UNIVERSITÉ DU QUÉBEC À MONTRÉAL

VERS L'UTILISATION DE DSL ET DE LANGAGES DYNAMIQUES EN ENTREPRISE. UNE ÉTUDE DE CAS AVEC GROOVY.

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

FRANÇOIS-XAVIER GUILLEMETTE

FÉVRIER 2009

Pour Jeanne et Annik. Merci pour votre patience, vos encouragements et votre amour, sans quoi, rien de tout ceci n'aurait été possible ou même imaginable...

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ACRONYMES

ANTLR	ANother Tool for Language Recognition
AOP	Aspect-Oriented Programmaing
API	Application Programming Interface
CRUD	Create Read Update Delete
CSS	Cascading StyleSheets
DOM	Document Object Model
DSL	Domain-Specific Language
HTML	HyperText Markup Language
HTTP	HyperText Transport Protocol
IDE	Integrated Development Environment
JDBC	Java DataBase Connectivity
JDK	Java Development Kit
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
KLOC	1000 Lines Of Code
LOP	Language-Oriented Programming
MOP	Meta Object Protocol
MRI	Matz's Ruby Interpreter
MVC	Model-View-Controller
ORM	Object-Relational Mapping
PCML	Program Call Markup Language
PF	Point de Fonction
RACC	Ruby yACC
REST	REpresentational State Transfer
RPC	Remote Procedure Call
RPG	Report Program Generator (obsolète)
SGBD	Système de Gestion de Base de Données
SOAP	Simple Object Access Protocol (obsolète)
SQL	Structured Query Language
URL	Uniform Resource Locator
XML	eXtensible Markup Language
XSLT	XSL Transformations
YACC	Yet Another Compiler Compiler

RÉSUMÉ

« *We dissect nature along lines laid down by our native language.* »
(Whorf, Carroll et Chase, 1956)

« *The limits of my language mean the limits of my world.* »
(Wittgenstein, 1981)

Tout comme l'indique Whorf, nous analysons le monde à l'intérieur du cadre sémantique de notre langue maternelle. Il en va de même pour la résolution de problèmes à l'aide de langages de programmation. Les possibilités offertes par un langage et sa syntaxe façonnent notre conception du domaine du problème.

Ainsi, si la langue forme notre pensée, les mots qui la composent ne sont parfois pas suffisants pour exprimer toute la richesse de nos idées. Par exemple, les spécialistes de domaines particuliers utilisent souvent un jargon. Pour s'en convaincre, il suffit d'écouter la conversation de deux professionnels d'un domaine qui nous est inconnu. Nous constatons qu'ils ne parlent pas tout à fait la même langue que nous. Ils s'échangent des idées et discutent de concepts en utilisant des mots et des expressions qui semblent étranges. Ainsi, les experts d'un même domaine développent leur propre jargon i.e. un langage spécifique à leur domaine d'affaire. Cette nouvelle langue leur permet de communiquer leur pensée de façon rapide et précise sans s'encombrer des détails et des lacunes de la langue « hôte ».

Dans le présent travail, nous posons un regard sur les constructions linguistiques permettant la remise en valeur de la construction de langages spécifiques à un domaine d'affaire (DSL) : système de types dynamiques, métaprogrammation et éléments syntaxiques divers. Nous réalisons un survol des DSLs. Que sont-ils ? Quand, comment et pourquoi les construisons-nous ? Nous voyons aussi quelques patrons de conception de DSLs. Finalement, nous appliquons la théorie exposée à un problème réel. Nous montrons les avantages apportés par l'élaboration et l'utilisation de DSLs dans le cadre du développement de logiciels.

Mots-clés : Langages dynamiques, DSL, Métaprogrammation, Groovy.

INTRODUCTION

L'activité de programmation est difficile. Le programmeur doit constamment jongler avec des concepts abstraits, des règles d'affaires et des considérations multiples. S'il n'en était que de la difficulté inhérente à l'activité de programmation, le travail du développeur serait compliqué mais surmontable. Dans de telles circonstances, il serait possible de documenter avec précision les règles de construction ainsi que les différents types de programmes pouvant être construis. Imitant ainsi, les architectes de l'Antiquité. La réalité est pourtant tout autre...

Il semble tout d'abord illusoire de penser pouvoir rassembler les « dix commandements » du développement de logiciels i.e. un ensemble fini de lois et de dictas incarnant la vérité absolue en matière de programmation dont le respect assurerait la place de notre programme au paradis des logiciels. Les connaissances sont plutôt accumulées et documentées sous la forme de patrons et de « bonnes pratiques ». Un patron est découvert lorsque la même solution est appliquée à répétition pour une même famille de problèmes. L'application de patrons, d'heuristiques, à la résolution des problèmes en dit long sur le type de situations rencontrées lors d'efforts de développement de logiciels. Chaque programme est différent et ce, malgré l'utilisation de *frameworks* communs et partagés. La capacité de pouvoir développer un programme « X » n'assure en rien la réalisation d'un programme « Y ». Si les outils technologiques restent les mêmes, l'environnement humain et organisationnel change sans arrêt. Les besoins des clients évoluent et deviennent de plus en plus sophisitiqués, le marché fluctue, les règles d'affaires changent, etc. Il est donc impossible de penser qu'un seul outil technologique peut régler tous les problèmes. Le *framework* J2EE est un exemple typique d'une telle *über-solution*. Le slogan « une-taille-pour-tous » implique que tous seront mal habillés.

Les langages de programmation n'échappent pas à ce constat. Comment est-il possible

d'imaginer qu'un seul langage de programmation soit suffisant pour exprimer tout ce qui est possible et nécessaire à la réalisation d'un programme ? Dans une application web de petite envergure, on dénombre déjà plusieurs langages qui coexistent et remplissent différents rôles :

- La présentation web est écrite en HTML, en CSS et en JavaScript ;
- La couche métier est réalisée en Perl ou en PHP ;
- Les interactions avec la base de données sont codées en SQL.

À ceci, on doit ajouter une pléthore de fichiers de configuration XML. S'ils possèdent tous la même syntaxe, ils sont définis à l'aide de grammaires différentes et donc, sont écrits en langages différents. Il ne faut pas oublier aussi tous les API utilisés. Un API est, en réalité, un langage spécialisé constitué de règles et de mots-clés au même titre que tous les langages énumérés précédemment. Ceci fait dire à certains que nous entrons dans une ère de « programmation polyglotte » où plusieurs langages sont utilisés en même temps (Ford, 2006).

En même temps que le besoin d'utiliser un ensemble de langages de programmation se fait sentir, la faiblesse des langages les plus courants apparaît soudainement. La montée en popularité récente des langages dynamiques est une réaction directe à la décision prise par l'industrie de restreindre les dommages collatéraux causés par des développeurs médiocres. Le langage Java, par exemple, est en partie bâti sur l'hypothèse qu'il est nécessaire d'empêcher les mauvais développeurs de faire « la mauvaise chose » (Ford, 2007). Il est donc impossible en Java, entre autres, de surcharger les opérateurs, d'hériter de plusieurs super classes, d'ajouter des méthodes à une classe lors de l'exécution ou d'intercepter des événements métaprogrammatiques à l'exécution. Les langages de programmation contraignants comme Java et VisualBasic sont au développement de logiciels ce que les ciseaux à bouts ronds sont au bricolage : un moyen d'empêcher les développeurs médiocres de se faire mal. En plus d'échouer sur ce point, les langages restrictifs limitent les développeurs chevronnés (Ford, 2007). C'est dans ce contexte que des langages plus puissants et expressifs que Java et C# bénéficient d'une popularité grandissante.

Grâce à des langages comme Ruby, Python et Groovy, les développeurs redécouvrent la métaprogrammation. Ces langages permettent, à différents degrés, de « programmer le programme ». En plus de pouvoir observer l'état des composantes internes d'une classe à l'exécution (introspection), il devient possible d'en modifier le comportement (intercession) et même d'intercepter des événements programmatiques grâce au protocole méta-objet (MOP). Ces « nouvelles » fonctionnalités permettent aux programmeurs de renouer avec une pratique que les adeptes du langage Lisp connaissent depuis toujours : la construction et l'utilisation de langages spécifiques à un domaine d'affaire spécifique (DSL).

Un DSL est un langage miniature conçu afin d'accomplir une tâche très précise. Par exemple, SQL est un langage ne servant qu'à interagir avec une base de données ; le langage servant à écrire des « Makefiles » est un DSL permettant de décrire la construction d'un programme. Un DSL s'oppose à un langage généraliste comme Java. La culture entourant les DSLs ne se limite par à leur utilisation. On cherche à construire des interfaces de plus hauts niveaux à travers l'élaboration de langages spécialisés. Les DSLs sont, en réalité, la continuation la plus élémentaire de ce que les programmeurs font de mieux lorsque confrontés à des interfaces compliquées : ils haussent le niveau d'abstraction afin d'en simplifier l'utilisation. La programmation orientée-langages (LOP) décrite dans (Ward, 1994) permet d'utiliser plusieurs DSL à différents niveaux d'abstraction afin d'en faire bénéficier les bienfaits à un nombre maximum d'intervenants. Le processus qualifié *d'outside-in* fonctionne en développant récursivement des DSLs de niveaux d'abstraction de plus en plus bas. On commence d'abord en élaborant un DSL qui servira à décrire l'application en entier et ce, à un niveau très élevé. Une partie de l'équipe sera responsable d'utiliser ce DSL pour développer l'application avec l'aide du client pendant qu'une autre partie implémentera le DSL. L'implémentation utilise elle-même des DSLs et ainsi, le processus recommencera récursivement. Un DSL de haut niveau permet une interaction plus productive avec les clients.

On dénote deux types de DSL : le DSL interne et le DSL externe. Le DSL interne est défini en exploitant la syntaxe d'un langage « hôte ». L'API d'une librairie peut donc

être considérée comme un DSL interne. Les capacités spéciales des langages dynamiques comme Lisp, Ruby, Groovy et Python sont particulièrement propices à la création de syntaxes simples et efficaces. Un DSL externe requiert des phases d'analyses lexicales, syntaxiques et sémantiques propres à la compilation « traditionnelle » (Appel et Palsberg, 2002). Ils requièrent l'utilisation de « compilateurs de compilateurs » tels que Antlr (Parr, 2007), SableCC (Gagnon et Hendren, 1998) ou JavaCC (Appel et Palsberg, 2002). Un langage XML peut être considéré comme un DSL externe.

Dans le présent travail, nous explorons l'élaboration et l'utilisation des DSLs dans le cadre du développement d'une famille d'applications web. Nous démontrons la simplicité de la réalisation de DSLs avec des langages dynamiques afin de résoudre des problèmes compliqués. En se basant sur le principe que le code source est plus souvent lu qu'il n'est écrit, nous montrons aussi que l'utilisation de DSLs internes améliore la lisibilité du code source produit et ainsi augmente la productivité des développeurs qui font l'implémentation initiale ainsi que la maintenance du programme. Pour ce faire, nous appliquons des solutions différentes à deux problèmes distincts. L'une utilisant une approche « classique » et l'autre, exploitant l'utilisation d'un DSL. Nous verrons aussi une des techniques d'élaboration de DSLs internes qui consiste à raffiner graduellement une interface programmatique.

Le chapitre un (page 6) se concentre sur les langages de programmation dynamiques. Nous y approfondissons les questions du typage statique et dynamique et le *duck typing*. Nous posons un regard sur la question de la métaprogrammation puis nous discutons de fonctionnalités implémentées par la plupart des langages dynamiques telles les fermetures et les modes d'évaluations. Le chapitre deux (page 28) explore le sujet des DSL. Nous y discutons des différents types de DSLs ainsi que des techniques menant à leur construction. Nous y faisons aussi un survol de quelques patrons de conception. Le chapitre trois (page 56) est une revue du domaine du problème. Nous y étudions le domaine d'affaire et les différents problèmes auxquels nous avons été confrontés lors du développement des applications qui ont mené à la construction des DSLs. Le chapitre quatre (page 72) montre comment nous avons réussi à résoudre les problèmes à l'aide

des DSLs. On y montre et explique le code source nécessaire à leur élaboration ainsi que leur utilisation.

CHAPITRE I

LANGAGES DYNAMIQUES

Dans ce premier chapitre, nous discutons des différences idéologiques entre les langages statiquement et dynamiquement typés. Nous tentons de répondre à la question «Qu'est-ce que le type d'un objet ?». Nous posons un regard sur les constructions de langages propres au langages dits «dynamiques» comme les fermetures et le protocole méta-objet. Finalement, nous verrons en quoi la montée des DSL coïncide avec celle des langages dynamiques et pourquoi les langages à typage statique sont en déclin.

* * *

L'évolution des technologies prend essentiellement la forme d'une spirale. Elle nous amène à reprendre les idées existantes et à les réutiliser d'une façon nouvelle et parfois inattendue. Le cycle est généralement caractérisé par une série de phases. L'état initial est un déclencheur technologique, une idée simple mais prometteuse. Au fil de la maturation du concept, il est enrichi et engraissé de fonctions additionnelles. Les limites du concept sont atteintes alors que celui-ci croule sous ses propres fonctionnalités. Devant cette situation insoutenable, un groupe d'individus se donne comme mission de révolutionner la technologie existante afin de lui redonner sa « pureté » originelle.

De nombreux exemples permettent d'illustrer ce cycle. Pensons à l'évolution d'XML à partir de SGML, de REST à partir des *WebServices* et de SOAP à partir de XML-RPC. Le cycle est toujours le même. Une technologie simple est adoptée puis enrichie jusqu'au point de corruption. Une nouvelle technologie naît des cendres de l'ancienne puis le cycle reprend.

Kuhn décrit l'évolution et l'avancement de la Science en une suite de changements de paradigmes, de révolutions (Khun, 1983). Les sciences « normales » sont le reflet des connaissances actuelles et forment un paradigme i.e. un schème de pensée. Les expériences réalisées sont structurées par le paradigme de la science « normale ». Plus d'expériences sont faites, plus d'anomalies sont détectées. Une anomalie est un fait qui ne peut être expliqué par le paradigme actuel. L'accumulation d'anomalies amène les scientifiques à concevoir une science « révolutionnaire ». Le changement de paradigme survient lorsque la science « révolutionnaire » devient « normale ». Le passage de la physique de Newton à celle d'Einstein est un changement de paradigme tout comme la transition du modèle planétaire ptoléméen au modèle copernicien. Le cycle décrit précédemment est essentiellement la version « informatique » des découvertes de Kuhn.

Le cycle « enrichissement/corruption/révolution » est aussi observable dans l'univers des langages de programmation. L'exemple typique est celui de Java qui a été conçu dans l'objectif de révolutionner C++ qui lui-même est le résultat de l'ajout d'une série de fonctionnalités au langage C. Plusieurs concepts ont été retirés au profit d'une plus grande « pureté » d'implémentation. L'héritage multiple et la surcharge des opérateurs en sont des exemples.

Les principaux langages de programmation utilisés en développement de logiciels aujourd'hui sont Java et C#. La richesse de leurs librairies ainsi que le sentiment de sécurité donné par le système typage statique font de ces langages le choix *de facto* lors d'une activité de développement.

Pourtant, des « anomalies » sont décelées et s'accumulent aux pieds des géants. La lourdeur du typage statique freine la flexibilité et l'expressivité des langages. Les *frameworks* de développement web peinent à donner une interface et une structure conviviale. L'absence de fermeture[1] se fait sentir en Java. Les techniques de programmation fonctionnelle sont souvent difficilement utilisables ou encombrantes. La métaprogrammation dynamique est absente de ces langages. La progression de la programmation orientée-

[1] en anglais : *closure*, *block* ou *lambda*

```
1 List<String> uneListe = new ArrayList<String>();
2 uneListe.add(12);            // => ERREUR !
3 uneListe = new Integer();    // => ERREUR !
4 uneListe = null;
5 uneListe = new LinkedList<String>();
```

Fig. 1.1 Typage statique avec Java 1.5

aspects en est le reflet. Finalement, le besoin de communiquer avec le client amène plusieurs développeurs à construire des langages de programmation spécifiques au domaine d'affaire. Java et C# sont cruellement déficients en ce qui concerne le nombre de structures de langage permettant la construction de DSLs[2].

Dans les sections qui suivent, nous explorons ces « anomalies révolutionnaires ».

1.1 *Duck Typing*

La confusion sur ce qu'est le « typage des données » impose l'ajout d'une discussion ici. Une grande incompréhension entoure les concepts de typage fort, faible, statique et dynamique. Commençons d'abord par quelques définitions.

Typage statique Un langage de programmation est dit « à typage statique » lorsque le type des objets est connu lors de la compilation (Pillai, 2004). Une fois défini et compilé, le type de l'objet ne pourra pas être modifié lors de l'exécution du programme. Ainsi, le type d'un objet est explicitement cité lors de sa définition. En Java, l'énoncé suivant permet de définir une liste de chaîne de caractères. Le type de l'objet « `uneListe` » sera pour toujours défini par le premier énoncé de la figure 1.1.

La variable `uneListe` est définie comme étant de type `List<String>`. L'implémentation choisie ne pourra varier qu'en restant à l'intérieur de la hiérarchie de types sous la classe `List<T>`. Ce comportement est voulu et souhaité par plusieurs développeurs. Le fait qu'une variable ne puisse changer « subitement » de type lors de l'exécution d'un programme caractérise le langage utilisé comme étant *type-safe*.

---

[2] *Domain-Specific Langages*

```
1 une_liste = []
2 une_liste << 12
3 une_liste = 0
4 une_liste = nil
5 une_liste = []
```

Fig. 1.2 Typage dynamique avec Ruby 1.8.6

Typage dynamique Le type des objets définis à l'aide d'un langage dit « à typage dynamique » est défini lors de l'exécution du programme (Pillai, 2004). Ce sont les méthodes publiques de l'objet qui déterminent son type. Un objet est donc défini non pas par ce qu'il *est* mais bien par ce qu'il *fait*. La figure 1.2 reprend l'exemple précédent.

Ici, le type de l'objet est évalué lorsqu'un message lui est envoyé. Dans l'exemple précédant, à la ligne 2, le message « << » est envoyé à l'objet `une_liste`. L'interpréteur Ruby s'enquit alors de savoir si `une_liste` peut répondre au message « << ». Les lignes 3 à 5 illustre le fait que l'affectation de types différents à un objet n'est pas une erreur.

Typage fort La force du typage est orthogonale à son dynamisme. La force et le dynamisme ont longtemps été confondus (Pillai, 2004). Cette perte de précision dans la catégorisation des systèmes de typage est principalement causée par le faible nombre de langages illustrant les cas atypiques. Le concept de typage fort est souvent associé à celui de typage statique. Il en va de même pour le typage dynamique qui est considéré comme étant synonyme avec typage faible. Même si la définition de ce qu'est un type varie d'un langage à l'autre, le respect du type de l'objet indique la force du typage.

