitre 76

Journalisation, diagnostic et traçabilité

La journalisation consigne l'exécution ; le diagnostic explique les échecs ; la traçabilité relie l'activité du runtime aux éléments du modèle.

76.1 Unités de trace

Les informations utiles comprennent l'identité de l'objet, son identifiant de classe, le nom de l'action ou de l'événement, l'état du processus, le contexte utilisateur ou agent et le canal de publication.

76.2 Diagnostic

Un message de diagnostic doit identifier l'élément du modèle concerné. Lorsqu'une vue ne peut pas s'ouvrir, indiquez son nom et son chemin. Lorsqu'une action échoue, précisez son nom et le paramètre manquant.

76.3 Traçabilité

Les événements et les états de processus fournissent une trace de niveau plus élevé que les seuls journaux techniques. Ils expliquent ce qui s'est produit dans les termes du métier.

76.4 Que faut-il journaliser ?

Journalisez les faits conceptuels, et pas uniquement les messages techniques. Une trace utile indique le concept, l'identifiant de l'objet, l'action, l'événement, l'état du processus et le résultat.

76.5 Diagnostic des événements

Dans une application pilotée par événements, le diagnostic doit montrer quel événement a été émis, quelle source l'a produit, quelles actions y étaient associées et si chaque gestionnaire s'est terminé correctement. L'exécution asynchrone devient ainsi auditable.

76.6 Traçabilité et gouvernance

La traçabilité soutient la gouvernance en rattachant une décision ou un changement d'état à l'élément du modèle qui l'a produit. C'est particulièrement important lorsque des services web, des agents ou des assistants d'IA appellent des actions publiées.

76.7 Synthèse

Un diagnostic efficace rend le modèle observable. Les journaux doivent servir l'architecture conceptuelle, et non la dissimuler.

Chapitre 77

Performances, profilage et complexité des modèles

La modélisation déclarative ne fait pas disparaître les enjeux de performance. Elle rend plutôt le chemin d'exécution plus explicite : activation des modèles, parcours de l'hypergraphe, exécution des providers, déplacement dans les jeux de données et les curseurs, expressions de chemin, actions, événements, sérialisation, publication des services, appels distants et ordonnancement des workers peuvent tous contribuer à la latence observée.

77.1 Aucun résultat de benchmark dans cette révision

Cette édition ne publie volontairement **aucun chiffre de débit, de latence ou de passage à l'échelle**. Les benchmarks relèvent d'une campagne de validation distincte. Cette absence de mesures ne doit pas être compensée par des affirmations qualitatives comme « rapide », « faible surcharge » ou « adapté aux grands graphes », sauf si un benchmark reproductible les étaye.

Le présent chapitre définit les grandeurs à mesurer et les choix de conception qui peuvent entraîner un travail superflu.

77.2 Mesurer le parcours complet

Une mesure utile précise les frontières temporelles de l'opération observée. Les couches typiques comprennent :

— le chargement des packages et des modèles, ainsi que l'activation du runtime ;
— l'analyse des requêtes H-Logic et leur exécution ;
— la recherche dans l'hypergraphe, son parcours et sa persistance ;
— le temps d'interrogation des providers de collections et la cardinalité des résultats ;
— la pagination et le rafraîchissement des DataSet/DataCursor dans les clients riches ;
— l'exécution des actions et les transitions des processus ;
— le routage des événements et messages ;
— la sérialisation et la publication des services ;
— les appels ABI ou RPC, y compris le *marshalling* et la réécriture des valeurs ;
— l'inférence IA locale ou résidente, le chargement du modèle et son exécution par requête.

Ne pas optimiser une couche uniquement parce qu'elle est visible. Mesurer d'abord depuis la frontière de l'appelant, puis décomposer le parcours jusqu'à identifier le coût dominant.

77.3 Complexité des modèles

La complexité augmente avec le nombre de concepts, relations, règles, jointures, événements et surfaces de publication actifs. Les volumes bruts ne suffisent toutefois pas à la caractériser. Le degré de ramification, la sélectivité des requêtes, les parcours répétés, la fréquence de rafraîchissement et les appels distants pèsent souvent davantage que le nombre de déclarations.

Privilégier les modèles dont les chemins fréquemment exécutés sont explicites et bornés. Une structure conceptuelle riche n'est pas en soi un défaut de performance ; les calculs répétés sans nécessité le sont.

77.4 Collections et jointures

Les collections et les jointures doivent garder une portée justifiée et bornée. Lorsque cela est possible, appliquer les filtres et le contexte avant les compositions coûteuses. Éviter de rafraîchir de larges jointures à chaque événement du client. Si le client n'a besoin que d'une page ou d'une projection, ne pas matérialiser toute la collection logique pour en éliminer ensuite l'essentiel.

77.5 Requêtes H-Logic et hypergraphiques

Lors d'une analyse de performance H-Logic, consigner le mode de requête, sa portée et la cardinalité des résultats. L'appariement homomorphe, l'unification et les requêtes de résolution n'empruntent pas nécessairement les mêmes parcours de recherche. Tout benchmark doit préciser la grammaire ou le mode de requête et la structure du jeu de données mesuré.

Pour une requête située sur un chemin critique, privilégier des points d'ancrage explicites et des motifs sélectifs plutôt qu'une exploration sans contrainte. Préserver d'abord la correction sémantique : ne jamais substituer une requête plus rapide mais plus faible à une requête sémantique sans documenter le changement de sens.

77.6 Frontières locales et distantes

La multiplication des échanges réseau peut dominer le coût d'opérations autrement peu onéreuses. Préférer des opérations runtime de granularité suffisante aux boucles du langage hôte qui récupèrent successivement un nœud, un champ ou une relation au travers de RPC. Le même principe vaut aux frontières ABI lorsque plusieurs conversions de paramètres peuvent être remplacées par une opération de plus haut niveau.