Un langage est dit « à typage fort » si, lors de l'exécution, le type de l'objet est vérifié et respecté. Les langages à typage fort imposent généralement une conversion explicite des opérateurs de types différents (Aahz, 2003). En Ruby, la conversion d'un type numérique vers un autre doit être définie par la méthode `coerce`. La figure 1.3 illustre ce fonctionnement.

Typage faible Contrairement aux systèmes de typage fort, les langages à typage faible ne font pas respecter les types des objets lors d'une opération. C, Perl et JavaScript

```
1 ['a'] + ['b']  # => ["a", "b"]
2 ['a'] + 'b'    # => can't convert String into Array
3                      (TypeError)
```

Fig. 1.3 Mécanisme de typage fort en Ruby

```
1 $x = "10";       # => x est une chaîne ("10")
2 $x = $x + 2;     # => x est un nombre (12)
```

Fig. 1.4 Typage faible en Perl

sont de tels langages. En C, il est possible d'additionner un `char` et un `int` sans que le compilateur ne lève une erreur. Un langage peut donc être caractérisé à typage faible si le type d'un objet peut changer lors de l'envoi d'un message. La figure 1.4 montre comment le langage Perl ne fait pas respecter le type des objets (Aahz, 2003).

À la ligne 1, `x` contient une chaîne de caractères. L'opération d'addition à la ligne 2 transforme implicitement `x` en nombre entier puis y ajoute le nombre 2.

Le tableau 1.1, classifie les langages de programmation selon les axes « statique vs. dynamique » et « fort vs. faible ».

Le *Duck Typing* caractérise les langages de programmation qui offrent des mécanismes de typage dynamique et fort (Thomas et Hunt, 2001). La catégorie tire son nom de l'expression anglophone « *If it walks like a duck and talks like a duck, it's a duck.* » Tel que décrit précédemment, le type d'un objet n'est pas défini par ce qu'il est (*un canard*) mais par ce qu'il fait (*cancane et marche comme un canard*).

Un tel paradigme apporte une flexibilité accrue et une dimension nouvelle aux programmes. Voyons un premier exemple à la figure 1.5. Ici, la fonction `multiplier` n'a pas besoin de savoir si `x` est un nombre ou une chaîne. L'important est de savoir si

	Statique	**Dynamique**
Fort	Java, C#	Python, Ruby
Faible	C, C++	Perl, JavaScript, PHP

Tab. 1.1 Catégoriation de certains langages

```
1 def multiplier x, y
2   x * y
3 end
4 multiplier 3, 4     # => 12
5 multiplier "3", 4   # => "3333"
```

Fig. 1.5 *Duck typing* en Ruby

```
1 def multiplier x, y
2   raise "Type Error" unless x.respond_to? :*
3   x * y
4 end
```

Fig. 1.6 *Duck typing* en Ruby

x répond au message « * ». D'ailleurs, l'opérateur équivalent à `instanceof` en Ruby est `responds_to`. L'exemple de la figure 1.5 peut être réécrit avec cet opérateur (voir figure 1.6). Ce type de vérification est souvent vu comme étant superflu et limite la flexibilité du code (Thomas et Hunt, 2001).

L'équivalent Java du code présenté à la figure 1.5 est illustré à la figure 1.7. La première chose à remarquer est l'ajout d'une méthode supplémentaire. Cette nouvelle méthode n'existe que pour traiter les différents types des paramètres. Toute variation dans les types impliquera la définition d'une nouvelle méthode. Le besoin de spécifier le type d'un objet lors de sa création et son utilisation limite donc grandement la flexibilité du programme résultant.

Il existe bien entendu des techniques pour contourner le problème. La plus répandue

```
1  int multiplier(int x, int y) {
2    return x * y;
3  }
4  String multiplier(String x, int y) {
5    String r = "";
6    for (int i=0; i<y; i++ ) { r += x; }
7    return r;
8  }
9  multitplier(3, 4);   // => 12
10 multiplier("3", 4);  // => "3333"
```

Fig. 1.7 *Duck typing* en Java?

```
1  interface Multipliable<T> {
2    T multiplier(int facteur);
3  }
4  // ...
5  public class StringMultipliable
6      implements Multipliable<String> {
7    private String str;

8    public StringMultipliable(String str) {
9      this.str = str;
10   }
11   public String multiplier(int facteur) {
12     String r = "";
13     for (int i=0; i<facteur; i++) { r += str; }
14     return r;
15   }
16 }
17 // ...
18 public <T> T multiplier(Multipliable<T> x, int y) {
19   return x.multiplier(y);
20 }
21 // ...
22 multiplier(new StringMultipliable("3"), 4)   // => "3333"
```

Fig. 1.8 Utilisation de génériques en Java

consiste à programmer les interfaces plutôt que les implémentations. Chaque comportement générique est encapsulé dans une interface. Les méthodes ne spécifient plus les implémentations mais plutôt les facettes (les interfaces) que doivent avoir les objets. De cette façon, la flexibilité de la méthode est assurée. La syntaxe de Java à partir de la version 1.5 du JDK permet de définir des interfaces génériques où le type des variables, des paramètres et des types de retour n'est pas connu lors de leur définition. Le type est fourni lors de l'instanciation des objets (Flanagan, 2005). La lourdeur et le bruit syntaxique rendent l'utilisation des génériques souvent trop compliquée pour justifier les avantages de la flexibilité accrue et de la prétendue sécurité du typage statique (Eckel, 2008) (voir figure 1.8).

1.2 Métaprogrammation

Mieux connu pour ses expériences sur la communication avec les dauphins sous l'influence de drogues psychédéliques, John C. Lilly offre l'une des premières définitions de

la métaprogrammation (Lilly, 1972) :

> *When one learns to learn, one is making models, using symbols, analogizing, making metaphors, in short, inventing and using language, mathematics, art, politics, business, etc. (...)*
>
> *To avoid the necessity of repeating learning to learn, symbols, metaphors, models each time, I symbolize the underlying idea in these operations as metaprogramming.*

Avant d'apprendre, on doit d'abord « apprendre à apprendre ». Les enseignants parleront de « métacognition » i.e. réfléchir à propos de ses propres méthodes de réflexions. En assumant que la définition simpliste de l'activité de programmation est « écrire du code », les techniques de métaprogrammation permettent d'écrire du code à propos du code qui est écrit.

La métaprogrammation expose trois facettes : l'introspection (alias, la réflexion), l'introduction (alias, l'intercession) et le protocole méta-objet[3].

Introspection

L'introspection consiste en le concept de découvrir de l'information sur soi-même. Dans un environnement orienté-objet, les techniques d'introspection permettent à un objet de découvrir de l'information supplémentaire à propos de lui-même ou d'un autre objet.

Plusieurs langages de programmation statiques orientés-objets incorporent des outils d'introspection à leurs librairies standards. En Java, il est possible d'obtenir la liste des méthodes, des champs et des constructeurs d'une classe. L'API d'introspection de Java permet d'utiliser des chaînes de caractères pour spécifier le nom de la classe d'un objet à créer ou le nom d'une méthode à appeler. L'exemple à la figure 1.9 illustre les appels dynamiques de constructeurs et la modification de champs privés à l'aide de l'API

[3]en anglais *Meta-Object Protocol* ou MOP

```
1  public class Bidule {
2    private String champPriveEtImmuable;
3    public Bidule(String champPriveImmuable) {
4      this.champPriveEtImmuable = champPriveImmuable;
5    }
6  }
7  //...
8  Class<?> clazz = Class.forName("Bidule");
9  Constructor<?> constructor =
10     clazz.getConstructor(String.class);
11 Bidule bidule = (Bidule) constructor.
12     newInstance("champ immuable?");
13 Field champPriveEtImmuable =
14     clazz.getDeclaredField("champPriveEtImmuable");
15 champPriveEtImmuable.setAccessible(true);
16 champPriveEtImmuable.set(bidule, "acces direct!");
```

Fig. 1.9 Utilisation de l'introspection en Java 1.5

d'introspection.

Plusieurs cadres de développement modernes utilisent l'introspection afin de donner un comportement dynamique aux objets qui le composent. Par exemple, la norme *JavaBean* est en partie basée sur la convention que les accesseurs sont nommé `getFoo` et `setFoo` . Les *frameworks* peuvent donc, par introspection, appeler ces méthodes.

Les outils d'introspection ont été ajoutés à Java après sa conception initiale. Les langages de programmation dynamiques modernes ont été *conçus* en pensant à l'introspection. Il est donc normal de constater que Ruby, par exemple, incorpore une gamme impressionnante de méthodes permettant d'obtenir de l'information à propos d'un objet. Le tableau 1.2 illustre quelques-unes de ces méthodes.

Puisque les langages dynamiques déterminent le type d'un objet par les méthodes qui le compose et non par sa classe, le support et l'utilisation de l'introspection sont nécessaires. L'utilisation de la méthode `respond_to ?` à la figure 1.6 le démontre bien.

Nom	Description
`class`	La classe de l'objet.
`methods`	Les méthodes de l'objet
`send`	Invoque une méthode par son nom
`instance_variables`	Liste des champs
`instance_variable_get`	Retourne la valeur d'un champ
`instance_variable_set`	Modifie la valeur d'un champ

Tab. 1.2 Extrait de l'API d'introspection de Ruby 1.8.6

```
1  interface IBidule {
2    String uneMethode();
3  }
4  // ...
5  public class Bidule implements IBidule {
6    public String uneMethode() {
7      return "une methode...";
8    }
9  }
10 // ...
11 public class BiduleProxy implements IBidule {
12   private Bidule sujet = new Bidule();
13   public String uneMethode() {
14     System.out.println("proxy...");
15     return sujet.uneMethode();
16   }
17 }
18 // ...
19 IBidule bidule = new BiduleProxy();
```

Fig. 1.10 Introduction de comportement en Java 1.5

Introduction (ou intercession)

L'introduction est un concept moins connu des développeurs non-initiés aux langages agiles et dynamiques. L'introduction, parfois nommée « intercession », permet de modifier les méthodes d'une classe ou d'un objet lors de l'exécution.

Dans un langage statique, il est impossible de redéfinir une classe après sa compilation. Il est aussi complètement impossible de modifier le comportement d'une méthode après sa déclaration. Le besoin de le faire est pourtant bien réel. Celui-ci est exprimé par le patron de conception *Proxy* (Gamma et al., 1995). La figure 1.10 illustre le fonctionnement d'un *proxy* ajoutant la capacité de journalisation à une méthode.

```
1 public aspect Journalisation {
2   before() : execution(* Bidule.uneMethode(..)) {
3     System.out.println("AspectJ...");
4   }
5 }
```

Fig. 1.11 Introduction de comportement avec AspectJ

Le comportement supplémentaire est encapsulé dans une nouvelle classe *Proxy*. Celle-ci intercepte tous les appels de son sujet. Le rajout de comportement se fait avant ou après la délégation de l'appel. Il est bien entendu aussi possible d'en modifier les paramètres.

Le patron *Proxy* est très puissant et est au cœur de plusieurs cadres applicatifs comme Spring. On le retrouve lors d'ajout de contextes transactionnels, de journalisation, de sérialisation, etc. Toutefois, ce patron ne permet pas d'ajouter ou de retirer des méthodes d'un objet « sujet » sans en modifier l'interface publique. Entre en jeu la programmation orientée-aspects.

Un aspect est une série de comportement que l'on ajoute à une classe. L'implémentation la plus complète de l'AOP en Java est AspectJ (Laddad, 2003). AspectJ définit une syntaxe permettant de décrire des comportements particuliers (nommés « conseils ») qui sont insérés à des endroits spécifiques (des « points de jonction » ou « points de coupe ») dans des classes Java. La figure 1.11 montre la définition d'un aspect identique à celui de la figure 1.10 mais en utilisant la notation AspectJ. AspectJ permet d'intercepter des appels et des retours de méthodes ainsi que l'accès à des champs. Il est aussi possible de changer la superclasse d'un objet et de rajouter des méthodes et des champs.

AspectJ est un pas dans la bonne direction. Le comportement est centralisé et peut être écrit de façon à modifier plusieurs classes ou plusieurs méthodes à la fois. Il est même possible de retirer un aspect du code source sans modifier le code lui-même. Par contre, l'utilisation d'AspectJ ne permet pas de retirer des méthodes d'une classe. La syntaxe d'AspectJ est aussi considérablement différente de celle de Java et peut sembler rébarbative aux premiers abords.

```
1  class Bidule
2    def une_methode
3      "une méthode"
4    end
5  end

6  class Bidule
7    alias_method :old_une_method, :une_methode
8    def une_methode
9      puts "aspect..."
10     old_une_methode
11   end
12 end
```

Fig. 1.12 Introduction de comportement en Ruby

Voyons l'équivalent des exemples précédents en Ruby (figure 1.12). Les premières montrent la définition initiale de la classe. Les lignes 6 à 12 correspondent à la définition de l'aspect. On remarque que la syntaxe est exactement la même. Ruby exploite le principe de « classes ouvertes ». Il est toujours possible de rouvrir une classe afin de la modifier. Dans ce cas, la méthode est dupliquée, renommée puis redéfinie.

Il est ainsi possible de rouvrir n'importe quelle classe pour en modifier le comportement. Ce comportement est inhérent aux langages dynamiquement typés. La raison est simple : le type d'un objet n'est pas défini par sa classe mais bien par les messages qu'il peut recevoir. Le rajout de la possibilité de recevoir un message revient donc à changer le type d'un objet.

Ruby permet même de modifier les méthodes *d'une seule instance.* Le besoin d'avoir des instances uniques n'est toutefois pas le propre des programmes écrits à l'aide de langages dynamiques. L'omniprésence du patron de conception « Singleton » représente bien ce besoin.

MOP

Le protocole méta-objet est un concept en pratique absent de la plupart des langages statiquement typés comme Java et C++. Il peut être représenté par un ensemble de comportements entourant et interceptant le cycle de vie d'un objet. Le MOP est un ensemble

```
1  class Poisson
2    def self.method_missing meth, *args
3      if meth.to_s =~ /find_by_(.+)/
4        critere = "#{$1} = '#{args.first}'"
5        sql = "select * from poissons where #{critere}"
6        # exécuter l'énoncé SQL...
7      end
8    end
9  end

10 poissons_rouges = Poisson.find_by_couleur :rouge
```

Fig. 1.13 Utilisation du MOP

de *hooks*, de *callbacks* et *listeners* répondant à des événements de métaprogrammation.

Un premier exemple permet de mieux saisir l'importance du MOP dans la conception de DSLs. Imaginons un objet représentant une table dans une base de données relationnelle. Il s'agit du patron de conception *Table Module* (Fowler, 2003). Cet objet nous permet de réaliser les opérations CRUD habituelles. En plus de la recherche par clé primaire, il est souvent intéressant de pouvoir filtrer les résultats de la recherche selon certains critères. Par exemple, trouver toutes les sortes de poissons selon leurs couleurs.

En Java, ces exigences seront représentées par la définition d'un ensemble de méthodes généralement appelées « *finders* ». Leur nom est souvent calqué sur le modèle `findByCritère(...)`. Si les concepteurs de l'interface souhaitent isoler les utilisateurs de la syntaxe SQL sous-jacente, ils devront écrire autant de méthodes *finder* qu'il y a de critères de recherche.

Plusieurs langages dynamiques disposant d'un protocole méta-objet permettent de capturer l'événement où une méthode inexistante est appelée. Il est alors possible pour le concepteur de l'API d'y insérer des comportements dynamiques.

Dans notre exemple, nous souhaitons créer des *finders* dynamiques *i.e.* sans avoir à définir toutes les méthodes. Nous pouvons alors capturer l'événement « méthode introuvable » et, si le nom de la méthode voulue débute par « `find_by_` », en extraire les critères de recherche et l'exécuter. La figure 1.13 illustre un tel comportement.

La possibilité d'intercepter des appels de méthodes existantes ou non et l'accès à des pro-

priétés est la fonctionnalité d'un MOP la plus exploitée. Certains langages dynamiques permettent toutefois de capturer une plus grande gamme d'événements. Par exemple, en Ruby, il est possible d'insérer des comportements lors de l'ajout et du retrait de méthodes, lors de l'héritage d'une classe, lors de la recherche d'une constante, etc.

1.3 Fermetures

En C/C++, les « pointeurs de fonctions » sont considérés comme un concept avancé, difficile à comprendre, doté d'une syntaxe rébarbative et d'une utilité limitée dans le cadre du développement d'une application. On peut les comparer à cet outil dispendieux et ésotérique qui traîne au fond de notre coffre. Il est tout aussi rarement utilisé qu'il est utile lorsqu'un problème particulier fait surface.

Étant associé au paradigme « procédural », les pointeurs de fonctions sont souvent vus comme ne faisant pas partie du monde « orienté-objets ». Il n'est donc pas surprenant qu'ils n'apparaissent pas dans les premières versions de Java. Le besoin de passer des blocs de code en paramètre n'est pourtant pas moins présent. L'incarnation « objet » du pointeur de fonction est décrite par le patron de conception « Stratégie » (Gamma et al., 1995). Le patron est réalisé ainsi : une interface contenant la signature de la fonction est définie. Celle-ci est peut-être implémentée d'une ou de plusieurs façons. L'implémentation est passée comme paramètre à une méthode ou un constructeur. L'objet-fonction peut donc être transmis dans un message d'un objet à un autre.

Quoique pratique et orientée-objet, cette façon de faire n'est pas sans défaut. Premièrement, elle requiert la création d'au moins deux classes, soient une interface et une implémentation. L'utilisation massive du patron « Stratégie » implique la création de plusieurs classes et fichiers. Deuxièmement, l'objet-fonction est exécuté en vase clos. C'est-à-dire que la stratégie ne dispose pas d'une référence vers l'objet qui l'appelle. À cause de ceci, le contexte dans lequel la fonction s'exécute doit entièrement lui être passé en paramètre. Il serait possible pour l'objet appelant de passer une référence vers lui-même à la stratégie. On parle donc ici du concept de « *double-dispatch* » (Beck, 2008)

```
1  interface UneStrategie {
2    void uneFonction(String local);
3  }

4  class UneClasse {
5    void uneMethode(UneStrategie strategie) {
6      strategie.uneFonction("valeur locale");
7    }
8  }

9  // ...
10 final String contexte = "valeur du contexte";
11 new UneClasse().uneMethode(new UneStrategie() {
12   public void uneFonction(String local) {
13     System.out.println(contexte + ", " + local);
14   }
15 });

16 // => "valeur du contexte, valeur locale"
```

Fig. 1.14 Utilisation de classes anonymes internes

où le comportement d'une fonction est dépendant du type de l'expéditeur et du receveur du message.

Tôt dans l'évolution du langage Java, on voit apparaître les « classes anonymes internes ». Comme leur nom l'indique, ces classes sont définies à l'intérieur d'une autre et ne sont pas nommées. Tout en alourdissant un peu la syntaxe et la lecture d'une classe, elles viennent palier au premier problème soulevé, soit la création de classes superflues. L'avantage principal des classes anonymes internes réside dans le fait que *celles-ci s'exécutent dans le contexte où elles sont définies et non dans celui de leur appel.* Cette distinction est très importante. Les classes anonymes internes ont accès aux variables déclarées « `final` » dans la classe où elles sont déclarées. L'exemple de la figure 1.14 illustre ceci.

C# améliore le concept de classes anonymes avec les *annonymous delegates* et, depuis la version 3.0, les *lambdas.* Toutes ces structures, éléments de langages et limitations ne servent qu'à une seule chose : implémenter le concept de « fermeture ».

Une fermeture est un bloc de code anonyme, pouvant être échangé dans des messages et qui s'exécute dans le contexte où il est défini (Thomas et Hunt, 2001).

```
1 liste = [1,2,3,4,5]
2 limite = 4
3 println liste.findAll {it < limite}
```

Fig. 1.15 Utilisation d'une fermeture Groovy comme critère de recherche

```
1 def findAll(critereDeRecherche) {
2   def resultats = []
3   delegate.each { item ->
4     if (critereDeRecherche(item)) { resultats << item }
5   }
6   resultats
7 }
```

Fig. 1.16 Fonctionnement probable de la méthode `findAll`

L'exemple de la figure 1.15 illustre l'utilisation d'une fermeture dans le filtrage d'une liste de nombres. L'objet `liste` contient une méthode nommée `findAll` qui prend en paramètre une fermeture. La figure 1.16 montre le fonctionnement probable de la méthode `findAll`. Le critère est passé en paramètre et est exécuté pour chaque valeur de la liste. La valeur est ajoutée à la liste de résultats si la fermeture retourne `true`. Notons que notre implémentation « maison » de `findAll` utilise la méthode `each` qui permet d'exécuter une fermeture pour chaque item d'une liste.

1.4 Contexte d'évaluation

Un concept plus distant du monde des langages compilés est celui de la modification du contexte d'évaluation.

En Java, le contexte d'évaluation d'un bloc de code est défini en même temps que le bloc lui-même. Lorsqu'une méthode d'instance est définie à l'intérieur d'une classe, nous statuons implicitement que cette méthode sera exécutée dans un contexte précis. Dans ce cas-ci, il s'agit du contexte d'une instance de la classe. Après la définition de la méthode, il est impossible de l'exécuter dans un autre contexte.

La nature « évaluée » des langages dynamiques permet de modifier dynamiquement le contexte d'exécution d'un bloc de code. Les figures 1.17 et figure 1.18 montrent comment

```
1  class A
2    def methode
3      "méthode d'instance de la classe A"
4    end
5  end

6  class B
7    def self.methode
8      "méthode de classe de la classe B"
9    end
10 end

11 bloc = lambda do
12   puts methode
13 end

14 a = A.new
15 a.instance_eval &bloc  #=> contexte d'une instance de A
16 B.class_eval &bloc     #=> contexte de B
```

Fig. 1.17 Évaluation d'un bloc dans plusieurs contextes en Ruby

exécuter un même bloc de code dans deux contextes différents. Une première exécution est réalisée dans la cadre d'une variable d'instance. L'autre est celui d'une classe. Dans notre exemple, une fermeture est passée aux méthodes `class_eval` et `instance_eval`. Il est aussi possible de passer des chaînes de caractères et des fichiers contenant le code à interpréter.

On retrouve aussi cette fonctionnalité dans le langage Groovy. Par contre, l'approche diffère quelque peu. En Groovy, ce n'est pas tant le contexte d'évaluation qui peut être modifié que la référence vers le pointeur `this`. Il est possible de changer l'objet vers lequel une fermeture envoie ses appels de méthodes.

1.5 La chute de l'Empire Romain

Le retour des langages dynamiques à l'avant-plan de développements majeurs signifie-t-il l'ouverture d'une boîte de Pandore ? Devant cette situation, plusieurs affirmeront que « oui ». La montée de ces langages est vue comme étant équivalente à la traversée du Rhin par les Germains. C'est le début de la fin des langages compilés, la chute de la Rome statiquement typée. Les Barbares et les Vandales ont franchi les *limes*...

```
1  class A {
2    def methode() {
3      "méthode d'instance de la classe A"
4    }
5  }

6  class B {
7    static methode() {
8      "méthode de classe de la classe B"
9    }
10 }

11 bloc = {
12   println methode()
13 }

14 a = new A()
15 bloc.delegate = a
16 bloc()
17 bloc.delegate = B
18 bloc()
```

Fig. 1.18 Évaluation d'un bloc dans plusieurs contextes en Groovy

Mais qu'en est-il vraiment ? Quels sont les avantages des langages dynamiquement typés qui sont si précieux aux yeux de leurs tenants ? Les peurs causées par la nature dynamique de ces langages sont-elles justifiées ?

Une peur commune est celle de la perte de contrôle au niveau de la façon dont les méthodes sont invoquées. En effet, si on ne peut pas contrôler le type d'un objet, comment peut-on assurer la cohérence de l'exécution du code ? Devrons-nous tester le type de tous les paramètres d'une méthode avant de lancer son exécution ? L'avantage et le but principal des langages statiques est de capturer le plus de bogues possibles lors de la compilation du programme. Le compilateur peut donc nous informer que telle méthode n'est pas supportée par tel objet ; qu'un objet particulier ne peut pas être converti vers tel type. Cette dernière affirmation peut par contre être contre-carrée : si les compilateurs Java pouvaient réellement prévenir les erreurs de conversion de types, l'exception `ClassCastException` n'aurait pas sa raison d'être... Aussi, les collections Java avant la version 1.5 n'étaient pas typés. Tous les objets d'une `List`, par exemple, étaient du type `Object`. Ceci va clairement à l'encontre du but d'un langage statiquement typé et

pourtant, les erreurs causées par cette réalité sont rares.

Toujours est-il que la possibilité de détecter ces erreurs au moment de la compilation ne garantit évidemment en rien l'absence de bogues dans le programme. Une suite de tests (unitaires, fonctionnels et de régression) est nécessaire. Quoique non suffisants à la détection de tous les problèmes, les tests unitaires viennent palier à la validation statique des types.

Une autre préoccupation soulevée par rapport aux systèmes de types dynamiques est la possibilité de passer n'importe quel objet aux méthodes. Tel qu'illustré précédemment, il est effectivement possible de définir une méthode qui sera appelée de diverses façons et ce, possiblement dans de nombreux contextes. Plusieurs se demanderont comment limiter ce phénomène. Là où les langages statiques tentent de limiter la flexibilité d'utilisation des modules, les langages dynamiques l'encouragent. Il est donc possible d'utiliser de façon totalement innovatrice un module construit dans un but différent.

La force d'un programme n'est pleinement révélée que lorsqu'il est utilisé dans un contexte différent de celui de sa conception initiale. Les langages dynamiques permettent de réaliser ceci à l'échelle du programme entier et aussi à celle des modules qui le constituent.

Tout de même, la possibilité de découvrir rapidement (lire « à la compilation ») les méthodes d'un objet est intéressante. Les environnements de développement modernes utilisent cette capacité du compilateur pour offrir de l'assistance contextuelle ainsi que des facilités au développeur. Les IDE offrent présentement un support limité aux langages dynamiques. Puisque le programme doit-être exécuté afin de connaître les méthodes d'un objet, celles-ci ne peuvent être connues de l'IDE au moment de la rédaction. L'adoption à grande échelle des langages dynamiques passe par le support des IDE. Eclipse, NetBeans et IntelliJ offrent tous des outils Ruby, Python et Groovy. Malgré ceci, la nature de ces langages rend souvent l'utilisation des IDE traditionnels inappropriée. Les outils offerts par ces environnements sont taillés sur mesure pour les langages statiques. Des éditeurs de texte plus simples sont souvent mieux adaptés aux réalités du développement avec

des langages dynamiques.

Une autre préoccupation majeure est reliée à la nature permissive et libérale des langages dynamiques en ce qui concerne l'encapsulation des données. Les variables déclarées en Groovy et en Python ont une visibilité publique. En Ruby, les variables sont privées mais l'accès direct est quand même réalisable. Cette exposition des données privées donne assurément des munitions aux avocats défendant les langages statiques et des maux de tête à plusieurs développeurs récalcitrants. Doit-on s'inquiéter de ceci ? Tel qu'illustré précédemment, Java permet à n'importe qui de jouer avec les parties privées d'une classe depuis longtemps. Pourtant, aucun problème n'est soulevé. Peut-être est-ce dû à la complexité rattachée à l'introspection en Java. La gestion d'une panoplie d'exceptions et les appels de méthodes ésotériques donnent l'impression au programmeur de faire effectivement quelque chose d'interdit voire de dangereux. La déclaration d'une variable `public` semble mal. Mais il ne faut pas oublier le mantra des concepteurs de langages dynamiques : permettre plutôt que de restreindre (Fowler, 2004). Cette philosophie amène à considérer les variables comme étant « publiques jusqu'à preuve du contraire ». Prenons la définition d'une classe Groovy :

```
1 class Poisson {
2   def couleur
3 }
4 p = new Poisson()
5 p.couleur = "rouge"
```

La variable `couleur` est, en apparence, publique. En fait, Groovy rajoute dynamiquement les accesseurs nécessaires. Lorsqu'une propriété est accédée en lecture ou en écriture, Groovy intercepte l'accès et redirige l'appel vers l'accesseur approprié. Ceci donne l'impression qu'il est possible d'accéder directement aux propriétés. La philosophie de Groovy devient intéressante lorsqu'on réalise qu'il est possible d'intercepter l'accès à la variable *sans modifier l'interface publique de la classe*. Dans l'exemple suivant, le *setter* est redéfini :

```
1 class Poisson {
2   def couleur
3   def setCouleur(val) { couleur = val.toUpperCase() }
4 }
5 p = new Poisson()
6 p.couleur = "rouge"
```

L'exposition au grand public de ses parties privées peut, aux premiers abords, sembler indécent voire malpropre. L'important ici est de prendre en considération les avantages qui résultent de ce choix : une réduction du « bruit » causé par les accesseurs publiques et la possibilité de changer d'avis. Une dernière question à se poser serait : même si Java permet de briser l'encapsulation, quand fut la dernière fois que nous l'avons volontairement fait (Kelly, 2006) ?

Une autre peur provient des capacités de métaprogrammation et de la possibilité de surcharger les opérateurs. Les programmes qui « se réécrivent » sont difficiles à comprendre, à maintenir et à déboguer. Cette phobie de la métaprogrammation prend ses origines dans les premières tentatives réalisées, jadis, en assembleur. Ce qui doit être compris est que la métaprogrammation est un moyen de *simplifier* le code à écrire. Ceci est réalisé en encapsulant les comportements à l'aide de super-classes, d'annotations, de fermetures et de changements de contextes d'évaluation. Le problème qui dérive de cette façon de penser est l'augmentation de la charge cognitive imposée au programmeur.

Un dernier obstacle aux « envahisseurs » est composé de préjugés divers colportés par les fantômes des langages passés. Les « scripts » sont souvent considérés comme des programmes impropres à la consommation à l'extérieur d'un *shell*. Javascript et PHP représentent, pour plusieurs développeurs web, la source de tous leurs maux et failles de sécurité. La syntaxe de Lisp et de Perl est largement ridiculisée. De plus, les notions de programmation orientée-objet de ces langages semblent avoir été ajoutées de façon malhabile et improvisée.

Rien de tout ceci ne s'applique aux langages dynamiques modernes. La simplicité et la clarté de leur syntaxe tranchent sur la lourdeur de celle de Java ou de C++. La diminution du bruit syntaxique réduit considérablement le nombre de lignes de code à

comprendre et maintenir et donc, baisse le coût total (coût de développement et coût de maintenance) d'un logiciel (McConnell, 2004).

Ayant triomphé de tous les obstacles, les « barbares » sont maintenant aux portes de Rome et, pour l'instant, rien ne semble les arrêter.

* * *

Pendant longtemps, la définition du «type d'un objet» a été confondu avec la «classe de l'objet». Aucune distinction n'était faite entre les deux concepts. On peut maintenant voir que ce n'est plus obligatoirement le cas. Le type d'un objet est dissocié de sa classe. Ici, le type est défini par «les méthodes auxquelles l'objet répond». Le protocole méta-objet permet de s'immiscer dans le fonctionnement d'un langage dynamique pour en contrôler le comportement. Des concepts préalablement impossibles ou inimaginables deviennent choses communes : définition de classes et de méthodes à l'exécution, interception d'événements programmatiques (héritage, appel d'une méthode), etc. Loin d'être une hydre indomptable, la méta-programmation est une technique légitime qui mérite d'être exploitée judicieusement. Les langages dynamiquement typés peuvent être considérés comme un pas en avant. Nous les reconnaissons comme la prochaine étape dans l'évolution des langages de programmation utilisés en entreprise.

CHAPITRE II

DOMAIN-SPECIFIC LANGUAGES

Dans ce chapitre, nous traitons des langages spécifiques aux domaines d'affaire. Nous tentons de déterminer ce qui distingue les langages «spécifiques» des langages «généralistes». Nous verrons aussi deux sortes de DSL soient les «internes» et les «externes». Nous discuterons des différents usages des DSL en entreprise et comment ils peuvent faciliter la communication avec les parties prenantes d'un projet. Les DSL permettent aussi de changer le paradigme de développement traditionnel. La programmation orientée-langages (LOP) transforme l'approche habituelle de développement en incorporant plusieurs couches successives de DSL. Finalement, nous analysons quelques patrons de conception de DSL internes

* * *

L'activité de programmation est compliquée. Elle requiert une compréhension en profondeur des algorithmes, des possibilités des langages utilisés, des API, de l'architecture du système et, entre autres, de la logique d'affaire. Puisque la charge mentale ne cesse de s'accroître, des solutions doivent être développées afin de la réduire.

Les meilleurs développeurs sont les plus paresseux. Le concepteur du langage Perl, Larry Wall, définit la paresse en ces termes :

> *Laziness : The quality that makes you go to great effort to reduce overall energy expenditure. It makes you write labor-saving programs that other people will find useful (...)* (Wall, Christiansen et Orwant, 2000)

Une des façons les plus répandues pour alléger la charge cognitive est le développement

```
1 all: monprogramme

2 monprogramme: main.o dependance.o
3         g++ main.o dependance.o -o executable

4 main.o: main.cpp
5         g++ -c main.cpp

6 dependance.o: dependance.cpp
7         g++ -c dependance.cpp
```

Fig. 2.1 Exemple d'un « makefile »

de programmes, d'outils, permettant d'effectuer automatiquement les tâches répétitives et ennuyantes. Prenons la tâche de construction d'un logiciel. Elle est généralement composée de plusieurs étapes. On doit d'abord compiler les fichiers sources, les assembler, les lier, générer l'exécutable, exécuter les tests unitaires, possiblement déployer l'application sur un serveur, etc. Il n'est pas surprenant de retrouver dans un écosystème de développement un ou plusieurs outils de construction de logiciels. L'utilitaire « make » est peut-être l'un des plus connus. Le contenu d'un « makefile » permet de définir les règles et les étapes d'assemblage d'un programme particulier. Le contenu du fichier est alors lu, interprété puis exécuté. Un « makefile » minimaliste est présenté à la figure 2.1.

Pour un programmeur, la lecture du fichier « make » se fait sans trop de difficulté. On retrouve les quatre règles de construction du programme. Chaque règle est définie à l'aide d'une syntaxe précise. On énumère d'abord l'artefact logiciel souhaité (par exemple « main.o »). On indique ensuite à sa droite les artefacts qui seront nécessaires à sa réalisation. La ligne suivante contient l'instruction qui sera exécutée lorsque les dépendances seront réunies. Pour un développeur ayant à construire des programmes régulièrement, la syntaxe d'un « makefile » est claire et efficace.

Mais pourquoi avoir créé une nouvelle syntaxe pour décrire la construction d'un programme ? N'aurait-il pas été possible d'utiliser un programme écrit en C ou en C++ pour faire le même travail ? Bien entendu. Mais l'équivalent C n'aurait pas eu l'élégance et la précision du « makefile » présenté. On y aurait retrouvé des structures et des appels qui n'ont absolument rien à voir avec le présent domaine d'affaire, c'est-à-dire la

construction et l'assemblage d'un programme. Certains affirmeront que l'utilisation d'un programme de construction écrit dans un langage généraliste est supérieur puisqu'il permet d'utiliser toute la puissance du langage. Le langage C est bien entendu plus riche en possibilités que le langage « make ». L'avantage majeur d'un « makefile » est sa clarté et son efficacité dans la description d'une tâche d'assemblage. Un compromis est donc fait : la liberté et les possibilités d'un langage généraliste sont sacrifiés au profit d'une efficacité et d'une précision accrue *pour un domaine d'affaire en particulier.*

Ainsi, la distinction entre un langage spécifique et un langage généraliste se trouve dans le fait que le premier cible un problème en particulier. Un DSL ne peut prétendre pouvoir adresser tous les problèmes à résoudre dans le cadre du développement d'une application.

2.1 DSL par l'exemple

Les DSL peuvent évidemment prendre plusieurs formes. Une de leur utilisation principale survient lors de la configuration d'un programme. Prenons le cas de la création d'un interface utilisateur graphique. La boîte à outils Java toute désignée pour ce genre de travail est Swing. Swing permet de construire des interfaces riches en spécifiant les composantes qui seront présentes à l'écran, leur agencement et leur positionnement. La figure 2.2 illustre un exemple Swing simpliste. Le fragment de code présenté affichera une fenêtre à l'écran contenant le mot « Bonjour » en bleu suivi d'un bouton. Il va sans dire que ce fragment de code peut être difficile à suivre et à comprendre aux premiers abords.