Pour les traitements longs, utiliser un état de processus, une exécution en arrière-plan ou une messagerie, plutôt que de bloquer un fil d'exécution d'interface utilisateur ou de requête.

77.7 Clients riches

Pour les clients fondés sur les View, mesurer les succès et échecs de cache, les rafraîchissements des DataSet, la pagination, la synchronisation et le nombre d'événements transmis au-delà du client. Un événement visuel local ne doit pas déclencher une opération serveur sauf si le modèle l'exige. Le guide complémentaire *Rich Client* détaille les modèles de conception propres au client.

77.8 Mesures du runtime IA

Pour évaluer une IA locale ou résidente, distinguer :

— le chargement du modèle et le démarrage ;

— la latence du premier token ;
— la génération de tokens en régime établi ;
— la réinitialisation du contexte et du cache KV ;
— l'empreinte mémoire ;
— le backend utilisé et la plateforme hôte.

Une comparaison n'a de sens que si le modèle, sa quantification, le contexte, les options de génération et le matériel sont maintenus constants.

77.9 Contrat de publication des benchmarks

Un futur benchmark publié par logiCells devrait fournir au minimum :
— la version du framework et du runtime, ainsi que l'identifiant du commit ou du build ;
— le système d'exploitation, l'architecture et le matériel ;
— la taille et la structure du jeu de données ou du modèle ;
— l'opération ou la requête exacte ;
— la politique d'échauffement et le nombre d'essais ;
— la médiane et une mesure de dispersion, et non le seul meilleur résultat ;
— l'état d'activation de la persistance, du réseau, des caches et de la synchronisation ;
— les sources ou fixtures permettant de reproduire la mesure.

En attendant de tels résultats, ce guide décrit les mécanismes de performance et la méthode de mesure, pas des performances garanties.

Chapitre 78

Modèles d'extension et erreurs fréquentes

Une extension est utile lorsqu'elle ajoute une capacité réutilisable sans affaiblir le contrat conceptuel. Les extensions les plus sûres distinguent clairement le sens métier, le contrat du runtime, l'implémentation et la publication.

78.1 Modèles d'extension recommandés

Privilégiez les capacités enregistrées, les adaptateurs explicites, les actions publiées, les services déclaratifs et les composants concrets remplaçables. Une extension doit mettre une nouvelle capacité à disposition du modèle sans obliger les auteurs métier à dépendre de son implémentation interne.

78.2 Points d'extension des composants et du runtime

Un bon point d'extension porte le nom de la capacité qu'il fournit. Les composants visuels doivent être sélectionnés à partir d'une intention fonctionnelle et d'un enregistrement. Les adaptateurs du runtime doivent exposer des contrats stables. Des classes propres à l'implémentation peuvent figurer dans les liaisons lorsque la grammaire actuelle des métadonnées l'exige, mais elles ne constituent pas des concepts métier et ne doivent pas devenir le vocabulaire du domaine. Les bindings générés dans le langage hôte offrent une projection de programmation distincte des capacités publiées par le runtime : ils ne doivent pas être réinjectés dans l'ontologie ni dans le vocabulaire déclaratif au seul motif qu'une opération utilise ce langage.

78.3 Erreurs fréquentes

Évitez quatre erreurs récurrentes :

— placer un processus métier dans une extension opaque plutôt que dans un processus ou une action déclarés ;
— exposer une fonction utilitaire d'implémentation comme capacité publique ;
— coupler un modèle conceptuel à une implémentation unique d'interface, de transport ou de stockage alors qu'une frontière enregistrée existe ;
— considérer un résultat généré ou externe comme valide avant son passage par la validation du parseur, de la grammaire ou du runtime concerné.

78.4 Règle de sûreté pour les extensions

Une bonne extension apporte des capacités sans rompre le contrat conceptuel. Le modèle reste compréhensible, le runtime peut toujours l'inspecter et l'implémentation peut être remplacée sans redéfinir le concept métier.

78.5 Synthèse

Étendez les capacités, non les détails accidentels d'implémentation. Préservez la surface conceptuelle, explicitez les liaisons d'implémentation et respectez les frontières de validation et de publication.

Chapitre 79

Gouvernance de l'évolution des modèles

L'évolution des modèles doit être gouvernée, car les modèles logiCells constituent des contrats exécutables.

79.1 Catégories de changements

Une modification peut ajouter un concept, changer un rôle, faire évoluer la projection d'une facette, ajouter une action, publier un service, introduire un événement ou modifier l'état d'un processus. Chaque catégorie entraîne des conséquences différentes sur la compatibilité.

79.2 Compatibilité

Modifier la signature d'une action publiée revient à modifier son contrat. Changer un chemin de rôle peut affecter la persistance et les vues. Renommer un événement peut rompre les abonnements. Ces changements doivent être examinés et versionnés.

79.3 Documentation

Le présent guide doit lui aussi suivre les règles de gouvernance des modèles : conserver des exemples appariés, éviter les noms propres à un client dans les contenus publics et séparer clairement les concepts de plateforme des exemples applicatifs.

79.4 Synthèse

La gouvernance permet aux applications conceptuelles d'évoluer sans affaiblir la confiance accordée à leurs modèles exécutables.

Onzième partie

Conclusion et références

Chapitre 80

Fonctionnalités du runtime et liste de vérification

Ce chapitre propose une liste de vérification destinée aux développeurs. Elle aide à choisir les interfaces d'intégration et à identifier les paramètres à contrôler pour chaque déploiement. Les commandes exactes et les modules facultatifs dépendent de la version installée ; consultez la documentation de référence correspondante.