Les DSL s'appuient sur les API existantes et en simplifient la lecture, l'écriture et la compréhension. XML est très certainement une norme en vogue. L'une des raisons qui explique son fort taux d'adoption est la simplicité avec laquelle il est possible de créer des DSL en XML. La structure d'un document XML est essentiellement un arbre syntaxique où les noeuds (*i.e.* les éléments et les attributs) sont des jetons. L'analyse lexicale et syntaxique habituellement tributaires d'outils tels que YACC et LEX n'est donc pas nécessaire. La structure du document impose la décomposition de l'information en jetons

```
1  JFrame frame = new JFrame();
2  frame.setTitle("Bonjour le monde");
3  frame.setSize(640, 280);
4  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
5  JPanel panel = new JPanel();
6  JLabel label = new JLabel();
7  label.setText("Bonjour");
8  panel.add(label);
9  JButton button = new JButton();
10 button.setText("le monde");
11 panel.add(button);
12 frame.add(panel, BorderLayout.CENTER);
```

Fig. 2.2 Construction d'une interface graphique en Swing

```
1  <frame size="640,280" title="Bonjour le monde
2         defaultCloseOperation="JFrame.EXIT_ON_CLOSE">
3    <panel constraints="BorderLayout.CENTER">
4      <label text="Bonjour"/>
5      <button text="le monde"/>
6    </panel>
7  </frame>
```

Fig. 2.3 DSL XML : Construction d'une interface Swing avec SwiXml

nommés et, ensuite, leur recomposition en forme d'arbre. On peut ainsi affirmer que la création d'un analyseur XML sur mesure est donc (presque) gratuite (lire : sans effort excessif). Il ne faut donc s'étonner du grand nombre de langages XML et du nombre de domaines dans lesquels on les retrouve. Par exemple, le langage SwiXml permet de décrire des interfaces Swing. La figure 2.3 décrit l'interface illustrée précédemment.

L'approche de la construction de DSL à l'aide d'un syntaxe XML n'est pas toujours la plus appropriée. Si le fichier de configuration doit être édité par une personne n'ayant les connaissances appropriées, la syntaxe XML devient rapidement un obstacle. XML

```
1  new SwingBuilder().frame(title:'Bonjour le monde',
2      size:[640,280]) {
3    panel(constraints: BorderLayout.CENTER) {
4      label(text:"Bonjour")
5      button(text:"le monde")
6    }
7  }
```

Fig. 2.4 DSL : Construction d'une interface Swing avec SwingBuilder

ajoute aussi une certaine lourdeur au langage et un bruit visuel considérable. Finalement, le document XML n'est pas intégré au programme. C'est-à-dire que le document doit absolument être lu et interprété afin d'interagir avec l'API sous-jacent. Une piste de solution possible à ce dernier cas est l'utilisation de constructions du langage hôte pour créer le DSL. Par exemple, le DSL Groovy nommé « SwingBuilder » permet, comme son nom l'indique, de bâtir des interfaces graphiques Swing. La figure 2.4 reprend notre exemple.

L'API « SwingBuilder » est considéré comme étant un DSL. Qu'en est-il des deux autres représentations ? On peut argumenter que tout langage XML est, en fait, un DSL. Peut-on aussi considérer le code Java de la figure 2.2 comme étant un DSL ? On pourrait affirmer qu'il en est un.

Le côté « déclaratif » des langages XML peut sembler très attrayant. En effet, un langage déclaratif est souvent simple à utiliser et sécuritaire. La courbe d'apprentissage n'est généralement pas très abrupte et la compréhension d'un script déclaratif peut sembler plus simple. Par contre, une syntaxe déclarative n'est parfois pas suffisante. Prenons le cas du langage XSLT. XSLT est une application XML qui permet de décrire la transformation d'autres documents XML. Le problème ici est que la description d'une transformation n'est pas de nature déclarative. Une transformation implique des boucles, des expressions conditionnelles, des tests de vérité, des appels de fonctions et de procédures, etc. XSLT est un langage procédural prisonnier d'une camisole de force « déclarative ». Un autre exemple probant est Ant. Ant est une application XML décrivant la construction et l'assemblage de programmes. Ici encore, la construction de programmes n'a rien de déclaratif. Plusieurs alternatives « procédurales » à Ant font leur apparition dont JRake et Gant et illustrent bien la nécessité d'une approche procédurale. Mais tout n'est pas noir pour les langages déclaratifs. En effet, les *frameworks* Spring et Hibernate sont deux cas où l'utilisation de fichiers de configuration déclaratifs plutôt que procéduraux est un succès. Ainsi, lors de la création d'un DSL, le choix entre une approche déclarative ou procédurale ne doit pas être pris à la légère.

Alors, qu'est-ce qu'un DSL ? Un DSL est un sous-ensemble d'un langage de programmation créé dans le but d'adresser un domaine de problèmes en particulier. Un DSL n'a pas besoin d'être « Turing-complet » ni même d'être exécutable. C'est une représentation du domaine du problème faite à l'aide d'une syntaxe adaptée à ce dernier.

Cette définition nous permet d'affirmer que les trois représentations de la même définition de fenêtres Swing peuvent donc être considérées comme étant des DSL. Leur syntaxe est pourtant très différente. La version Java est fortement couplée au langage « hôte ». En fait, il existe peu de différences entre un DSL « interne » et un API classique. La principale différence réside dans la facilité de lecture et de compréhension des DSL. La représentation XML est un DSL dit « externe ». Ceux-ci se distinguent des DSL « internes » par le fait qu'ils sont définis à l'extérieur du langage hôte. Un compilateur ou un interpréteur doit être écrit afin d'utiliser un tel DSL. L'intérêt des fichiers de configuration XML dans la communauté Java est marqué. XML est omniprésent dans presque tout développement Java de moyenne envergure. Des cadres et des conteneurs applicatifs comme Spring, Hibernate et Tomcat utilisent des documents XML pour gérer leur configuration. L'interprétation de ces fichiers au moment de l'exécution permet de modifier le comportement d'une application en court-circuitant le cycle de compilation-déploiement-exécution.

« SwingBuilder » utilise les possibilités propres au langage Groovy pour faciliter la création d'une interface graphique Swing. On remarque tout d'abord le passage de paramètres nommés aux fonctions. Ces paramètres forment en réalité un dictionnaire (une *Map* au sens de Java). L'ensemble des clés représente le nom des propriétés accessibles en écriture. Ces propriétés recevront les valeurs associées aux clés du dictionnaire. Autre point important, le nom des méthodes appelées correspond au nom de la classe des objets qui formeront l'interface (le préfixe « J » en moins). « SwingBuilder » intercepte l'appel de méthodes et, grâce au nom de celles-ci, appelle le constructeur approprié. Finalement, on utilise les fermetures pour assembler les composantes graphiques. Le fait d'appeler la méthode *panel* à l'intérieur de la fermeture passée à la méthode *frame* fera en sorte que la composante créée par cette dernière contiendra la composante créée par

```
1 maintenant = Time.now
2 apres_demain = 2.days.from_now
3 semaine_derniere = 1.week.ago
```

Fig. 2.5 DSL interne permettant la création de dates

la première. Cette recette, ou patron, est communément appelée *Builder* dans le jargon des DSL. On retrouve généralement les *builders* partout où la construction de structures hiérarchiques est nécessaire : XML, HTML, Swing, interfaces web, etc.

Évidemment, l'application des DSL n'est pas limitée à la construction de ces structures. Le *framework* Rails incorpore un DSL permettant de représenter des dates de façon simple et concise. La figure 2.5 en illustre l'utilisation. Les DSL de ce type abondent. Ils sont généralement caractérisés par l'ajout de méthodes à un type « quasi-primitif » comme les nombres, les chaînes et les dates. Ils sont présents partout où les conversions entre différentes représentations sont nécessaires. Par exemple, si on imagine un domaine de problèmes où les conversions monétaires sont importantes, un DSL permettant d'exprimer certaines règles d'affaire pourrait être utilisé : « `effectuer_achat lorsque prix.en_CAD + taxe.fédérale excède frais_douane.en_USD` ».

2.2 Buts, avantages et inconvénients

L'élaboration d'un DSL requiert, sans l'ombre d'un doute, un investissement majeur en temps. Sa justification se trouve dans le retour sur investissement souhaité. Les bénéfices doivent donc être suffisamment importants pour justifier le temps alloué à la construction d'un DSL.

Un des buts recherchés est une meilleure collaboration avec les experts du domaine d'affaire et les programmeurs. Le manque de communication et/ou de compréhension entre ces deux parties peut devenir la cause principale de la livraison d'une application incomplète ou incorrecte. Le schème de communication entre ces deux parties peut être résumé ainsi.

1. L'expert énonce une règle d'affaire ou une réalité du domaine ;

2. Le développeur traduit ce fait en code exécutable ;
3. L'expert confirme ou infirme que la réalité a été traduite correctement.

Ceci écarte l'expert du développement de l'application. Cette communication en trois temps entraîne un autre problème. L'expert est habituellement incapable, pour diverses raisons, de verbaliser tous ces besoins du premier coup. Loin de nous l'idée d'affirmer que l'expert ne connaît pas son domaine d'affaire. Bien au contraire ! Par contre, la logique d'affaire (les cyniques diraient « l'illogisme d'affaire ») est souvent si profondément ancrée dans ses habitudes de travail quotidiennes qu'il lui est impossible de toutes les réciter pour le bénéfice du programmeur. Cette réalité est embrassée par les tenants de la pratique du prototypage rapide et des techniques dites « Agiles ». Le prototypage et les techniques de développement incrémental visent à mettre dans les mains du client un programme incomplet et ce, le plus rapidement possible. S'il est difficile d'exprimer ce que l'on souhaite, il est beaucoup plus simple d'exprimer ce qui ne fonctionne pas correctement. De cette façon, il est possible d'inciter le client à découvrir et à exprimer ces besoins plus clairement.

L'intégration des règles d'affaire dans l'application peut se faire de plusieurs façons. Il est évidemment impossible pour un développeur de maîtriser le domaine d'affaire au même niveau que l'expert dans les limites de temps imposées au projet. Inversement, il est impossible pour un expert du domaine d'affaire d'en faire autant en ce qui concerne l'art de la programmation. Le principe de séparation des préoccupations nous indique qu'il est préférable pour chaque expert de se concentrer sur son propre métier.

L'utilisation d'un DSL pour programmer ou configurer une partie du logiciel peut servir de langage commun entre l'expert du domaine et le développeur. Ce DSL n'utilisera pas une syntaxe inventée de toute pièce. En fait, les notations des experts seront utilisées. Le code source produit sera interprété par les programmeurs et ultimement, directement par les programmes eux-mêmes. Évidemment, si ces notations sont graphiques plutôt que *typo*graphiques, il serait important que le DSL le soit aussi.

Dans cette optique, le rôle du programmeur se transforme. Il passe de « traducteur de

la logique d'affaire » à « concepteur de langages et de compilateurs ». On permet ainsi à chaque expert de se concentrer sur ce qu'il connaît le mieux. Pour l'un, ce sont les règles du domaine d'affaires et pour l'autre, la programmation.

Les avantages des DSL ne se limitent pas à l'amélioration de la communication avec le client. Le code source est beaucoup plus souvent lu qu'il n'est écrit. En partant de cette réalité, il est normal de penser que du code lisible est préférable à du code qui ne l'est pas. N'importe quel effort de *refactoring*, de « déboggage » ou de maintenance commence par la lecture et l'appropriation du code existant. On doit le comprendre avant de le modifier. L'élaboration d'un API simple et efficace, conçu pour faciliter la lecture du code source est donc un atout pour toute équipe de maintenance. Nous qualifierons cet API de « DSL interne ».

La première inquiétude soulevée par cette situation est la potentielle perte de travail pour les programmeurs. Cette peur est largement infondée. Plusieurs technologies ont déjà proclamé pourvoir réaliser l'irréalisable en commençant par COBOL. Le métier de programmeur n'est donc pas du tout en péril. L'utilisation de DSL dans un objectif de coopération et de synergie entre le travail des programmeurs et des experts ne peut être que bénéfique.

Une autre préoccupation souvent entendue est reliée au problème que nous nommerons la « Tour de Babel ». Pour plusieurs, l'apprentissage d'un nouveau langage de programmation est une aventure périlleuse. L'apprentissage est vu comme étant un obstacle à la productivité et à l'agilité du développement. La prolifération de langages de toutes sortes est une inquiétante source de problèmes en devenir. Les équipes de maintenance devront apprendre ces langages « maisons », les maîtriser, en maintenir le compilateur, etc. Cette peur est contre-balancée par l'applicabilité et la syntaxe limitée des DSL. Lorsque l'on considère les langages XML et les API comme étant des DSL à par entière, on se rend compte combien de langages il est nécessaire de maîtriser pour développer une application modeste. Le problème de la « Tour de Babel » est, plus souvent qu'autrement, lié à une incompréhension de la nature d'un DSL.

L'utilisation d'un DSL externe nécessite la construction d'un compilateur. Peu de programmeurs souhaitent relever le défi que ceci implique. Les compilateurs sont perçus comme des boîtes noires, mystiques et insondables où sont enfermées des secrets qui ont intérêts à le rester. Ces appréhensions sont parfois justifiées. Heureusement, plusieurs outils permettent de tuer le Dragon. On les catégorise généralement de « CC » (*Compiler Compilers*) et viennent sous différents noms : Lex/YACC, JavaCC, SableCC, RACC et Antlr pour ne nommer que ceux-là.

Le véritable défi n'est donc pas l'écriture de compilateurs. Il se trouve plutôt dans le design du langage lui-même. Le design d'un langage utilisable et simple, même s'il est limité à un domaine d'affaire spécifique, requiert des connaissances particulières.

2.3 Programmation orientée-langages

Jusqu'ici, la portée des DSL illustrés semble limitée. Ils sont qualifiés d'interfaces programmatiques lisibles ou de fichiers de configuration modifiables par les experts du domaine. L'utilisation des DSL peut toutefois être poussée beaucoup plus loin.

Le programmation orientée-langage (Ward, 1994) est une approche différente à la conception d'applications. Elle est basée sur l'utilisation et l'élaboration récursives de plusieurs langages afin de décrire une application. Afin d'illustrer le fonctionnement de la programmation orientée-langages, prenons exemple sur une application Java ordinaire. Un programme est décrit premièrement avec le langage Java. Le code est traduit, compilé, en code-octet. Le code-octet est alors compilé ou interprété en langage machine. Finalement, le langage machine est interprété en micro-code, en impulsions électriques. Le fonctionnement de notre application Java est donc assuré par un ensemble de langages, une chaîne de compilations et d'interprétations successives. La suite de compilations et de langages permet de hausser progressivement le niveau d'abstraction, de l'impulsion électrique à l'énoncé Java. Il nous est ainsi possible de construire des programmes plus complexes et plus rapidement en Java qu'il ne l'est en micro-code.

Java est un langage généraliste. On peut donc l'utiliser dans plusieurs circonstances et

```
1 Query query = QueryFactory.queryForDatabase(database);
2 query.addTable("poissons");
3 query.addColumn("id");
4 query.addColumn("nom");
5 query.addColumn("couleur");
6 query.getOrderingScheme().setDescendingBy("couleur");
7 List<Result> results = query.execute();
```

Fig. 2.6 Tentative de définition d'un API de recherche « tout-java »

pour différents besoins. Cette polyvalence est un avantage majeur pour tout langage. Par contre, ce même avantage rend Java inadéquat pour décrire une application dans un domaine précis. Prenons le domaine des requêtes dans des bases de données relationnelles. Il serait possible d'utiliser Java pour décrire une opération de lecture. La figure 2.6 illustre une tentative d'un tel API. L'équivalent SQL est évidemment, beaucoup plus simple et plus lisible que ceci. La raison est simple : SQL est spécifiquement conçu pour ce domaine en question.

La programmation orientée-langage pousse ce concept à un degré supérieur. On commence par définir un langage dédié de très haut niveau permettant de décrire l'application en entier. Une attention particulière doit être apportée à l'élaboration de la syntaxe de ce langage. Lorsque ceci est fait, le travail peut être divisé en deux. Une équipe, dirigée par l'expert du domaine d'affaire est chargée d'écrire l'application en utilisant le nouveau DSL. La seconde équipe doit implémenter le DSL à l'aide d'interpréteurs ou de compilateurs. Ces opérations sont reprises de façon récursive jusqu'à ce que le niveau d'abstraction soit suffisamment bas pour que l'implémentation soit réalisée à l'aide d'un langage généraliste exécutable. La réutilisation des langages implémentés est un argument de vente majeur.

On peut rapidement voir l'intérêt d'une telle méthode de conception dans le cadre du développement d'une famille de logiciels. Les programmes essentiellement semblables mais accidentellement différents pourraient bénéficier de ces langages communs. Prenons comme exemple les applications web de type *Wizard*. Ces applications sont toutes, du point de vue de leur essence du moins, identiques. L'utilisateur débute une session. On

```
1  PageA {
2    contenu "Bonjour!"
3    question "Entrez votre code postal", :code_postal
4  }
5  PageB {
6    contenu "Vous habitez à Montréal."
7  }
8  PageC {
9    contenu "Vous n'habitez pas à Montréal."
10 }
11 PageD {
12   contenu "Au revoir!"
13 }

14 Transitions {
15   PageA => PageB if :code_postal.starts_with? "H"
16   PageA => PageC unless :code_postal.starts_with? "H"
17   PageB => PageD
18   PageC => PageD
19 }
```

Fig. 2.7 DSL décrivant une application *Wizard*

lui présente un ensemble de questions, si les réponses sont valides, il peut procéder à la page de questions suivante. Celle-ci est déterminée par les réponses aux questions précédentes. L'utilisateur peut aussi revenir à la page précédente à l'aide d'un bouton mis à sa disposition à cet effet. On peut rapidement imaginer un DSL permettant de décrire ce type d'applications. La figure 2.7 est un exemple de ce DSL. Le contenu et les questions de chaque page sont définis. Ensuite, on élabore les règles de navigation. Dans ce cas-ci, une question est posée à l'utilisateur. La page suivante est déterminée par la réponse à cette question.

2.4 Techniques, patrons et recettes

Le design d'un DSL est similaire à tout autre effort de conception. Il requiert une rigueur et une expertise particulière. Comme le design « objet », l'élaboration d'un DSL interne demande certaines connaissances en matière de construction de langages. Heureusement pour nous, la portée d'un DSL est beaucoup plus limitée qu'un langage généraliste. La conception est donc quelque peu simplifiée. À cet égard, l'élaboration d'un DSL interne et d'un API traditionnel sont apparentés.

Deux approches peuvent mener à la construction d'un DSL interne. Nous qualifierons la première de *top-down*. Elle consiste à commencer par l'élaboration de la syntaxe du DSL. On procède ensuite à son implémentation. Cette technique est analogue à la programmation orientée-langages décrite à la section 2.3. Nous nommerons la seconde approche *bottom-up*. Elle consiste à simplifier l'utilisation d'un API existante à l'aide d'un DSL (Mernik et Sloane, 2005).