80.1 Parcours essentiels du développeur

Domaine	**Points à contrôler dans l'application**
MODEL et assemblage des applications	Valider les déclarations YAML, les packages importés et l'application assemblée avec le compilateur qui sera déployé.
Requêtes H-Logic et hypergraphiques	Choisir explicitement le mode de requête et la portée contextuelle ; tester les correspondances attendues et les contraintes rejetées.
Objets et actions du runtime	Contrôler l'identité, la propriété des objets, les autorisations et les frontières transactionnelles de chaque opération à effets.
Bindings Python et C#	Utiliser les bindings générés pour la version cible ; respecter la propriété des handles, leur libération et la durée de vie des callbacks.
Windows, Linux, macOS et conteneurs	Vérifier le package runtime, les dépendances natives et les outils nécessaires pour le système et l'architecture réellement utilisés.
Publication MCP et OpenAPI	Valider les descripteurs générés ; configurer l'authentification et les permissions des appels entrants indépendamment de la publication des points d'accès.
Sessions d'agents et requêtes de connaissances	Borner les objectifs, fournir un contexte explicite, examiner les demandes en attente et confirmer les effets sensibles.
Métadonnées compilées et compilation incrémentale	Distinguer le stockage natif, le graphe sémantique, le cache d'images compilées et le cache de signatures ; déclencher une compilation complète lorsque l'outil signale un changement non pris en charge.
Clients riches et synchronisation distribuée	Définir le fournisseur, les règles de fusion et de résolution des conflits, l'idempotence et les reprises hors ligne ; ne pas présumer une fusion automatiquement exempte de conflits.

80.2 Conditions préalables à l'exploitation

— Aligner le runtime, la CLI, les bindings, les descripteurs générés et le schéma du modèle sur une même version prise en charge.

— Vérifier l'intégration sur chaque combinaison réelle de plateforme et de fournisseur.
— Pour les actions publiées modifiant l'état, tester en préproduction les demandes autorisées, refusées, expirées ou répétées. Vérifier l'autorisation et la confirmation sur le chemin *entrant* réellement utilisé.
— Ne pas supposer qu'une transaction distribuée englobe automatiquement les ressources SQL et l'hypergraphe sémantique ; définir les compensations et la reprise lorsque plusieurs ressources participent.
— Mesurer les performances sur les données, la configuration de compilation et le matériel de l'application, plutôt que de reprendre des chiffres obtenus dans un autre environnement.

80.3 Fonctionnalités facultatives ou en évolution

Certains parcours, notamment le débogage source dans YAML, le chargement d'extensions natives et le partage avancé de connaissances, dépendent d'un adaptateur, d'une plateforme ou d'un composant facultatif. Consultez les fonctionnalités et prérequis de la distribution cible avant de les inscrire dans une procédure d'installation ou un engagement client. Une feuille de route ne constitue pas un contrat d'exécution.

80.4 Cohérence de la documentation

Le Guide du développeur et les ouvrages associés partagent les mêmes principes de compatibilité. Décrivez ce que l'utilisateur peut configurer et vérifier, sans déduire une certification globale d'un seul sous-système ; conservez les éléments d'ingénierie propres à une version dans les dossiers de validation internes plutôt que dans un tutoriel public.

Chapitre 81

Ce que change la programmation conceptuelle

La programmation conceptuelle déplace le centre de gravité du développement applicatif. Elle ne supprime ni le code, ni le stockage, ni les protocoles, ni l'ingénierie du déploiement : elle redéfinit l'artefact qui détient le sens métier mis en œuvre par ces mécanismes.

81.1 Du code au sens exécutable

Plutôt que de partir des contrôleurs, des tables et des routes, logiCells commence par les concepts, rôles, relations, actions, événements et processus. Le code demeure, mais il réalise un sens déclaré et respecte les contrats du runtime.

81.2 Des écrans aux projections

Un écran n'est pas l'application. C'est une projection d'un concept dans un contexte d'interaction. L'architecture View de logiCells constitue une possibilité de réalisation ; les clients CLI, API/services et MVP ou sur mesure restent des alternatives valides.

81.3 Des API aux capacités

Un point d'entrée de service doit exposer une capacité publiée du modèle, et non devenir un deuxième endroit où inventer la sémantique métier. OpenAPI, RPC, ABI, MCP et les autres protocoles peuvent modifier la frontière d'accès sans changer le sens conceptuel de l'action.

81.4 Des prompts d'IA aux propositions gouvernées

L'IA devient plus sûre lorsqu'elle travaille sur des concepts structurés et des actions publiées plutôt que sur un texte illimité ou des tables brutes. Dans le framework actuel, une proposition H-Logic peut porter une origine et des éléments de provenance, puis suivre un cycle de validation où l'analyse syntaxique et la validation sémantique restent distinctes de l'activation. Une proposition générée ne devient donc pas une connaissance au seul motif que le parseur l'accepte.

81.5 D'une capacité « prise en charge » à un statut étayé par des preuves

Une description mature de plateforme doit distinguer disponibilité, vérification et intention architecturale. La matrice commune de capacités dans Chapitre 80, "Fonctionnalités du runtime et liste de vérification" précise ces frontières. La disponibilité d'un runtime sur un système d'exploitation ne signifie pas, par exemple, que chaque extension native ou chargeur ABI y ait été validé.

81.6 Ce qui relève toujours de l'ingénierie classique

La programmation conceptuelle ne supprime pas :
— les mesures de performance et la planification de capacité ;
— la gestion du cycle de vie, de la propriété des objets et de l'arrêt déterministe ;
— les opérations de base de données et de réseau ;
— la configuration de sécurité et le durcissement des déploiements ;
— les dépendances natives propres aux plateformes ;
— les tests du comportement métier au-delà de la validation de la syntaxe déclarative.
Le bénéfice réside dans le fait que ces préoccupations entourent un contrat conceptuel explicite, au lieu de devenir implicitement le contrat lui-même.

81.7 Synthèse

Le changement essentiel est simple : le sens reste explicite et exécutable, tandis que les interfaces, liaisons et choix de déploiement demeurent des projections remplaçables.

Chapitre 82

Organisation d'un projet logiCells

82.1 Périmètre

— applications ou configurations ;
— packages ;
— modèles ;
— vues ;
— services ;
— ressources ;
— modèles de bases de données ;
— journaux, caches, synchronisation et instantanés.

82.2 Arborescence publique recommandée

Un projet logiCells doit rendre visible la séparation des modèles, vues, services et artefacts de déploiement. Les noms exacts des répertoires peuvent varier, mais l'organisation doit permettre de distinguer les fichiers qui définissent les concepts, les projections, les services d'intégration et l'assemblage de l'application runtime.

```
meta/models/             declarative concepts, relations, actions and processes
meta/views/contracts/    channel-independent interaction contracts
meta/views/models/       view-owned MODEL redefinitions and overlays
services/                service groups, external channels and publication settings
packages/                package descriptors used by the runtime loader
resources/               icons, translations, styles and visual assets
deploy/                  generated release trees, update descriptors and installers
```

Listing 82.1 – Responsabilités typiques des dossiers

L'essentiel n'est pas le nom du dossier, mais l'indépendance entre les concepts exécutables, l'apparence visuelle et le déploiement. Un même modèle peut ainsi être réutilisé dans une application de bureau, une vue web, un processus agentique ou un contexte de publication de services.

82.3 Ce qui appartient à la couche MODEL

La couche de modèle contient la structure conceptuelle stable : identifiants de classes, rôles, facettes, actions, événements, collections et contraintes. Elle ne doit pas dépendre d'une boîte à outils graphique particulière. Un modèle peut référencer une vue par son nom, mais ne doit pas faire de son apparence une sémantique métier.

82.4 Ce qui appartient à la couche VIEW

La couche VIEW décrit des contrats d'interaction plutôt qu'une technologie d'écran concrète. Un contrat peut déclarer des curseurs, jeux de données, composants fonctionnels, événements et mécanismes de navigation, liaisons de composants ainsi que des références MODEL-behind. Le participant peut être une personne ou un agent. Les moteurs de rendu natifs et web matérialisent les contrôles, tandis qu'un runtime d'agent peut employer le même contrat pour présenter l'information, demander les valeurs manquantes et invoquer les capacités déclarées sans afficher de formulaire.

L'arborescence canonique générée comporte `contracts` et `models` directement sous `meta/views` ; les anciens miroirs de dossiers de vues ne doivent pas être recréés à côté. Les manifestes locaux utilisent `_.manifest.yaml`.

Chapitre 83

Aide-mémoire de modélisation conceptuelle

83.1 Notions abordées

— concept ;
— rôle ;
— facette ;
— contrainte ;
— collection ;
— action ;
— événement ;
— vue ;
— processus ;
— capacité publiée.

83.2 Liste de contrôle du modèle conceptuel

Commencez par nommer le concept, puis déclarez les rôles qui lui donnent un sens opérationnel. Ajoutez ensuite les facettes, actions, événements, collections et vues.

```
1. Donner au concept un classId stable.
2. Déclarer ses rôles dans une facette hypergraphique.
3. Déclarer les projections persistantes avec rolePath si nécessaire.
4. Ajouter une action lorsqu'elle exprime une capacité significative.
5. Publier une action seulement si elle constitue un contrat public.
6. Employer les événements pour signaler un changement, non pour masquer l'état métier.
7. Utiliser un modèle de processus pour coordonner plusieurs objets.
8. Traiter les vues comme des projections, non comme des responsables de la logique métier.
```

Listing 83.1 – Liste de contrôle d'un modèle conceptuel

83.3 Liste de contrôle d'un nouveau module déclaratif

Lors de la création d'un module, au-delà de la modification d'un modèle déjà chargé, vérifiez l'ensemble du chemin d'assemblage, et non le seul fichier de modèle.

```
1. Choisir des classId stables appartenant au projet.
2. Créer le descripteur de package.
3. Pointer schemaDirectories/schemaPath vers la racine des modèles.
4. Créer \texttt{\_.manifest.yaml} et déclarer chaque modèle chargeable.
5. Créer les modèles déclaratifs.
6. Pour les entités sémantiques, déclarer explicitement le concept Entity.
```

```
7. Ne pas ajouter de champ Me littéral ; le rôle Me des entités est implicite dans la sémantique.
8. Déclarer Actions, Events et Processes comme des capacités du modèle.
9. Créer un package de tests distinct et ses modèles de test.
10. Vérifier l'absence de doublons de classId dans l'application assemblée.
11. Démarrer ou compiler l'application depuis sa configuration .lgc.
12. Vérifier la compilation des modèles et l'initialisation sémantique dans les journaux.
```

Listing 83.2 – Liste de contrôle d'un nouveau module

La distinction cruciale oppose le fichier source au modèle d'application assemblé. Un fichier correct, mais inaccessible depuis la chaîne de chargement des packages et manifestes, ne fait pas partie de l'application en cours d'exécution.

83.4 Conseils concernant les espaces de noms

L'espace de noms `lgc#` désigne les concepts de plateforme et ceux du runtime destinés à être réutilisés. Un espace comme `lbs#` peut être réservé aux exemples applicatifs. La séparation aide à distinguer ce qui relève de la plateforme de ce qui appartient au domaine métier d'une application.

83.5 Signaux de mauvaise conception

Un modèle devient difficile à maintenir lorsque les scripts contiennent une orchestration implicite, lorsque les vues portent les règles métier, lorsque des champs de base de données remplacent les rôles conceptuels ou lorsque les processus n'existent que sous forme de gestionnaires de boutons. La solution consiste souvent à introduire un concept de processus, un concept de relation, une action publiée ou un événement explicite.