L'approche *top-down* trouve son utilité dans la communication de l'intention du programmeur au client. Pour qu'un langage soit qualifié de « DSL », les éléments de syntaxe qui le constitue doivent être calqués sur le domaine d'affaire. Lorsque le DSL est élaboré avant l'implémentation, son but premier est de communiquer avec l'expert du domaine. C'est une façon d'impliquer le client dans la construction de son produit. À ce moment, il n'est plus spectateur, il devient un acteur actif. Un DSL n'a pas besoin d'être exécutable. Son rôle de communication avec le client est parfois suffisant. Le programme écrit avec le client à l'aide du DSL doit alors être converti dans un langage généraliste afin de pouvoir être exécuté. Cette conversion introduit un élément d'incertitude dans le développement. Une erreur de traduction peut évidemment survenir. De façon plus fréquente, le programmeur peut interpréter la spécification incorrectement. Ainsi, il traduira correctement son incompréhension du domaine d'affaire. En plus des erreurs d'interprétation, la traduction du DSL entraîne une duplication du code. La même spécification se retrouve à deux endroits différents. Tous les problèmes reliés à la duplication de code s'appliquent. La solution idéale est d'utiliser directement le code source rédigé et validé avec le client.

Une première façon d'accomplir ceci est de construire un générateur de code i.e. un compilateur. Le langage « source » est le DSL ; le langage « destination » est le langage généraliste choisi. Quoiqu'il existe de nombreux outils permettant de « compiler des compilateurs », la rédaction d'une grammaire, de règles de dérivation, d'arbres syntaxiques et de « Visiteurs » (Gamma et al., 1995) ne sont pas choses simples à apprendre et à maîtriser. De plus, l'ajout d'éléments de langage au DSL devient une aventure de refactorisation de grammaire et de tests de régression.

Le générateur de code peut aussi être écrit en XSLT. En combinaison avec des outils de validation comme XMLSchema, RELAX NG et Schematron, on peut affirmer que l'analyse lexicale et syntaxique d'un document XML est « gratuite ». XSLT permet de décrire la transformation d'un arbre syntaxique XML vers un autre format. Ainsi, un DSL XML serait une solution possible. Toutefois, la verbosité d'XML et le bruit lexical entraîné par la présence des balises font en sorte qu'un DSL XML n'est pas toujours le plus approprié. XML permet de créer de bons langages déclaratifs et hiérarchiques. Lorsque le domaine d'affaire s'y prête, XML est choix intéressant. Par contre, dès que le problème sort de cette zone de confort, XML devient rapidement un handicap. Prenons l'exemple du langage XML Ant. Ant est un DSL permettant de décrire les étapes de construction d'un programme et leurs dépendances. Malheureusement, la construction d'un programme n'est pas une activité purement déclarative. Elle nécessite souvent des actions supplémentaires et donc, des éléments de langages inexistants dans Ant. Il n'est pas rare de faire appel à des scripts à l'intérieur d'un document Ant afin de réaliser une ou plusieurs étapes de construction. Même le concepteur de Ant admet que XML n'a pas été le meilleur choix (Davidson, 2004).

La construction d'un générateur de code implique souvent que le langage source et le langage destination sont différents. Au lieu de transformer le code source, il est aussi possible d'implémenter le DSL *à l'aide du langage généraliste*. De cette façon, la transformation n'est plus nécessaire. Le DSL devient un sous-ensemble du langage hôte. On parlera alors de DSL interne i.e. un DSL qui ne requiert pas de transformation additionnelle mais qui est plutôt directement lié au langage exécutable. Contrairement à un DSL nécessitant une transformation de code, un DSL interne est automatiquement utilisable. De plus, toutes les structures et les éléments du langage hôte sont accessibles à l'intérieur du DSL. Ceci nous permet d'élaborer des DSL plus complets et ce, en moins de temps. Par contre, les DSL internes ont les défauts de leurs qualités. Si le DSL doit résider à l'intérieur d'un autre langage, il est automatiquement limité dans son expressivité par rapport à une syntaxe définie sur mesure par le programmeur. Lors du design d'un DSL interne, on doit donc obligatoirement se plier aux exigences et limitations du langage

hôte.

L'approche *bottom-up* mène très souvent à l'élaboration d'un DSL interne. Dans ce cas-ci, le DSL est une interface plus conviviale à un API déjà existant. Le *SwingBuilder* de la figure 2.4 est un exemple typique d'un *Builder* : un DSL permettant de construire une structure hiérarchique. Dans cet exemple, le *SwingBuilder* est une interface plus fluide à la construction d'une interface graphique Java.

L'idée d'un DSL interne n'est pas nouvelle. On peut la voir dans plusieurs programmes Lisp. Son utilisation est plutôt restreinte dans les langages « grand public » comme Java et C#. On redécouvre aujourd'hui les DSL internes grâce aux nouveaux langages dynamiques comme Python, Groovy et Ruby. Certains éléments rendent ces langages *DSL-friendly*. Très certainement le *Meta Object Protocol* et les fermetures sont les premiers responsables. Nous présentons ici quelques patrons de design de DSL souvent utilisés. Plusieurs de ces patrons sont documentés dans (Fowler, 2008).

2.4.1 Méthodes indéfinies

L'utilisation du *Meta Object Protocol* afin de créer des DSL est une technique largement répandue. Pour illustrer nos propos, imaginons une méthode permettant d'utiliser des nombres romains. La méthode « parseNombreRomain » de la classe suivante permet de transformer une chaîne de caractères représentant un nombre romain en nombre entier.

```
1  // Groovy
2  class Romain {
3    def parseNombreRomain(String nombre) {
4      def total = 0, precedente = 0
5      def valeurDe =
6        [i:1, v:5, x:10, l:50, c:100, d:500, m:1000]
7      nombre.toLowerCase().each {
8        total += valeurDe[it]
9        if (valeurDe[it] > precedente) total -= 2*precedente
10       precedente = valeurDe[it]
11     }
12     total
13   }
14 }
15 r = new Romain()
16 r.parseNombreRomain("LXXXIX")       // => 89
17 r.parseNombreRomain("MMMCDXLVIII") // => 3448
```

Comme le montrent les dernières lignes de cet exemple, l'utilisation de cette classe pour effectuer de l'arithmétique est encombrante. On peut améliorer la situation en interceptant l'accès en lecture à une propriété indéfinie. On effectue ceci en redéfinissant la méthode « getProperty(String nom) ». Ici, on traitera les propriétés indéfinies comme étant la représentation d'un nombre romain. On ajoute aussi une validation simpliste permettant d'identifier si le nom de la propriété est composé des symboles d'un chiffre romain. Une erreur sera lancée dans le cas contraire :

```
1  class Romain...
2    def getProperty(String prop) {
3      if (prop =~ /[IVXLCDM]+/)
4        return parseNombreRomain(prop)
5      else
6        return this.@prop
7    }
8  }
9  r = new Romain()
10 assert r.LXXXIX + r.MMMCDXLVIII == r.MMMDXXXVII
```

2.4.2 Modification du délégué

Tel qu'illustré à la section 1.4, les langages supportant le concept de fermetures permettent d'en modifier le contexte d'évaluation. Dans le cas de notre exemple de nombres romains, il est encore nécessaire d'instancier un objet de type Romain. Pour simplifier

encore plus l'utilisation de la classe, on ajoute une méthode statique recevant un bloc de code à exécuter. On change ensuite le contexte d'évaluation du bloc de code en lui assignant une instance de la classe Romain. Tous les appels à des propriétés inconnues seront d'abord attrapés par la méthode « getProperty » du nouveau délégué de type Romain.

```
1 class Romain...
2   static eval(bloc) {
3     bloc.delegate = new Romain()
4     bloc()
5   }
6 }
7 Romain.eval {
8   assert LXXXIX + MMMCDXLVIII == MMMDXXXVII
9 }
```

2.4.3 Évaluation externe

La technique de la modification du délégué est aussi utilisée afin de sortir le code du DSL du moteur permettant de l'exécuter. Supposons que la classe Romain a été implémentée en Ruby. Voici le contenu du code source.

```
1 # Fichier : test.romain
2 lxxxix + mmmcdxlviii
```

Pour interpréter le code source contenu dans le fichier, on doit d'abord en lire le contenu. Ensuite, la méthode statique `module_eval` permet d'évaluer le contenu du fichier dans le contexte de la classe Romain.

```
1 # Ruby
2 code_source = File.new("test.romain").read
3 Romain.module_eval code_source
```

Ce patron est souvent utilisé lorsque le code source a besoin d'être modifié *après* la compilation du programme. On pense ici à des fichiers de configuration, des règles d'affaires, des structures de données. Les fichiers de configuration XML en sont un bel exemple. Ce

patron est utilisé par des cadres applicatifs comme Log4J, Spring, Hibernate, Servlet, etc. Le fait reste que le code source (qu'il soit XML ou non) reste un DSL interne. On bénéficie donc du meilleur des deux mondes : la puissance du langage hôte et la division du code DSL du code « réel ».

Ce patron illustre aussi un aspect important des langages pleinement interprétés : la possibilité de modifier directement le code source sans phase de compilation. Ceci peut évidemment être vu comme un avantage et un problème. L'interprétation directe de code potentiellement malicieux ouvre la porte à un monde d'attaques à injections diverses... Ici, un DSL XML permet, à l'aide d'outils comme XMLSchema, d'effectuer une validation minimale de la forme du document. Cette phase de validation permet d'éviter certaines attaques contre les analyseurs XML. Toutefois, les attaques d'injection XPath et XQuery sont toujours possibles. Les attaques de déni de service via XML sont aussi inévitables.

2.4.4 Chaînage de méthodes

Le patron de chaînage de méthodes est une forme d'API permettant de construire des structures de façon plus conviviale. On peut le considèrer aussi comme la récupération du patron *Builder* par les concepteurs d'API utilisant des langages n'ayant de fermetures. Le patron de chaînage de méthodes est parfois appelé « *fluid API* ».

Le chaînage de méthodes permet de construire des contextes d'évaluation de méthodes à l'aide de la valeur de retour de la méthode précédente dans la chaîne. La figure 2.2 illustre l'utilisation de l'API Swing. Lorsqu'une méthode est appelée, le contexte d'évaluation n'est pas modifié. Peu importe ce qu'une méthode a créé précédemment, les méthodes suivantes n'en seront pas conscientes. Le programmeur doit passer manuellement le contexte d'évaluation en paramètre aux méthodes suivantes. L'utilisation de *Builders* permet de modifier le contexte d'évaluation des méthodes.

Les langages statiques sont souvent maladroits dans leur façon d'attribuer un tel contexte. Dans un API fonctionnant par chaînage de méthodes, le contexte d'évaluation de la méthode *n* est donné par la valeur de retour de la méthode *n-1*. Une interface « fluide »

pour l'API Swing pourrait ressembler à ceci :

```
1  // Java
2  new SwingGUI().
3    frame().
4      title("Bonjour le monde").
5      size(640, 280).
6      panel().
7        constraint(BorderLayout.CENTER).
8        label("Bonjour").
9        button("le monde").
10     end().
11   end().
12 end();
```

L'indentation donnée à cet extrait de code source est purement une question d'esthétique. Elle permet par contre de montrer clairement les différents contextes d'évaluation. En lisant ce code, on constate que la méthode `frame()`, en plus de créer une instance de la classe JFrame retourne un objet qui représente un contexte d'évaluation. Ce contexte contient minimalement la méthode `title()`. Cette dernière retourne le contexte. On procède ensuite à l'appel de `size()` puis de `panel()`. La méthode `panel()` crée un nouveau contexte d'évaluation. Pour revenir au contexte du `frame`, on doit indiquer à l'API de fermer, de terminer un contexte. Ainsi, les API « fluides » contiennent des *enders* : des méthodes indiquant de basculer au contexte précédent. De cette façon, la méthode `end` à la ligne 10 finalise le contexte d'exécution du JPanel et retourne le contexte d'évaluation du JFrame.

On retrouve ce patron dans plusieurs circonstances :

1. La création de structures de données (XML, JSON)
2. La création d'objets maquettes (jMock)
3. La définition de règles de *workflows* (Apache Camel)

2.4.5 *Builder* statique

Les *builders* statiques sont ainsi nommés puisqu'ils n'utilisent pas les capacités du *Meta Object Protocol*. Un *Builder* statique permet de construire des structures dont les élé-

ments sont connus et leurs imbrications sont régulières et prédéterminées.

Reprenons notre exemple de construction d'interfaces graphiques. Les langages ne disposant pas de fermetures doivent utiliser d'autres moyens pour modifier le contexte d'évaluation de fonction. Le chaînage de méthodes est l'une de ces techniques. L'utilisation de fermetures imbriquées de la modification du délégué nous permet de faire l'équivalent en réduisant la quantité de bruit syntaxique et de ponctuation. Voici un exemple d'utilisation de l'API que nous souhaitons construire :

```
1  # Ruby
2  GuiBuilder.frame do
3    title "Bonjour le monde"
4    size [640,280]
5    panel do
6      constraint :CENTER
7      label "Bonjour"
8      button "le monde"
9    end
10 end
```

La méthode `frame` prend en paramètre un bloc de code i.e. une fermeture. Celle-ci sera exécutée dans un certain contexte d'évaluation. Il en va de même pour la méthode `panel`. L'implémentation d'un *Builder* statique est une recette un peu répétitive.

1. On définit la classe de tous les objets qui font partie du modèle. Dans notre exemple, on retrouve Frame, Panel et Button.
2. On définit un `Builder` par contexte d'évaluation. Ici, il y a un contexte pour la construction du Panel, du Frame et de l'interface en entier.
3. Chaque *Builder* contient une méthode qui active la construction. Cette méthode instancie l'objet du modèle et évalue le bloc de code. Lorsque l'évaluation est terminée, la méthode retourne l'objet créé et initialisé par le bloc.
4. Chaque *Builder* doit contenir les méthodes qui seront appelées par le bloc.

Débutons avec la classe `GuiBuilder`. Celle-ci contient une seule méthode nommée `frame`. Cette méthode est le point d'entrée de la construction. Elle crée un tableau d'objets `Frame`. L'évaluation du bloc de code est déléguée au *Builder* d'objets `Frame` :

```ruby
# Ruby
class GuiBuilder
  def self.frame &block
    @frames ||= []
    @frames << FrameBuilder.new.build(&block)
    @frames
  end
end

class FrameBuilder
  def build &block
    @frame = Frame.new
    instance_eval &block
    @frame
  end
  def title arg
    @frame.title = arg
  end
  def size arg
    @frame.size = arg
  end
  def panel &block
    @frame.panels << PanelBuilder.new.build(&block)
  end
end
```

Les *Builders* de `Frame` et de `Panel` sont presque identiques. La méthode `build` crée l'objet du modèle et les méthodes du *builder* modifient cet objet.

Les *Builders* statiques sont particulièrement intéressants lorsque les structures qui doivent être créées sont régulières. Le nom des opérations permises et le niveau d'imbrication des fermetures doivent être préalablement connus. Toutes modifications à cet ordre implique la modification ou la création de *Builders*. Cette particularité permet une certaine validation lors de l'utilisation d'un *Builder* statique : si la méthode n'existe pas, on utilise l'API incorrectement.

2.4.6 *Builder* dynamique

La forme de certaines structures ne peut pas être préalablement connue. Pensons à un document XML. On pourrait écrire un API permettant de créer n'importe quels documents XML à l'aide des patrons « Chaînage de Méthodes » et « *Builder* statique », mais son utilisation serait maladroite :

```
// Java
new XmlBuilder().
  element("racine").
    attribute("clé", "valeur").
    element("branche").
      element("feuille").
        text("Bonjour le monde").
      end().
    end().
  end();
```

Ceci reflète bien la structure du document et est supérieur à l'API DOM en ce qui concerne la création de documents XML. Par contre, la quantité de ponctuation nécessaire pour faire fonctionner cet API est impressionnante. Chaque sorte de noeud XML correspond à une méthode. Le bruit visuel nuit à la compréhension de la structure.

Puisque le nom des annotations XML est préalablement inconnu, on ne peut pas écrire un *Builder* statique. De plus, on voit que le passage du nom de l'annotation en paramètre aux méthodes `element` et *attribute* est peu lisible. Nous pouvons par contre utiliser le *Meta Object Protocol*. Celui-ci nous permet d'intercepter le nom de méthodes inconnues. Afin de traiter un niveau arbitraire d'imbrication de contextes, une seule instance de *Builder* est nécessaire. Celle-ci est passée récursivement lors de chaque ouverture de contexte. Voici un exemple de l'utilisation souhaitée :

```
# Ruby
XmlBuilder.new.racine do
  branche do
    feuille "Bonjour le monde"
  end
end
```

L'implémentation simpliste de la classe `XmlBuilder` pourrait être :

```
# Ruby
class XmlBuilder
  def method_missing meth, *args, &block
    puts "<#{meth}>"
    instance_eval &block if block
    puts "</#{meth}>"
  end
end
```

Évidemment, cette implémentation ne traite pas le contenu texte, les attributs, les espaces de nommage, etc. Des librairies bien plus complètes permettent de faire ceci. La classe `XmlMarkup` en est une.

```
1  # Ruby
2  document = lambda {
3    instruct!
4    comment! "Un commentaire..."
5    racine "clé" => "valeur" do
6      branche do
7        feuille "Bonjour le monde"
8      end
9    end
10 }
11 builder = XmlMarkup.new :target => $stdout, :indent => 2
12 builder.instance_eval &document
```

Les exemples de *Builders* présentés ici et à la section précédente utilisent tous la modification de l'objet délégué pour alléger la syntaxe du DSL. Quoique pratique et souvent utilisée, cette technique est récemment tombée en disgrâce. Le problème évoqué par la communauté Ruby (dans ce cas-ci) est la dissonance cognitive qu'il existe entre ces constructions et les véritables fermetures (Bini, 2008; the Lucky Stiff, 2008). En effet, l'utilisation de cette technique modifie les règles du jeu : l'objet délégué de la fermeture de construction n'est plus l'objet englobant mais bien le *Builder* qui a été instancié. Il est maintenant préférable de passer l'instance du *Builder* en paramètre à la fermeture de construction plutôt que de modifier l'objet délégué.

2.4.7 Définition dynamique par la super-classe

La puissance de l'héritage et son rôle dans le design objet sont connus et largement acceptés depuis longtemps. L'héritage est au centre de presque tous les vingt-et-un *Design Patterns* du *Gang of Four* (Gamma et al., 1995). Plusieurs heuristiques visant le découplage et l'augmentation de la cohésion sont basés sur une utilisation judicieuse de l'héritage des types. La pratique de l'héritage vise très souvent à configurer la super-classe, à la spécialiser. Les exemples les plus criants sont ceux des patrons *Template Method* et *Strategy*. Plusieurs *frameworks* se servent de l'héritage pour personnaliser un

comportement pré-établi.

Ainsi, l'héritage est souvent utilisé pour faire abstraction des détails d'implémentation d'une classe. Le but est d'utiliser la super-classe et non l'implémentation. Les *frameworks* fonctionnent de cette façon. Des comportements sont pré-définis dans le système. Pour personnaliser ces comportements, le programmeur écrit des classes qui héritent de celles faisant partie du *framework*. De cette manière, les instances de ces classes peuvent participer aux interactions avec le système.

Tout ceci est fantastique du point de vue des langages statiques où le type d'un objet est déterminé par sa classe. Mais qu'en est-il des langages dynamiques où le type d'un objet est donné par les messages qu'il peut recevoir ? Les langages dynamiques permettent donc de rajouter dynamiquement des méthodes à un objet. En Ruby, absolument tout est un objet. Ajouter des méthodes à une classe (une classe est objet) ou à une instance (aussi un objet) est du pareil au même.

La définition dynamique par la super-classe est une extension de l'héritage où un comportement abstrait est défini dans la super-classe et la paramétrisation se trouve dans la sous-classe. Mais n'est-ce pas là ce que l'héritage « classique » nous permet ? Lorsque plusieurs classes d'objet partagent les mêmes comportements ou état interne, une pratique courante est de pousser ces similitudes vers la super-classe. Ceci fonctionne jusqu'à un certain point. Les accesseurs (*getters* et *setters*) sont des comportements partagés par plusieurs classes d'objets. Pourtant, leur création n'est pas déléguée à une super-classe. Pourquoi ? La raison semble évidente : quoique très similaires, les accesseurs sont tous différents. Ils n'ont pas tous la même signature. Pourtant leur essence est toujours la même. Nous souhaiterions déléguer à la super-classe la création des accesseurs. Pour ce faire, la super-classe aura besoin minimalement du nom des attributs. La pratique commune en Ruby est de nommer les accesseurs en lecture et en écriture par le même nom que la variable elle-même. Voici une première tentative :

```ruby
# Ruby
class Poisson
  def couleur
    @couleur
  end
  def couleur= val
    @couleur = val
  end
end

guppy = Poisson.new
guppy.couleur = "rouge"
```

Tout ceci est assez répétitif. Chaque classe doit définir ses propres accesseurs. La redondance et la monotonie de cette activité est accablante... Une solution rapide serait de rendre publique toutes variables mais ceci violerait l'encapsulation et, de plus, est tout simplement impossible en Ruby. Puisque ce comportement se répète dans plusieurs classes, utilisons une super-classe pour définir nos attributs. Ruby permet d'appeler une méthode de la super-classe à partir de la définition de la sous-classe :

```ruby
class SuperA
  def self.accesseurs
    class_eval do
      define_method "ma_variable" do
        @ma_variable
      end
      define_method "ma_variable=" do |val|
        @ma_variable = val
      end
    end
  end
end

class A < SuperA
  accesseurs
end
```

Ici, `self.accesseurs` est une méthode de classe qui appartient à la classe `SuperA`. Cette méthode est appelée lors de l'évaluation de la définition de la classe A. Elle ajoute deux méthodes dans le contexte de la sous-classe. Deux améliorations peuvent être apportées. Premièrement, on ajoute un paramètre à la méthode `self.accesseur` qui représente le nom de la propriété. Ensuite, en bénéficiant du fait que les classes Ruby sont «ouvertes»,

on ajoute la méthode `self.accesseurs` à la classe `Object`. Ainsi, toutes les classes pourront en bénéficier.

```
1  class Object
2    def self.accesseurs attribut
3      class_eval do
4        define_method attribut do
5          @attribut
6        end
7        define_method "#{attribut}=" do |val|
8          @attribut = val
9        end
10     end
11   end
12 end

13 class Poisson
14   accesseurs :couleur
15 end

16 guppy = Poisson.new
17 guppy.couleur = "rouge"
```

En fait, ce comportement fait déjà partie du langage Ruby. La méthode se nomme `attr_accessor` et prend en paramètre une liste de nom d'attributs.

L'utilisation de la définition dynamique par la super-classe est particulièrement intéressante dans le cadre de la création de DSL internes plus techniques. Le *framework* ActiveRecord l'utilise pour définir l'appariement Objet-Relationnel de classes. ActiveRecord permet de déclarer les relations entre les classes en utilisant le mécanisme décrit. Le *framework* génère alors les méthodes et les requêtes SQL nécessaires pour assurer la persistance des données.

```
1 class Aquarium < ActiveRecord::Base
2   has_many :poissons
3 end
4 class Poisson < ActiveRecord::Base
5   belongs_to :aquarium
6 end
```

Réaliser le même exercice en Groovy nécessite un peu plus de gymnastique de programmation. Voici la version Groovy de l'exemple ci-haut :

```
1  // Groovy
2  class Aquarium extends ActiveRecord.Base {
3    static {
4      domainClass Aquarium; hasMany "poissons"
5    }
6  }
7  class Poisson extends ActiveRecord.Base {
8    static {
9      domainClass Poisson; belongsTo "Facture"
10   }
11 }
```

On utilise ici le bloc `static` pour exécuter les méthodes *domainClass*, *hasMany* et *belongsTo*. La super-classe pourrait avoir l'air de ceci :

```
1  // Groovy
2  package ActiveRecord
3  class Base {
4    static subClass
5    static domainClass(klass) {
6      subClass = klass
7      subClass.metaClass.id = new Integer()
8      subClass.metaClass.create = { ... }
9      subClass.metaClass.retreive = { ... }
10     subClass.metaClass.update = { ... }
11     subClass.metaClass.delete = { ... }
12   }
13   static hasMany(others) {
14     subClass.metaClass."$others" = []
15   }
16   static belongsTo(other) {
17     subClass.metaClass."$other" = new Object()
18   }
19 }
```

Pour ajouter des méthodes et des attributs à la sous-classe, nous utilisons sa méta-classe. L'introduction de méthodes en Groovy est relativement simple. Ici, on ajoute les méthodes CRUD. Les méthodes *hasMany* et *belongsTo* ajoutent les attributs appropriés.

* * *

Ainsi, un DSL est un langage qui exploite le jargon du domaine du problème. Le gain immédiat relié à leur utilisation est au niveau de la compréhension du problème. L'art

de la programmation ne réside pas tant dans la construction du programme que dans la détermination de ce qui doit être construit. Pour ce faire, il existe un ensemble de paradigmes nous permettant à la fois de communiquer nos idées et de traduire la réalité. La nécessité de décrire et de simuler les interactions entre des objets réels est à la base du paradigme objet. Les DSL permettent d'atteindre un niveau supérieur d'abstraction en exploitant le phrasé, le jargon, des experts du domaine d'affaire. Il est important de mentionner que malgré le fait qu'un DSL utilise les termes propres aux experts, le but n'est pas de les faire programmer. Faire programmer les experts du domaine d'affaire était l'un des buts du langage COBOL (Sammet, 1981). Le succès de ce côté de COBOL est discutable... Ce que nous recherchons plutôt avec un DSL est la possibilité d'exprimer plus fidèlement la réalité. Un code source écrit à l'aide d'un DSL bien conçu peut être compris par l'expert du domaine. Ceci ne rend pas obsolète le métier de programmeur. En fait, c'est tout à fait le contraire. L'élaboration d'un DSL interne ou externe requiert la connaissance et la maîtrise de techniques de programmation avancées telles la méta-programmation et la construction d'interpréteurs. La connaissance des patrons décrits précédemment est nécessaire mais non suffisante à la conception d'un DSL efficace.

CHAPITRE III

MISE EN SITUATION

Dans ce chapitre, nous situons le contexte du domaine d'affaire. Nous discutons du type d'applications qui sont développées, des frameworks *et des librairies qui sont utilisés. L'infrastructure logicielle est principalement basée sur la plateforme Java. Nous traitons des décisions qui ont menées à l'utilisation du langage Groovy et du framework web Click. Nous regardons aussi les interactions logicielles avec le système de données patrimonial AS/400. Finalement, nous identifions les endroits qui pourraient bénéficier de l'utilisation d'un DSL.*

3.1 Description du domaine d'affaire

L'utilisation du web pour simplifier et automatiser certaines tâches est omniprésente. En 2008, on ne s'étonne plus de retrouver des applications web dans plusieurs domaines. Plusieurs métaphores d'interaction avec l'utilisateur sont utilisées. La plus commune de celles-ci est sans aucun doute le formulaire. Il est pratiquement impossible d'utiliser un service internet sans remplir un formulaire. C'est le moyen privilégié par les organisations souhaitant obtenir de l'information de leurs utilisateurs. Une raison possible pour l'omniprésence des formulaires web est la familiarité des utilisateurs avec leur équivalent « papier ». Cette similitude est aussi la source des pires horreurs en ce qui concerne le design de formulaires web.

Toujours est-il que le formulaire web est partout. On l'utilise lors d'inscription à un service, de paiements, de recherches, de l'ouverture d'une session... La rédaction d'un courriel est, en réalité, ni plus ni moins que la complétion d'un formulaire.

On peut répertorier deux familles d'utilisations possibles d'un formulaire. La première est caractérisée par le besoin d'obtenir l'information d'un utilisateur à un moment précis du déroulement d'un processus métier. Ce processus peut être complexe ou très simple. L'ouverture d'une session à son service de courriel internet en est un exemple. Des exemples plus complexes de gestion de processus métier requièrent des outils parfois plus sophistiqués que le simple formulaire HTML. Prenons par exemple le processus d'approbation d'une feuille de temps. Le consultant doit d'abord saisir son temps dans un formulaire. Lorsque la période de facturation est terminée, notre consultant soumet le formulaire pour approbation. Une notification est envoyée à son supérieur immédiat et aux chargés de projets respectifs. Ceux-ci consultent et modifient l'information à l'aide de formulaires supplémentaires. Cette nouvelle information permet de diriger le processus métier. Ici, les formulaires sont des intrants à des processus métiers et/ou génèrent des événements captés par ces processus. Orbeon est un cadre applicatif permettant d'utiliser les formulaires de cette façon. Il exploite les capacités de la norme XForms, des pipelines de transformations XSLT et un modèle de programmation événementiel pour bâtir des processus métiers. Une telle approche vise à remplacer l'approche traditionnelle. Celle-ci est basée sur une utilisation conjointe de répertoires partagés, de documents Word ou Excel, de courriels et de « base de données » Access.

La seconde famille d'utilisation des formulaires dans une application est directement calquée sur leur version « papier ». Le formulaire traditionnel (celui en papier) est habituellement un savant mélange de questions, de conséquences, de sections optionnelles ou obligatoires et d'instructions particulières. On les retrouve partout où un individu doit s'enregistrer, fournir de l'information personnelle ou formaliser une demande quelconque à un organisme. Avec la première forme d'utilisation d'un formulaire, le processus métier est extérieur au formulaire. Ici, l'inverse s'applique : le processus est internalisé. Le formulaire se transforme dynamiquement selon les réponses et les actions de l'utilisateur. Ce type de formulaire est facilement repérable et son utilisation est beaucoup plus répandue. On le détecte dès que l'on retrouve une instruction semblable à : « Si vous avez répondu "oui" à la question 2B ... » Cette façon de faire est appropriée aux formulaires

papier. Les émetteurs du formulaire se fient sur la logique et l'intelligence de l'appliquant pour que celui-ci répondent correctement et de façon cohérente aux questions.

Évidemment, ceci n'est pas toujours le cas. L'appliquant peut répondre incorrectement, ne pas répondre aux bonnes questions ou à trop de questions. Un certain cynisme peut se développer chez les responsables du traitement manuel des formulaires vis-à-vis les appliquants. Entre en scène le processus d'informatisation des formulaires papiers. Le remplacement d'un traitement manuel par une application informatique exige un certain degré d'automatisation. L'entrée manuelle des données exige un temps considérable au personnel affecté. Un des buts de l'automatisation des processus manuels est donc l'élimination du temps nécessaire à la saisie des données. Toutefois, cette activité, quoique répétitive et monotone, requiert parfois que l'exécutant ait des connaissances particulières du domaine d'affaire. Cette personne doit déceler et gérer les cas d'exception, les erreurs, les valeurs invalides ou impossibles, les formulaires incomplets, etc.

L'informatisation bête d'un formulaire est une tâche à la portée de tout développeur. Les choses se compliquent considérablement lors de l'automatisation du processus de validation de ce formulaire. Comment traiter le cas où, l'utilisateur ayant répondu telle ou telle valeur à cette question, doit répondre à telle autre ? Comment valider sa prise de connaissance d'instructions importantes ? Comment valider que tel champ de saisie contient bel bien un numéro de téléphone, par exemple, et non un numéro d'assurance sociale ? Comment valider qu'un appliquant n'a pas déjà effectué une demande ?

Certaines de ces considérations peuvent être exprimées par des expressions régulières, des vérifications avec une base de données, etc. La question du chemin parcouru par l'utilisateur à l'intérieur même du formulaire reste entière. Le formulaire doit donc tracer le chemin à parcourir au fil des réponses de l'utilisateur. Le développeur doit représenter le formulaire non pas comme une longue suite de questions mais plutôt comme une machine à états. Chaque état est représenté par un formulaire. Les réponses contenues dans un formulaire permettent de transiter vers un autre. Cette machine à états se complique lorsque l'application permet à l'utilisateur de revenir sur ses pas et de modifier

les réponses fournies. L'utilisation de formulaires décrite ici est souvent qualifiée de *Wizard*. Un *Wizard* est une machine à états déterministe où les états sont des formulaires et les transitions sont les réponses fournies.

Le domaine d'affaire qui nous intéresse est truffé de formulaires web de type *Wizard*. Plusieurs applications de saisie de données doivent être créées et ce, en peu de temps. La navigation entre les états dépend des réponses saisies ainsi que des données externes (date et heure du jour, état et situation de l'utilisateur dans le système, etc.) Les formulaires n'ont rien d'extravagants et exploitent les artefacts web habituels : champs de saisie texte, listes déroulantes et cases à cocher mutuellement exclusives ou non. Certains des formulaires papier qui sont transformés en formulaires web contiennent plusieurs dizaines de pages. Les pages HTML résultantes sont relativement complexes. Les applications web développées ne sont en ligne que pour quelques semaines par année. Par contre, les *wizards* doivent être très flexibles car des modifications sont apportées aux formulaires à chaque mise en production.

L'arrière boutique et le système de gestion de données des applications sont assurés par un système AS/400. La majorité des règles d'affaire y sont hébergées sous la forme de procédures stockées écrites en RPG. La réimplémentation de ces règles vers d'autres langages de programmation n'est pas une issue possible étant donné les contraintes imposées aux projets de développement. Un tel effort de refonte de système n'apporterait aucun gain à court ou moyen terme. De plus, les développeurs « résidents » ne possèdent pas tous une connaissance approfondie des langages de programmation modernes et en vogue. Un travail de formation considérable devrait être entrepris avant que l'équipe de maintenance puisse prendre les rênes. On préférera donc une approche visant à encapsuler les accès au système de données dans l'attente que ce dernier sera un jour remplacé.

Les applications web développées sont, par contre, les premières applications externes à devoir interfacer avec le système de données existant. Par « externes », on souhaite mettre l'accent sur le fait que les applications « internes » sont des écrans de saisies de type « terminal » directement rattachés au système de gestion de données et aux

programmes RPG. Il est donc nécessaire de trouver un moyen efficace de faire appel aux programmes RPG à partir de l'environnement dans lequel sera exécuté les applications web. Nous souhaitons aussi exploiter au maximum le modèle d'interaction avec le SGBD par des appels à des procédures stockées. Plusieurs circonstances nous mènent à ce constat : tel que discuté ci-haut, la réimplémentation n'est pas une solution viable ; on souhaite éviter la distribution de la logique d'affaire à travers les applications ; les nouvelles applications ne doivent être que des interfaces supplémentaires au SGBD. Pour toutes ces raisons, l'appel d'une procédure stockée sera une activité récurrente dans le déroulement de l'application. Il est donc important que l'appel soit effectué de façon simple, facile, flexible et efficace.

3.2 Description de l'environnement de développement

Le contexte de développement de nos applications web est régi par plusieurs facteurs : l'environnement d'exécution, les langages de programmation utilisés, le *framework* de développement web et le modèle d'interaction avec le système de gestion de données.

Langages de programmation

Aujourd'hui, l'environnement d'exécution est principalement désigné par la machine virtuelle sur laquelle seront développées les applications. Il est important de faire la distinction entre la machine virtuelle et le langage qui est utilisé pour rédiger les programmes. En ce sens, la JVM n'est pas Java et vice versa. La JVM est, à la base, un interpréteur de code octet, un langage assembleur pour une architecture machine à pile. Un programme écrit à l'aide du langage Java peut être (pré)compilé en *bytecode* puis interprété par la JVM. Au-dessus de 240 langages pour la JVM ont été répertoriés (Tolksdorf, 2008). Parmi ceux-ci, on retrouve quelques grands classiques comme Java (le langage éponyme), Lisp, Lambda Calculus, Prolog, Eiffel et Smalltalk. Plusieurs langages plus ou moins récents sont réimplémentés afin de fonctionner sur la JVM. Ruby, Python et JavaScript possèdent tous leur compilateur de *bytecode* (respectivement JRuby, Jython et Rhino). De plus, plusieurs nouveaux langages sont créés ayant comme objectif premier d'être

intégré avec la machine virtuelle Java. Groovy, Scala et AspectJ sont quelques exemples intéressants de ceci.

Ainsi, le choix d'une machine virtuelle peut se faire presque indépendamment du choix du langage de programmation. Dans plusieurs cas, il est possible de choisir l'un et l'autre séparément. À titre d'exemple, le langage Ruby est implémenté de plusieurs façons. En plus de l'implémentation officielle faite en C (MRI - *Matz's Ruby Interpreter*), un programme Ruby peut aussi être exécuté sur la machine virtuelle Java (JRuby), dans l'environnement .NET (IronRuby) et même sur la machine virtuelle de Smalltalk (MagLev).

Plusieurs arguments politiques ainsi que technologiques ont amené les décideurs à privilégier la plateforme de développement de Java pour nos projets. Deux langages sont utilisés pour programmer les applications : Java et Groovy. Nous passerons sous silence les avantages et les inconvénients du langage Java. Ceux-ci sont et seront débattus encore longtemps et ce, même après sa disparition du paysage de développement informatique contemporain. Groovy est un langage doté d'un système de types dynamiques, fortement typé, compatible avec Java, pouvant être interprété ou compilé vers le *bytecode* Java. Groovy est donc un langage dynamique fait spécifiquement pour la machine virtuelle Java. Tout code source Java est automatiquement du code source Groovy. Ce dernier n'enlève rien à Java. Il ajoute un niveau de simplicité et d'agilité sur la base très solide de la JVM.