Chapitre 84

Aide-mémoire H-Logic

84.1 Définitions et instances

```
define .:#Person(me)[name::#string];
#alice:#Person;
_:#Person;
```

Listing 84.1 – Formes fondamentales

84.2 Rôles, attributs et ordre

```
define .:#worksFor(person:#Person, company:#Company, context:#Context);
define .:#Order(me)[amount::#float, status:#OrderStatus];
define .:#Evidence(item:#Item, context^2:#Context);
```

Listing 84.2 – Positions structurelles et descriptives

84.3 Interfaces et termes nommés

```
doc42::#Auditable;
projectBeta := #projectBeta;
exists target=&projectBeta on .:#VisibleIn(label42, target);
```

Listing 84.3 – Rattachement et liaison à un terme nommé

84.4 Contraintes

```
define .:#Required(x:#A not null);
define .:#Defaulted(x:#A default &a);
define .:#Fixed(x:#A=&a);
```

Listing 84.4 – Contraintes locales sur les termes

84.5 Relations logiques

```
#A ->_[weight=1, context=#c] #B;
#A <->_[context=#shared] #B;
#Diagnosis :-_[confidence=0.8] #Pattern;
#Candidate !-_[reason=#invalid] #Bottom;
```

Listing 84.5 – Opérateurs logiques et compléments

84.6 Quantificateurs et requêtes

```
any x:#A, exists y:#B on .:#Pair(x, y);
every y:#B on .:#R(y);
ask .:#A(x) and .:#R(x, y) return x;
? .:#Person(Tom)[name='Tom Hanks'] return Tom;
```

Listing 84.6 – Quantification et formes de requête

84.7 Modes de requête et portée

```
ask homomorphic subtypes instances .:#Document(d)
  return d;

ask unifying (.:#Pair(x, #b), .:#Pair(#a, y))
  return (x, y);

ask resolving .:#reachable(#a, target)
  return target;
```

Listing 84.7 – Mode de raisonnement et options de portée

Mode ou option	Signification
`homomorphic`	Correspondance de motifs structurels persistés ; mode par défaut.
`unifying`	Unification canonique de plusieurs opérandes compatibles.
`resolving`	Résolution de buts au moyen de faits et de règles de type Horn.
`instances [only]`	Inclure les instances ou limiter les résultats à celles-ci.
`types [only]`	Inclure les types ou limiter les résultats à ceux-ci.
`subtypes [only]`	Parcourir le type demandé et ses descendants.
`supertypes [only]`	Parcourir le type demandé et ses ancêtres.
`masked`	Appliquer les restrictions actives de masquage ou de visibilité.

Forme des résultats. Zéro réponse produit `nil` ; une réponse unique est renvoyée directement ; plusieurs réponses constituent une liste conceptuelle chaînée. Un tuple ou une liste exprimés dans `return` ne sont pas enveloppés une seconde fois.

Valeurs héritées. Les instances descendantes peuvent être retournées par des vues adaptées. L'identité de l'instance réelle est préservée, tandis que les positions visibles suivent l'ordre de l'ancêtre demandé. Le mappage polyadique normal, et non une permutation réduite, fait autorité sémantiquement.

84.8 Définitions composites

```
define .:#ContextualRule(rule, context) as
  .:#Rule(rule)
  and .:#Context(context)
  and .:#validIn(rule, context);
```

Listing 84.8 – Composition logique

84.9 Formes à éviter

— ne pas employer `&x` comme nom de rôle ;
— ne pas combiner typage et égalité dans `y:#A=&a` ;
— ne pas nommer les variables de requête `?x` ;
— ne pas encoder un contexte comme un appel de fonction non pris en charge ;
— ne pas employer un type structurellement indéfini dans une application ou un rattachement d'interface ;
— ne pas exposer de rôles publics non contraints dans une définition projetée ;
— ne pas interpréter un joker comme une contrainte exactement égale à `nil` ;
— ne pas supposer qu'une instance de sous-type présente le même ordre de positions que son parent ;
— ne pas ajouter une liste artificielle autour du résultat unique d'une requête.

Chapitre 85

Motifs H-Logic vérifiés par les tests

Les exemples de cette annexe sont alignés sur la grammaire H-Logic 3.4 et les catégories de tests actuelles du parseur et de l'évaluateur. Leur concision est volontaire : chaque motif illustre une fonctionnalité stable du langage.

85.1 Valeurs primitives et collections

```
#an_apple;
1;
'abc';
1.75;
true;
();
(x,);
(x, y);
[a, b];
{#a, #b};
```

Listing 85.1 – Expressions primitives et collections

85.2 Définitions, rôles et attributs

```
define .:#P0;
define .:#Person(me)[name::#string];
define .:#worksFor(person:#Person, company:#Company, context:#Context);
define .:#Transfer(me)[sender:#Person, receiver:#Person, amount::#float];
```

Listing 85.2 – Définitions de types

85.3 Instances nommées et anonymes

```
define .:#Person(me)[name::#string];
#alice:#Person;
_:#Person;
.:#Person(Tom)[name='Tom Hanks'];
```

Listing 85.3 – Modèles d'instances

85.4 Rattachement d'interface

```
define .:#Customer(me)[name::#string];
#alice::#Customer;
_:#Customer(#alice);
```

Listing 85.4 – Encapsulation et rattachement

85.5 Quantificateurs

```
exists y:#B on .:#R(y);
every y:#B on .:#R(y);
any x:#A, exists y on .:#Pair(x, y);
any x:#A, every y:#B on .:#Pair(x, y);
any x:#A, exists y:#B, exists z:#C on .:#R(x, y, z);
```

Listing 85.5 – Motifs quantifiés

85.6 Relations logiques et compléments

```
.:#a(x) -> .:#b(x);
#A ->_[weight=1, context=#c] #B;
#Diagnosis :-_[weight=0.8, context=#clinical] #SymptomPattern;
#Candidate !-_[context=#validation] #Bottom;
```

Listing 85.6 – Motifs de relations logiques

85.7 Définitions composites et projetées

```
define .:#f(x, y) as .:#a(x) and .:#c(x, y);
define .:#fExists(x, y) as
  .:#u(x) and exists y:#B and .:#v(x, y);

define .:#PersistedU(me, x);
define .:#PersistedV(me_, x, y);
define .:#ProjectedF(x, y) as
  .:#PersistedU(me, x)
  and .:#PersistedV(me_, x, y);
```

Listing 85.7 – Motifs de définitions composites

85.8 Commandes et requêtes

```
projectBeta:#Context;
tell projectBeta:#Context;
do projectBeta:#Context;
ask .:#A(x) and .:#R(x, y) return x;
```

Listing 85.8 – Motifs de commandes

85.9 Catégories de tests négatifs

La suite de tests rejette également des formes invalides dont le contrat a été précisé. La documentation publique doit respecter ces limites :

— les variables de requête préfixées, comme `?x` ;

— les termes nommés employés explicitement comme noms de rôles ;
— les appels de fonction de contexte non pris en charge ;
— l'utilisation structurelle de types encore indéfinis ;
— les rôles publics projetés absents du corps composite.