Typage dynamique Quoique fortement typé, Groovy permet de déterminer le type d'un objet lors de l'exécution du programme plutôt que lors de sa compilation. Sans revenir sur le débat et l'explication « statique versus dynamique », rappelons que le type dynamique d'un objet est donné par les méthodes de cet objet. Il ne faut pas oublier que Groovy *ajoute* des capacités au langage Java. Il est donc possible de typer de façon statique les objets.

Fermetures natives Une fermeture est un bloc de code anonyme ou non qui est traité par le langage comme des objets de premier ordre (*first-class objects*). Java permet

une forme de définition de fermetures grâce aux classes anonymes internes (*anonymous inner classes*). Seules les variables déclarées *final* peuvent être utilisées à l'intérieur de ces fermetures à la saveur Java. Quoiqu'elles ont la possibilité d'éliminer la duplication de code et d'améliorer sa maintenabilité, l'utilisation des classes anonymes internes est perçue comme étant une activité complexe et rébarbative qui tend à rendre le code illisible (Beck, 2008).

Initialisation littérale Groovy permet l'initialisation littérale de listes, de dictionnaires et d'expressions régulières. En Java, seuls quelques types bénéficient d'une déclaration littérale. Il est effectivement possible de définir directement des chaînes de caractères et des valeurs numériques à même le code (i.e. `String s = "une valeur littérale";`). Il est pas contre impossible de déclarer des listes et des dictionnaires de cette façon. Par exemple, pour initialiser une liste avec les nombres de 1 à 5 en Java, on doit instancier un objet de type *List*, construire une boucle et envoyer successivement le message *add* à l'objet *List*. Groovy procure une syntaxe d'initialisation littérale des listes et des dictionnaires. La même liste en Groovy pourrait être définie ainsi : `liste = [1,2,3,4,5]`.

Métaprogrammation Groovy utilise et simplifie l'API d'introspection déjà présente dans Java. Il procure la possibilité d'ajouter dynamiquement des méthodes dans une classe. Malheureusement, le langage n'offre pas de mécanismes spécialisés pour l'ajout de méthodes à un objet en particulier. Ceci est toutefois possible via des techniques ad hoc (voir figure 3.1). Groovy dispose aussi de quelques intercepteurs (*hooks*) permettant d'attraper les appels vers des méthodes indéfinies ou non.

Ajout de méthodes Groovy bonifie plusieurs types standards de Java en y ajoutant des méthodes. Ces méthodes sont souvent inspirées des techniques de programmation fonctionnelle. La possibilité de facilement passer des fermetures en paramètres le permet.

Il n'est pas surprenant d'apprendre que le projet Groovy a remporté le *JAX Innovation Award 2007*. Pour toutes ces raisons, Groovy est le premier choix de langage de programmation pour nos applications.

```
1  class Poisson {
2    def __singletons = [:]
3    def methodMissing(String name, args) {
4      def method = __singletons[name]
5      if (method)
6        return method(args)
7      else
8        throw new MissingMethodException(name, this.class, args)
9    }
10 }

11 guppy = new Poisson()
12 betta = new Poisson()

13 guppy.__singletons["parle"] = { println "blub blub..." }

14 guppy.parle() // => "blub blub..."
15 betta.parle() // => Exception lancée !
```

Fig. 3.1 Ajout de méthodes à un objet en particulier en Groovy

Couche de présentation

Le choix de la couche de présentation web n'est pas une décision simple à prendre. Le *framework* retenu structure le développement et impose un ensemble de patrons et d'idiomes. Le patron adopté par la plupart des *frameworks* est le *Model-View-Controller* (Fowler, 2003). MVC a été établi lors du développement d'applications utilisant une interface utilisateur graphique. Dans une application GUI, la *Vue* est présentée à l'utilisateur. Elle contient une représentation de l'état actuel du *Modèle* ainsi que les actions possibles. Ces actions sont transmises de la *Vue* vers le *Contrôleur*. Celui-ci modifie l'état du *Modèle*. Le nouvel état du *Modèle* est transmis de celui-ci vers la *Vue*.

Le patron MVC a été repris par les premiers concepteurs d'interfaces web. Dans ce contexte, la *Vue* est habituellement une page HTML. Le *Contrôleur* est une interface programmatique à une application. Il contient les appels métiers. On parlera ici d'un *Contrôleur Frontal* (Fowler, 2003). Le *Modèle* est un réseau d'objets pouvant être persisté dans une base de données. Dans le monde Java, la *Vue* est souvent incarnée par des pages JSP/JSF et le *Contrôleur* par des *Servlets*. Ce patron est habituellement nommé MVC2. Ainsi, MVC2 prend une application et l'encapsule dernière un *Contrôleur*. Des

gabarits HTML actifs sont responsables de traduire les actions utilisateurs en appels au *Contrôleur Frontal*.

Une seconde façon de concevoir une application web est en considérant non pas ses entrailles (une interface HTML à une application déjà existante) mais sa forme finale (un ensemble de pages HTML). Au lieu de disposer d'un unique *Contrôleur Frontal* pour toute l'application, chaque page dispose de son propre *Contrôleur*. Le *Contrôleur Page* (Fowler, 2003) n'expose pas des actions métiers. Les méthodes « publiques » d'un *Contrôleur Page* sont directement rattachées au protocole HTTP. On y retrouve donc des méthodes comme `doGet` et `doPost` à la manière d'un *Servlet* Java. Les *Contrôleurs Page* sont ainsi responsables de recevoir les actions HTTP, d'interagir avec le modèle de données, de produire une vue HTML et de l'expédier au client.

Le *framework* retenu pour développer les applications web de type formulaire est « Click ». Click est un *framework* Java exploitant le patron *Contrôleur Page*. La génération des vues se fait à l'aide de gabarits. Ceux-ci sont similaires à des gabarits JSP/JSF. Click utilise le moteur Velocity pour générer les pages HTML qui représenteront les *Vues* du *Modèle*.

La philosophie de conception de Click s'apparente à celle du *framework* maintenant célèbre *Ruby On Rails* : *Convention plutôt que Configuration*. Le niveau de complexité de *frameworks* tels que *Tapestry* ou *Struts* provient en partie de la quantité impressionnante de personnalisation nécessaire à leur fonctionnement. Absolument tout doit être configuré via des fichiers XML ou programmatiquement. Exploiter le principe de « Convention plutôt que Configuration » implique la présence d'un ensemble de normes et d'opinions par rapport à la manière que le *framework* doit être utilisé. Par exemple, Maven, un outil permettant de gérer le cycle de construction d'un logiciel, est un tel *framework*. Maven force les développeurs à utiliser une certaine structure de répertoires pour les projets, impose une convention de nommage pour les *packages* Java, les noms de modules et les artefacts logiciels. Des règles aussi strictes semblent limiter les développeurs dans leurs mouvements et donc, leurs programmes.

Pourtant, le chemin tracé par ces *frameworks* est plus libérateur qu'il ne le laisse paraître. Click impose certaines conventions. En voici quelques-unes :

- Le nom des gabarits Velocity doit être similaire au nom des *Contrôleurs Page*. Par exemple, si la classe du contrôleur d'une page d'accueil se nomme « AccueilPage », le nom du gabarit doit être « accueil.htm ».
- La structure des *packages* Java contenant les contrôleurs doit être imitée par la structure des répertoires contenant les gabarits Velocity.
- Si l'on souhaite qu'une variable soit disponible à l'intérieur d'un gabarit Velocity, cette variable doit avoir une visibilité « public ».

La plupart de ces conventions peuvent être contournées. Toutefois, le coût de ne pas suivre « le chemin pavé de briques jaunes » peut être élevé. On applique alors le principe de Pareto : les conventions imposées permettent d'implémenter 80% des fonctionnalités avec 20% de l'effort habituellement requis. Ainsi, les conventions, loin d'être limitatives, libère le programmeur d'une foule de considérations accidentelles. Ceci lui permet de se concentrer sur l'essence du problème et de la solution.

Voyons un exemple très simple d'une page Click. La classe d'un *Contrôleur Page* doit héritée du super-type « Page ». Le gabarit de le *Vue* de ce *Contrôleur* aura accès aux champs publics de ce dernier. Tout d'abord, le code Java du *Contrôleur* :

```
1 package examples.page;

2 import net.sf.click.Page;

3 public PoissonsPage extends Page {
4   public String couleur = "rouge";
5 }
```

Le comportement par défaut d'un *Contrôleur Page* répond à un GET HTTP en générant la vue. Le gabarit de la *Vue* pourrait ressembler à ceci :

```
1 <html>
2   <body>
3     Le poisson est $couleur !
4   </body>
5 </html>
```

Lorsqu'un message GET est envoyé à l'URL de la page, le *Servlet* central de Click crée une instance de la classe « PoissonsPage ». L'instance est insérée dans un flot d'exécution qui appellera ses méthodes. Finalement, la *Vue* sera générée et la page HTML résultant renvoyée au fureteur client.

L'une des forces de Click est son architecture permettant de facilement construire et gérer des formulaires web. Click est distribué avec un ensemble de classes représentant, à différents degrés d'abstraction, des *widgets* de formulaires web. Parmi ceux-ci, on retrouve entre autres « Form », « Button » et « TextField ». Les *widgets* de Click sont à la fois responsables de leur affichage, de leur validation et de leur gestion interne. Voyons comment construire un formulaire très simple en Click :

```
1  public class FormulairePage extends Page {
2    public Form form = new Form();
3    public String message;

4    public FormulairePage() {
5      form.add(new TextField("couleur", true));
6      form.add(new Submit("OK"));
7    }

8    public void onPost() {
9      if (form.isValid()) {
10       message = "La couleur est " +
11           form.getFieldValue("nom");
12     }
13   }
14 }
```

Le comportement de cette application est assez simple : présenter à l'utilisateur un formulaire contenant deux *widgets*, un champs de texte obligatoire et un bouton. Lorsque le bouton est cliqué, un POST est envoyé au serveur. La valeur du champs de texte est récupéré et affiché dans la page HTML résultante. Deux variables sont publiques : le formulaire lui-même ainsi qu'un message informatif. L'objet « Form » est un conteneur où l'on ajoute des composantes. La méthode « onPost » est réécrite. On y vérifie la validité du formulaire. Dans l'affirmative, la valeur du message est modifiée. Dans ce cas-ci, le formulaire est valide si le champs « couleur » contient une valeur. Il serait possible de limiter la validité de ce champs par rapport à la longueur de la valeur contenue ou aux

caractères qui la composent. Puisque chaque composante est responsable de son affichage, l'appel à la méthode « toString() » du formulaire génère tout le code HTML nécessaire à son fonctionnement. On dira alors que le gabarit HTML est en mode « mise-en-page automatique » (*auto-layout*). Le gabarit HTML est d'une simplicité désarmante :

```
1 <html>
2 <head>$cssImports</head>
3 <body>
4   $form
5   $!message
6 </body>
7 </html>
```

Le code HTML généré contient la mise en page des composantes ainsi que celle des erreurs de validation. La mise en page « automagique » fait partie du « chemin pavé de briques jaunes » de Click discuté précédemment. Quoiqu'il est possible de faire entièrement la mise en page manuellement, le temps gagné par l'utilisation de *l'auto-layout* est un argument de poids lors du choix d'un *framework* de développement web. De plus, puisque Click est un *framework* Java, il est donc possible de programmer une application Click avec la langage Groovy. Ces deux arguments suffisent pour justifier l'utilisation de Click.

Couche métier

Les systèmes patrimoniaux de l'organisation assurent la persistance des données à l'aide d'un système AS/400. IBM a développé des pilotes JDBC permettant d'interfacer avec ce type de système. La couche métier peut donc être entièrement développée en Groovy ou en Java. Il est même possible d'utiliser les *frameworks* d'appariement relationnel-objet (ORM) comme Hibernate ou iBATIS. Le développement de la couche métier en Java/Groovy a été envisagé et même implémenté lors du développement d'un projet antérieur. Pour plusieurs raisons, cette implémentation n'a pas été un succès. Il faut tout d'abord noter que presque toute la logique de l'organisation a déjà été implémentée. Des procédures RPG/400 stockées dans le système de persistance assurent le respect des règles d'affaire. Au fil du temps, ces procédures sont devenues fortement couplées à toutes les règles d'affaire internes de l'organisation. Extraire la logique du système

```
<pcml version="4.0">
  <struct name="params-sortie">
    <data name="un-entier" type="packed"
          length="5"  precision="0" usage="inherit"/>
    <data name="une-string" type="char"
          length="50" usage="inherit"/>
  </struct>
  <struct name="params-entree">
    <data name="un-float" type="packed"
          length="11" precision="2" usage="inherit"/>
    <data name="un-booleen" type="char"
          length="1" usage="inherit"/>
  </struct>
  <program name="PROGRAMME" path="/PROGRAMME.PGM">
    <data name="params-entree"
          type="struct"
          struct="params-entree"
          usage="input"/>
    <data name="params-sortie"
          type="struct"
          struct="params-sortie"
          usage="output"/>
  </program>
</pcml>
```

Fig. 3.2 Exemple d'un document PCML

demanderait donc de repenser en profondeur les liens qui unissent les systèmes informatiques actuels. Des interfaces graphiques ont aussi été développées pour les utilisateurs internes. Ces interfaces sont directement reliées aux procédures stockées et au reste du système. Elles font partie des outils que les utilisateurs ont besoin pour accomplir leurs tâches quotidiennes. Encore une fois, lorsque la logique d'affaire est extraite des procédures stockées, les utilisateurs ne peuvent plus utiliser leurs outils. Il en résulte une perte de productivité possible et un mécontentement assuré.

Pour toutes ces raisons, il nous ait apparu important de laisser tomber cette « réinvention de la roue » et d'appeler directement les procédures stockées à partir des *Contrôleurs Page*. La couche métier des applications web est donc très mince. La logique est entièrement déléguée aux procédures stockées. Ainsi, l'application ne contient qu'une mince couche métier formée presque uniquement de DAO.

Le problème de l'appel de ces procédures reste entier. Les procédures RPG/400 sont

rendues disponibles sur le réseau via un protocole RPC. En plus du pilote JDBC, IBM a aussi développé une interface RPC aux procédures stockées. La signature d'un programme RPG distribué est décrite à l'aide d'un document PCML (pour *Program Call Markup Language*). Un document PCML décrit :

1. le nom du programme ;
2. son emplacement sur le système ;
3. et, pour chaque paramètre :
   (a) sa position dans la liste de paramètres ;
   (b) sa longueur maximale ;
   (c) son « usage » (en entrée et/ou en sortie) ;
   (d) son nom ;
   (e) son type ;
   (f) son nombre d'occurrence(s).

La sérialisation de l'appel et de la réponse n'ont rien de mystérieux. Toutes les données d'un message sont concaténées et envoyées sur le fil. Lors de la désérialisation, l'extraction des valeurs se fait avec des sous-chaînes déterminées à l'aide de la position de début de chaque valeur ainsi que de sa longueur. C'est un système simple mais efficace. Par contre, l'ajout, le retrait ou la simple réorganisation des paramètres devient une expérience périlleuse. La figure 3.2 illustre un exemple de document PCML.

L'interface d'IBM fonctionne de façon assez directe. Un objet « Document » est créé à partir d'un fichier PCML. Des appels successifs à des méthodes *setters* permettent d'attribuer une valeur à chaque paramètre utilisé en entrée. Le programme est ensuite appelé. L'extraction des valeurs de retour est fait en appelant des méthodes de type *getter* sur l'objet « Document ». Voici un extrait du code nécessaire pour effectuer l'appel au programme décrit à la figure 3.2 :

```
1  AS400 cnx = new AS400(systeme, username, password);
2  ProgramCallDocument doc = new ProgramCallDocument(
3      cnx, "chemin/vers/document.pcml");
4  doc.setValue(
5      "PROGRAMME.params-entree.un-float", 123.12);
6  doc.setValue(
7      "PROGRAMME.params-entree.un-booleen", "0");
8  doc.callProgram("PROGRAMME");
9  int unEntier = doc.getValue(
10     "PROGRAMME.params-sortie.un-entier");
11 String uneString = doc.getValue(
12     "PROGRAMME.params-sortie.une-string");
```

3.3 Zones d'améliorations possibles

L'utilisation conjointe de Click, Java et des procédures stockées RPG/400 se fait relativement bien. Les programmes réalisés avec la recette décrite dans les sections précédentes sont simples et peu complexes. Par contre, plusieurs améliorations pourraient être apportées.

Malgré tous les avantages de Click, la création de formulaires est laborieuse. Il est difficile de distinguer la structure d'un formulaire d'après les déclarations. L'initialisation des propriétés des composantes du formulaire se fait sur plusieurs lignes et alourdit inutilement le code. Un formulaire modeste nécessite plusieurs dizaines de lignes de code. La modification, le débogage et la maintenance d'un formulaire Click s'avèrent être un processus délicat.

Les requêtes vers les procédures stockées sont aussi problématiques. L'exemple ci-haut montre un appel contenant seulement deux valeurs en entrée et deux autres en sortie. Les cas réels sont bien différents. La présence de plus de quarante paramètres n'est pas chose rare. De plus, le type des données qui sont envoyées ou reçues n'est généralement pas compatible avec les types Java attendus. Par exemple, le type booléen n'existe pas dans le système de données. Une donnée booléenne est représentée par le caractère « O » ou « N ». Il en va de même pour les types de données numériques. Ainsi, en plus de

devoir supporter une interface relativement lourde et encombrante, le programmeur doit traiter tous les cas de conversion de type de données les plus triviaux.

* * *

Les défis propres à ce domaine d'affaire ne sont pas de l'ordre de la modélisation mais bien techniques. Les mécanismes de construction de formulaires et de communication avec le système patrimonial sont encombrants et empêchent les développeurs de se concentrer sur l'essence du problème. Ils doivent composer avec des librairies qui méritent grandement d'être simplifiées. L'une des améliorations possibles est l'introduction de comportements par défaut pour les cas les plus communs. Les mécanismes actuels offrent un degré de paramétrisation beaucoup trop élevé. Ceci implique que les cas les plus simples comme les plus complexes requièrent le même nombre de paramètres et de fichiers de configuration. Ainsi, le niveau d'effort nécessaire ne varie pas en fonction de la complexité de l'implémentation. Nous cherchons à élaborer des DSL qui nous permettraient d'implémenter 80% des cas avec 20% de l'effort nécessaire. Une seconde amélioration est au niveau de la structure du code de construction d'un formulaire. La structure actuelle ne communique pas du tout l'intention du programmeur en ce qui concerne les agencements des composantes.

CHAPITRE IV

AMÉLIORATION PAR L'UTILISATION DE DSL

Suite aux constatations faîtes au chapitre précédent, nous utilisons les patrons de conception décrits au chapitre 2 et les techniques de méta-programmation du chapitre 1 pour construire deux DSL internes. Nous voyons d'abord comment transformer l'API de construction de formulaires actuel en Builder dynamique. Finalement, nous construisons un Builder statique permettant de décrire la signature d'un appel RPG/400. En plus de simplifier la syntaxe, ce second DSL interne y incorpore des mécanismes de conversions par défaut. De plus, nous voyons qu'il est possible de donner une saveur «déclarative» à un API programmatique à l'aide d'un Builder.