85.10 Règle documentaire

Avant d'ajouter un exemple H-Logic au guide, vérifiez-le avec la grammaire actuelle et, si possible, associez-le à un test du parseur ou de l'évaluateur. La documentation H-Logic devient alors une spécification exécutable, et non un simple pseudo-code d'illustration.

Chapitre 86

Modèles d'actions déclaratives

86.1 Périmètre

— action d'objet ;
— action de classe ;
— action publiée ;
— action exposée par un service ;
— action de traitement d'événement ;
— action de transition de processus.

86.2 Actions d'objet

Une action d'objet appartient à une instance. Elle convient lorsque la capacité dépend de l'état d'un objet conceptuel particulier. Exemples courants : `Start`, `Validate`, `Finalize`, `Retry` et `Open`.

```
actions:
- action:
    name: Validate
    type: ObjectItem
    published: true
    params:
    - Result:
        dataType: Boolean
        paramType: ptReturn
```

Listing 86.1 – Modèle de déclaration d'une action d'objet

86.3 Actions de classe

Une action de classe appartient au concept lui-même. Elle convient aux fabriques, aux recherches, aux points d'entrée publiés et aux opérations métier qui ne dépendent pas d'une instance déjà sélectionnée.

86.4 Actions publiées

Une action publiée est un contrat. Elle peut être appelée par une vue, un agent, un autre processus ou une couche de publication de service web. La déclaration de ses paramètres fait donc partie de l'interface publique du modèle conceptuel.

Chapitre 87

Modèles d'événements conceptuels

87.1 Périmètre

— événement métier ;
— événement visuel ;
— événement de processus ;
— événement de messagerie ;
— événement local ;
— événement distant.

87.2 Un événement signale un changement observable

Un événement doit signaler qu'un fait significatif s'est produit. Il ne doit pas remplacer implicitement l'état métier. Son nom, sa source, ses paramètres et les actions qui lui sont associées constituent un contrat déclaratif d'acheminement.

```
events:
  - lbs.Process.Completed:
      actions:
        - action:
            path: ./Action[@name='OnCompleted']
```

Listing 87.1 – Association d'un événement à une action

87.3 Événements locaux, visuels et distants

Un événement visuel coordonne une vue ou un composant ; un événement métier coordonne des objets conceptuels ; un événement distant franchit une frontière de processus, d'agent ou de messagerie. Séparer ces catégories évite de coupler l'interface utilisateur au processus métier.

87.4 Événements de processus

Un modèle de processus doit émettre des événements tels que `Started`, `Validated`, `Completed`, `Rejected` ou `Failed`. Ils rendent l'orchestration observable et permettent aux contrôleurs, aux agents et aux services de réagir sans encapsuler tout le déroulement dans un script procédural unique.

Chapitre 88

Modèles de processus métier

88.1 Périmètre

— processus synchrone ;
— processus asynchrone ;
— processus soumis à validation humaine ;
— processus piloté par messages ;
— processus s'appuyant sur un service.

88.2 Quand créer un modèle de processus ?

Créez un modèle de processus lorsqu'une séquence mobilise plusieurs objets métier, lorsque son état doit être observable, lorsqu'il faut gérer les échecs ou lorsqu'un utilisateur, un service ou un agent doit pouvoir reprendre le travail ultérieurement.

88.3 Processus synchrone

Un processus synchrone enchaîne des actions exécutées par une même action de processus. Ce choix convient à un traitement court, déterministe et réalisable dans un seul contexte d'exécution.

88.4 Processus asynchrone

Un processus asynchrone émet des événements ou des messages, puis attend les réponses des participants. Il convient aux coordinations impliquant une interaction humaine, des appels de services, des agents, des files de messages ou des activités de longue durée.

88.5 Anti-modèle

Ne dissimulez pas un processus impliquant plusieurs objets derrière un bouton de vue. Le bouton peut lancer le processus ; c'est au modèle de processus de porter l'orchestration.

Chapitre 89

Modèles d'assistants d'intelligence artificielle

89.1 Périmètre

— explication d'un modèle ;
— synthèse d'un processus ;
— appel d'un service ;
— génération de tests ;
— interprétation d'une trace d'exécution.

89.2 Un assistant fondé sur les concepts

Un assistant doit agir sur les concepts, les actions, les règles et les vues, plutôt que manipuler des écrans dont la structure n'est pas explicitée. Le modèle lui fournit des noms stables, des rôles typés et des capacités exécutables.

89.3 Lire, expliquer, agir

Un modèle d'assistant utile comporte trois niveaux. Il consulte d'abord l'état conceptuel au moyen des requêtes et des vues. Il explique ensuite cet état avec le vocabulaire du modèle. Enfin, il n'agit que par l'intermédiaire d'actions déclarées, de préférence publiées et dotées de paramètres explicites.

89.4 Garde-fous

L'assistant ne doit pas contourner le runtime conceptuel. L'autorisation, la validation, l'état des processus et la traçabilité restent régis par le modèle. C'est à cette condition que l'intégration de l'IA demeure compatible avec les exigences de gouvernance.

Chapitre 90

Recettes de programmation du runtime

90.1 Périmètre

— créer un objet ;
— retrouver un concept de classe ;
— générer un fournisseur ;
— ouvrir un tampon ;
— enregistrer les données ;
— construire un événement ;
— émettre un événement.

90.2 Créer un objet

Utilisez le conteneur du runtime pour créer un objet à partir d'un concept déclaré. Renseignez les rôles prévus dans le modèle, puis persistez l'objet au moyen du mécanisme de persistance habituel.

```
Item := ObjectsContainer.CreateObjectItem('lbs#work_item', 1, '');
Item.Name := 'Example';
Item.PostData();
```

Listing 90.1 – Création d'un objet du runtime

90.3 Rechercher un concept

Résolvez un `ClassItem` à partir de son identifiant de classe lorsque le script doit passer par le modèle conceptuel plutôt que par une classe d'implémentation codée en dur.

90.4 Exécuter un processus

Créez l'objet de processus, associez-lui les objets métier participants, puis appelez son action de démarrage publiée. Le processus prend en charge les événements, changements d'état et traces d'exécution ultérieurs.

Chapitre 91

Éléments essentiels du langage functionalP

Cette annexe présente les fonctionnalités du langage sous-jacentes au sous-ensemble pratique introduit dans Chapitre 37, “Premiers pas avec functionalP dans logiCells”. Elle est volontairement centrée sur le langage. Les nouveaux développeurs logiCells gagneront à apprendre d’abord les modèles de métadonnées, puis à revenir ici lorsqu’ils devront comprendre un script utilisant des listes, fonctions, lambdas ou structures de contrôle avancées.

91.1 Objectif

`functionalP`, souvent abrégé FPL dans la documentation du runtime, est un langage compact permettant d’exprimer des calculs, des corps d’actions, des fragments de processus et des opérations sur les graphes. Il ne remplace pas le modèle déclaratif : il apporte une couche programmable stockable dans les métadonnées, exécutable dynamiquement et reliée aux conteneurs, actions, services et structures conceptuelles.

La syntaxe résumée ici provient de la grammaire actuelle et des tests stabilisés. La cartographie détaillée des sources reste disponible dans le dépôt pour les mainteneurs ; les exemples ont été adaptés pour faciliter l’apprentissage.

91.2 Structure d’un programme

Un programme peut comprendre un en-tête facultatif de bibliothèque et d’import, suivi d’une séquence de termes.

```
1 library Runtime.Demo;
2 import Runtime.Core;
3
4 Result := 1 + 2;
```

Listing 91.1 – En-tête de bibliothèque et d’import

La plupart des scripts de métadonnées logiCells n’ont pas besoin de cet en-tête :

```
1 5 + 10;
```

Listing 91.2 – Expression minimale

91.3 Séquences

Un script s'écrit généralement comme une suite d'expressions séparées par des points-virgules.

```
t1 := 5;
t2 := 8;
t1 - t2;
```

Listing 91.3 – Séquence d'affectations suivie d'une expression

Les expressions intermédiaires peuvent modifier l'environnement d'exécution. L'expression finale peut devenir la valeur du script. Lorsqu'une valeur de retour stable est importante, affectez explicitement `Result`.

91.4 Valeurs primitives

Le langage accepte les constantes primitives usuelles :

```
'Hello world!';
42;
42.65;
true;
false;
nil;
```

Listing 91.4 – Valeurs primitives

Les tests du runtime peuvent formater les booléens sous la forme `T` ou `F`. Dans les scripts applicatifs, utilisez des valeurs booléennes et non les chaînes issues de leur formatage.

91.5 Affectations et variables

L'opérateur d'affectation est `:=`.

```
a := 'Bonjour';
a;
```

Listing 91.5 – Affectation et lecture

Les affectations servent fréquemment à nommer des datasets, objets, événements, classes et valeurs intermédiaires.

91.6 Arithmétique, comparaisons et logique booléenne

La grammaire prend en charge les opérateurs arithmétiques et de comparaison courants, ainsi que la composition de conditions booléennes.

```
5 + 10 * 2 + 6;
5 <> 10;
5 <= 5;
true and false;
false or true and false;
```

Listing 91.6 – Expressions arithmétiques et booléennes

`functionalP` est analysé comme un langage d'expressions, et non comme un langage de macros textuelles.

91.7 Listes et intervalles

Les listes contiennent des expressions séparées par des virgules. La liste vide s'écrit `()`.

```
(1, 2, 3);
();
(1, 2, (5, 4, 6));
1 .. 5;
```

Listing 91.7 – Listes et intervalles

Le langage propose également des opérateurs sur les listes, notamment `head`, `tail`, `last`, `length`, `reverse`, `drop` et `take`.

```
tail (1, 2, 3);
length (1, 2, 3, 4);
reverse (1, 2, 3);
```

Listing 91.8 – Opérateurs de listes

91.8 Structures de contrôle

Le langage prend en charge les conditions et les boucles.

```
a := 0;
if (a = 0) then
begin
  b := 1;
end
else
begin
  b := 2;
end;
b;
```

Listing 91.9 – Expression conditionnelle

```
res := 0;
for value in (1, 2, 3, 4) do
begin
  res := res + value;
end;
res;
```

Listing 91.10 – Boucle for sur une liste

Limitez les boucles dans les scripts de métadonnées. Lorsqu'un script devient complexe, envisagez de déclarer une action ou un processus, ou de recourir à une implémentation native.

91.9 Fonctions

Une fonction se déclare avec `function`, un nom, une liste de paramètres et un corps.

```
function double(a)
begin
  a * 2;
end;

double(4);
```

Listing 91.11 – Fonction simple

Les fonctions peuvent être combinées avec des opérateurs fonctionnels :

```
function double(a)
begin
  a * 2;
end;

map double (1 .. 5);
```

Listing 91.12 – Application d'une fonction à un intervalle

91.10 Application partielle et lambdas

Certains tests couvrent l'application partielle et les expressions lambda. Considérez-les comme des fonctions avancées dans les métadonnées applicatives.

```
(lambda(x) x * x) 4;
```

Listing 91.13 – Expression lambda

Privilégiez les actions nommées explicites pour les comportements métier. Les lambdas conviennent aux transformations locales, mais pas à la dissimulation de règles durables du domaine.

91.11 Expressions case

Les expressions `case` utilisent des clauses proches des motifs, introduites par `:-` et éventuellement séparées par `|`.

```
function normalize(a)
begin
  case a of
    () :- ()
  | _  :- a
  end;
end;

map normalize (((1,2)), (), ((3,2)));
```

Listing 91.14 – Expression case avec valeur de repli pour la liste vide

Le caractère de soulignement est un motif générique de type joker.

91.12 Expressions pointées et objets du runtime

`functionalP` peut appeler les méthodes et propriétés des objets exposés dans l'environnement d'exécution.

```
Logger.WriteLog('functionalP script executed', 1);
```

Listing 91.15 – Appel d'une méthode du runtime

Un script peut également accéder aux objets injectés :

```
aInstance.Name := 'RuntimeObjectName';
aInstance.Name;
```

Listing 91.16 – Lecture et écriture d'une propriété d'objet du runtime

Dans les métadonnées logiCells, ces expressions servent le plus souvent à accéder à `Self`, `Sender`, `ObjectsContainer`, `PageControler` et `DataSetsManager`.

91.13 Littéraux d'hypergraphe et constructions de processus

La grammaire comprend des constructions orientées graphes et processus. Elles sont importantes pour le runtime, mais relèvent d'un niveau avancé dans un guide d'initiation.

Pour du code de production, utilisez des déclarations H-Logic testées, des déclarations de modèles ou les API du runtime, sauf si la forme littérale est explicitement documentée pour le module employé.

91.14 Erreurs d'interprétation fréquentes

Idée fausse	**Rectification**
`functionalP` est le langage du modèle	C'est une couche de scripts exécutables utilisée au sein des métadonnées YAML.
Il suit les mêmes règles qu'Object Pascal	Sa syntaxe emprunte quelques éléments à Pascal, mais c'est un langage d'expressions de runtime doté de sa propre grammaire et de son propre modèle d'évaluation.
Chaque test du runtime est un exemple de production	Les tests démontrent des fonctions prises en charge et des cas limites. Il faut adapter les exemples avant de les employer dans une initiation ou une documentation de production.
Le formatage des booléens fait partie du contrat métier	Utilisez des booléens typés ; ne dépendez pas d'un formatage de test comme `T` ou `F`.
L'expression finale est toujours évidente	Affectez explicitement `Result` lorsque le contrat de l'action prévoit une valeur de retour.

91.15 Ordre d'apprentissage recommandé

1. les expressions simples ;
2. les affectations et `Result` ;
3. `Self`, `Sender` et les objets injectés ;
4. les scripts de datasets et du contrôleur de pages ;
5. les fonctions et les listes ;
6. les lambdas et les expressions `case` ;
7. les constructions de processus, une fois le modèle des expressions compris.

Chapitre 92

Glossaire de la programmation conceptuelle

92.1 Périmètre

— Concept : unité de sens nommée ;
— Facette : projection opérationnelle d'un concept ;
— Rôle : position typée au sein d'un concept ou d'une relation ;
— Action : intention exécutable ;
— Événement : changement observable ;
— Processus : objet de coordination ;
— Vue : projection contextuelle ;
— Composant fonctionnel : fonction d'interaction déclarée ;
— Action publiée : capacité accessible depuis l'extérieur.

92.2 Notions essentielles

Concept. Unité de sens déclarée pouvant porter des rôles, des contraintes, des facettes, des actions, des événements et des vues.

Mémoire conceptuelle. Couche de graphe ou d'hypergraphe dans laquelle les structures conceptuelles peuvent être représentées, interrogées et évaluées.

Facette. Projection d'un concept consacrée à une préoccupation particulière, par exemple sa structure hypergraphique ou sa persistance en base de données.

Action publiée. Action déclarée et rendue accessible de l'extérieur en tant que capacité. Elle peut être utilisée par des vues, des processus, des agents ou des services web.

Composant fonctionnel. Composant de vue déclaratif qui nomme une fonction d'interface utilisateur indépendamment de toute implémentation concrète.

Modèle de processus. Modèle destiné à coordonner plusieurs objets, actions, événements ou participants.

Plateforme de messagerie. Mécanisme de plateforme réutilisable pour la coordination asynchrone par messages.

Chapitre 93

Guide de diagnostic

93.1 Points à vérifier

— les identifiants de classe ;
— les chemins de rôles ;
— les filtres de fournisseurs ;
— les paramètres d'actions ;
— les abonnements aux événements ;
— les chemins de vues ;
— la publication des services ;
— la parité des représentations YAML.

93.2 Une vue n'affiche pas les données attendues

Vérifiez d'abord le curseur, puis le jeu de données et enfin le chemin de collection. Assurez-vous que les chemins de rôles utilisés par la collection correspondent aux rôles déclarés dans le modèle.

93.3 Une action n'est pas accessible depuis l'extérieur

Vérifiez que l'action est publiée, que les types de ses paramètres sont déclarés et que la couche de publication est autorisée à exposer le concept qui la porte.

93.4 Un processus est difficile à déboguer

Recherchez une orchestration implicite dans les scripts de vues ou les actions d'objets. Déplacez la coordination entre objets vers un modèle de processus et émettez des événements explicites à chaque changement d'état majeur.

93.5 Un composant dépend trop d'une plateforme

Remplacez l'hypothèse d'un composant concret par la déclaration d'un composant fonctionnel. Enregistrez séparément son implémentation pour chaque plateforme cible.