4.1 Construction de formulaires web

Le patron de conception « Builder » s'applique tout naturellement lors de la construction de structures hiérarchiques. De telles structures n'existent pas uniquement dans des documents XML. Un formulaire web est une structure hiérarchique. La racine de l'arbre est un conteneur de type « Formulaire ». On peut y ajouter d'autres conteneurs ou différents *widgets* comme des boutons, des boîtes à cocher ou des listes déroulantes.

Imaginons d'abord un formulaire simple. Il contiendra un encadré avec un titre. À l'intérieur de cet encadré, un paragraphe de texte sera disposé au-dessus d'un groupe de boutons mutuellement exclusifs. Des boîtes de saisies de texte suivront. Finalement, dans le bas de l'encadré, on trouvera une liste déroulante. Un bouton permettant l'envoi du formulaire se trouvera en bas de l'encadré.

Voyons d'abord la version Java de ce formulaire Click. Pour simplifier la lecture du

code, nous avons enlevé les lignes qui n'ajoutent rien à sa compréhension. Le patron est toujours le même :

(a) Création d'une composante graphique ;

(b) Configuration de la composante ;

(c) Si la composante est un conteneur

  (i) La nouvelle composante devient la composante parent ;

  (ii) Le patron reprend récursivement à l'étape (a) ;

(d) Ajout de la nouvelle composante à la composante parent.

Le code Java nécessaire est illustré à la figure 4.1. En suivant le principe qu'une tâche simple ne devrait pas être compliquée à réaliser, il est raisonnable de penser que la construction d'un formulaire web ne devrait pas prendre autant de lignes de code. La métaphore de l'API utilisée est calquée sur celle de l'API DOM. Le *Document Object Model* permet à un programmeur de définir un arbre de noeuds en les emboîtant les uns dans les autres. Chaque noeud de l'arbre doit être configuré manuellement. Son lien filial vers un noeud dit « parent » doit aussi être explicité à l'aide de l'appel d'une méthode. Les mêmes techniques qui permettent de simplifier la création de documents XML s'appliquent dans le cas présent.

La construction *bottom-up* d'un DSL doit commencer par un API. Dans le cas d'un DSL servant à créer un formulaire Click, il est question de construire un API par-dessus l'API existant. La figure 4.2 illustre l'utilisation souhaitée du DSL.

Dans la construction de structures hiérarchiques, trois aspects doivent être pris en considération : la construction d'un noeud, sa configuration et sa position dans l'arbre. Un API comme DOM ou celui de Click utilise l'instanciation d'objets et l'appel de méthodes pour réaliser ces trois buts. Dans le cas de notre DSL, nous utilisons un appel de méthode pour construire un objet et en faire la configuration. L'imbrication indique le lien de parenté entre deux noeuds. On voit donc clairement que l'encadré est à l'intérieur du formulaire et que les boutons « radio » sont dans le « radioGroup ». Le nom des méthodes appelées correspond au nom de la classe du *widget* qui doit être instancié.

```
1  Form form = new Form("form");
2  FieldSet encadre = new FieldSet("encadre");
3  encadre.setLegend("Titre de l'encadré");
4  Label paragraphe = new Label("Texte du paragraphe...");
5  encadre.add(paragraphe);
6  RadioGroup groupe = new RadioGroup("groupe");
7  groupe.setRequired(true);
8  Radio choixUn = new Radio();
9  choixUn.setValue("choixUn");
10 choixUn.setLabel("Premier choix");
11 groupe.add(choixUn);
12 Radio choixDeux = new Radio();
13 choixDeux.setValue("choixDeux");
14 choixDeux.setLabel("Second choix");
15 groupe.add(choixDeux);
16 encadre.add(groupe);
17 TextField champTexte = new TextField("champTexte");
18 champTexte.setLabel("Entrez du texte");
19 champTexte.setSize(30);
20 champTexte.maxLength(30);
21 champTexte.setRequired(true);
22 encadre.add(champTexte);
23 Select listeDeroulante = new Select("listeDeroulante");
24 listeDeroulante.setLabel("Faîtes un choix");
25 Option itemUn = new Option("1", "Item 1");
26 listeDeroulante.add(itemUn);
27 Option itemDeux = new Option("2", "Item 2");
28 listeDeroulante.add(itemDeux);
29 Option itemTrois = new Option("3", "Item 3");
30 listeDeroulante.add(itemTrois);
31 encadre.add(listeDeroulante);
32 form.add(encadre);
33 Submit bouton = new Submit("bouton");
34 bouton.setLabel("Soumettre le formulaire");
35 form.add(bouton);
```

Fig. 4.1 Construction d'un formulaire Click en Java

```
1  form = new Builder().form (name:"form") {
2    fieldSet (name:"encadre", legend:"Titre de l'encadré") {
3      label "Texte du paragraphe"
4      radioGroup (name:"groupe", required:true) {
5        radio (value:"choixUn", label:"Premier choix")
6        radio (value:"choixDeux", label:"Second choix")
7      }
8      textField (
9          name:"champTexte", label:"Entrez du texte",
10         length:30, required:true)
11     select (
12         name:"listeDeroulante", label:"Faîtes un choix",
13         options:["1":"Item 1", "2":"Item 2"])
14   }
15 }
```

Fig. 4.2 Construction d'un formulaire Click avec un DSL

Chacune de ces méthodes prend un paramètre. Il s'agit d'un dictionnaire où les clés correspondent au nom du *setter* à appeler.

Groovy offre une foule d'outils propres au langage nous permettant d'implémenter ce DSL. Tout d'abord, le MOP de Groovy permet d'intercepter des appels à des méthodes non-définies. Nous utiliserons donc cette propriété pour capturer les appels aux méthodes. Elles serviront à instancier les composantes graphiques. Lorsqu'un dictionnaire est passé en paramètre à un constructeur, Groovy appelle, pour chaque item du dictionnaire, la méthode « nouvelObjet.set<Clé>(<Valeur>) ; ». Cette particularité du langage nous permettra de configurer les composantes graphiques via un dictionnaire. En Groovy, il est possible de passer des blocs de code à une fonction. On réalise ceci en définissant le bloc de code après l'appel de la méthode et en l'encadrant d'accolades. Ainsi, chaque imbrication est passée en paramètre à la fonction qui crée le noeud parent. Finalement, puisque que Groovy permet de modifier le délégué, il n'est pas nécessaire de préciser le nom de l'objet sur lequel sont invoquées les méthodes dynamiques. Le pseudo-code expliquant le fonctionnement du Builder va comme suit :

(a) Capturer l'appel de la méthode inconnue.

(b) Transformer le nom de la méthode en nom de classe.

(c) Instancier un nouvel objet de cette classe en passant en paramètre au constructeur

le dictionnaire reçu.

(d) Si un bloc de code est passé en paramètre :

(i) Sauvegarder le noeud courant sur la pile.

(ii) Le nouvel objet devient alors le noeud courant.

(iii) Modifier le délégué du bloc de code.

(iv) Exécution récursive du bloc de code.

(v) À la fin de l'exécution, on dépile le noeud courant.

(e) On appelle la méthode « noeudCourant.add(nouvelObjet) ; »

La figure 4.3 illustre une implémentation possible de cette description.

4.2 Appel de procédures stockées

Un langage spécifique est une interface offrant un niveau d'abstraction plus élevé que le précédent. Ainsi, tout API peut être considéré comme un DSL à part entière. Une façon intéressante de procéder à l'élaboration d'API est par raffinements successifs.

La figure 3.2 montre l'utilisation de la première interface au système d'appels de procédures stockées. Son utilisation est quelque peu redondante et n'offre aucune facilité en ce qui concerne la transformation des données en entrée ou en sortie. Il est effectivement souvent nécessaire de transformer les données qui seront persistées et celles qui proviennent de la base de données. Par exemple, certains systèmes de persistance patrimoniaux ne permettent pas de conserver des données booléennes. Dans de tels cas, on utilise différents moyens palliatifs. La valeur de vérité est parfois représentée par le caractère « Y » et son inverse par « N ». Une conversion est donc nécessaire.

Pour chaque paramètre en entrée, on doit donc fournir le nom de la variable PCML, sa valeur et, la transformation qui doit y être appliquée. La structure de données détaillant tous les paramètres d'une procédure stockée est représentée à la figure 4.4 en format JSON.

```
class ClickFormBuilder {
  def parent
  def invokeMethod(String meth, Object args) {
    def klass = getClassWithName(meth)
    def child
    if (mapGiven(args)) {
      child = klass.newInstance(args[0])
    } else {
      child = klass.newInstance()
    }
    if (closureGiven(args)) {
      callClosureWithThisParent(child, args[-1])
    }
    if (this.parent) {
      this.parent.add(child)
    } else {
      this.parent = child
    }
    child
  }
  def getClassWithName(name) {
    def pack = "net.sf.click.control"
    Class.forName(${pack}.${capitalize(name)}")
  }
  def mapGiven(args) {
    args.length >= 1 && args[0] instanceof Map
  }
  def closureGiven(args) {
    args.length >= 1 && args[-1] instanceof Closure
  }
  def callClosureWithThisParent(newParent, closure) {
    def oldParent = this.parent
    this.parent = newParent
    instanceEval(closure)
    this.parent = oldParent
  }
  def capitalize(str) { str[0].toUpperCase() + str[1..-1] }
  def instanceEval(closure) {
    closure.delegate = this
    closure()
  }
}
```

Fig. 4.3 Implémentation possible du DSL

```
parametres : {
  variable_1 : {
    "valeur" : valeur_de_la_variable,
    "transformation" : fonction_a_appliquer
  }
  variable_2
    ...
  variable_n
    ...
}
```

Fig. 4.4 Représentation JSON de la structure de données des paramètres

```
retours : {
  alias_1 : {
    "nom_pcml" : nom_de_la_variable_pcml,
    "transformation" : fonction_a_appliquer
  }
  alias_2
    ...
  alias_n
    ...
}
```

Fig. 4.5 Représentation JSON de la structure de données des valeurs de retour

Un certain nombre d'informations est nécessaire pour traiter les valeurs de retour. Évidemment, le nom de la variable PCML est nécessaire. La transformation de la valeur peut être fournie. Le nom des variables PCML étant encombrant, il est intéressant de fournir un nom désiré pour la variable, un alias. La spécification d'un alias permet un appariement plus aisé entre la valeur de retour et l'attribut d'un objet. La structure de données détaillant les valeurs de retour d'une procédure stockée est représentée à la figure 4.5 en format JSON.

En plus du nom du programme PCML à appeler et de l'emplacement du fichier XML, ces deux structures de données peuvent servir d'intrants à l'API initial. Il ne reste donc qu'à définir un nouvel API qui utilise les structures de données. L'utilisation de ce nouvel API est illustrée à la figure 4.6

La méthode statique `RpgCaller.call` retourne un dictionnaire dont les paires « clé-valeur » représentent respectivement les alias associés à leur valeur.

```
def identite = {val -> val}
def params = [
  "PROGRAMME.params-entree.un-float" : [
    "valeur" : 123.12,
    "transformation" : identite
  ]
  "PROGRAMME.params-entree.un-booleen" : [
    "valeur" : true
    "transformation" : {val -> val == true ? "Y" : "N"}
  ]
]
def retours = [
  "unEntier" : [
    "nomPcml" : "PROGRAMME.params-sortie.un-entier",
    "transformation" : identite
  ]
  "uneString" : [
    "nomPcml" : "PROGRAMME.params-sortie.une-string",
    "transformation" : {val -> val.trim()}
  ]
]
RpgCaller.call("chemin/vers/document.pcml",
    "PROGRAMME", params, retours)
```

Fig. 4.6 Utilisation du second API d'appels RPG

Ce second API est, devons-nous le répéter, un DSL. Ce n'est toutefois pas un très bon DSL. Le bruit syntaxique est très présent. On retrouve plusieurs symboles de ponctuation qui sont tout aussi nécessaires qu'ils sont nuisibles : = " : ( ) [ ] ,. L'utilisation de dictionnaires permet une grande flexibilité. L'API peut être augmenté facilement en ajoutant le support pour des clés supplémentaires. Ceci s'est révélé un atout lorsque nous avons dû ajouter le support pour des valeurs de retours « multivaluées ». Par contre, une structure non typée ne permet pas une validation lors de la compilation. La structure des dictionnaires ne peut être vérifiée avant l'exécution du code.

Pour toutes ces raisons, il devient nécessaire de construire un troisième API qui enveloppe le second. Ce nouvel API encapsule la création des dictionnaires de paramètres et de valeurs de retour ainsi que l'appel à l'objet « RpgCaller ». Afin de réaliser ce troisième API nous utiliserons des imbrications de « builders » statiques (voir la section 2.4.5). Voyons d'abord l'utilisation du « builder » qui permettra de construire le dictionnaire des paramètres. La figure 4.7 en illustre l'utilisation.

```
def param = new RpgInputBuilder().build {
  name "PROGRAMME.params-entree.un-float"
  value 123.12
  transformation {val -> val}
}
```

Fig. 4.7 Utilisation du *builder* de paramètres

```
new RpgBuilder().build {
  classPath "chemin/vers/document.pcml"
  program "PROGRAMME"
  inputInBigDecimal {
    name "PROGRAMME.params-entree.un-float"
    value 123.12
  }
  inputInYesNo {
    name "PROGRAMME.params-entree.un-booleen"
    value true
  }
  outputToInteger {
    name "unEntier"
    pcmlName "PROGRAMME.params-sortie.un-entier"
  }
  outputToString {
    name "uneString"
    pcmlName "PROGRAMME.params-sortie.une-string"
  }
}
```

Fig. 4.8 Utilisation du *builder*

Quoique ce *builder* ne permet la construction que d'un seul paramètre, on constate que son utilisation est déjà plus sophistiquée que le dictionnaire brut. La classe `RpgInputBuilder` contient quatre méthodes. `build` reçoit et exécute un bloc de code. À la fin de son exécution, les trois informations nécessaires ont été capturées. La méthode `build` peut alors construire l'entrée du dictionnaire. Les méthodes `name`, `value` et `transformation` ne font que capturer leur paramètre dans une variable d'instance. Le *builder* de valeurs de retour fonctionne exactement de la même façon.

Les *builders* de paramètres et de valeurs de retours ne peuvent construire qu'une seule entrée dans leur dictionnaire respectif. Ces dictionnaires doivent donc se retrouver dans un *super-builder*. Nous nommerons celui-ci `RpgBuilder`. Son utilisation est présentée à la figure 4.8.

```
1  class RpgBuilder {
2    def classPath, programName
3    def inputs = [:], outputs = [:]
4    def identite = { val -> val }

5    def build(closure) {
6      closure.delegate = this
7      closure.call()
8      RpgCaller.call(classPath, programName, inputs, outputs)
9    }
10   def classPath(classPath) {
11     this.classPath = classPath
12   }
13   def program(programName) {
14     this.programName = programName
15   }
16   def input(closure) {
17     new RpgInputBuilder(this, identite).build(closure)
18   }
19   def output(closure) {
20     new RpgOutputBuilder(this, identite).build(closure)
21   }
22 }
```

Fig. 4.9 RpgBuilder

Tout comme les *builders* précédents, `RpgBuilder` possède une méthode `build` qui reçoit et exécute un bloc de code. Les méthodes `classPath` et `program` récupèrent leur paramètre dans des variables d'instance. Les autres méthodes sont plus intéressantes. Les méthodes débutant par `input` instancient un `RpgInputBuilder` et lui passent le bloc de code reçu à la méthode `build`. Les différentes variantes `inputInXyz` et `outputToXyz` initialisent les *builders* internes avec des transformations préenregistrées. Un extrait du code source des trois *builders* est présenté aux figures 4.9, 4.10 et 4.11.

* * *

Dans le cas des deux DSL, nos buts étaient l'amélioration de la lisibilité du code, l'introduction de comportements par défaut et la réduction du temps nécessaire aux efforts de développement subséquents. Le Builder de formulaires permet d'écrire le code source de façon à faire ressortir la structure du formulaire. Au Builder de formulaires présenté précédemment, de nombreux ajouts ont été faits. Le but est toujours d'accomplir la majorité des formulaires avec un minimum d'effort. Pour ce faire, des comportements par

```
1  private class RpgInputBuilder {
2    def parent
3    def name, value, transformation

4    RpgInputBuilder(parent, transformation) {
5      this.parent = parent
6      this.transformation = transformation
7    }
8    def build(closure) {
9      closure.delegate = this
10     closure.call()
11     parent.inputs[name] = [
12       'value': value,
13       'transformation': transformation
14     ]
15   }
16   def name(name) {
17     this.name = name
18   }
19   def value(value) {
20     this.value = value
21   }
22   def transformation(transformation) {
23     this.transformation = transformation
24   }
25 }
```

Fig. 4.10 RpgInputBuilder

```
1  private class RpgOutputBuilder {
2    def parent
3    def name, pcmlName, transformation
4
5    RpgOutputBuilder(parent, transformation) {
6      this.parent = parent
7      this.transformation = transformation
8    }
9    def build(closure) {
10     closure.delegate = this
11     closure.call()
12     parent.outputs[name] = [
13       'pcmlName': pcmlName,
14       'transformation': transformation
15     ]
16   }
17   def name(name) {
18     this.name = name
19   }
20   def pcmlName(pcmlName) {
21     this.pcmlName = pcmlName
22   }
23   def transformation(transformation) {
24     this.transformation = transformation
25   }
}
```

Fig. 4.11 RpgOutputBuilder

défaut y ont été introduits. On a ajouté, entre autres, le remplacement des caractères non-ASCII par des entités HTML dans des libellés, l'initialisation simple des composantes graphiques et l'ajout dynamique d'attributs HTML.

Dans le cas du second DSL, nous cherchions aussi à redonner l'aspect déclaratif initial à la définition de la signature d'un programme RPG/400. L'API existant obscurcissait le côté déclaratif et auto-documenté du fichier PCML au profit d'une interface programmatique. Notre conception du DSL avec un Builder statique est une approche hybride impérative/déclarative à la fois flexible et très lisible.

CONCLUSION

L'utilisation de langages dynamiques et plus spécifiquement de DSL dans le développement d'un logiciel peut avoir plusieurs effets sur le code produit. Certains de ces effets sont néfastes. Nous avons remarqué une certaine tendance vers un style de programmation plus « relâché » où l'écriture de code prend le dessus sur un design méticuleux (McConnell, 1996). Une conséquence indésirable liée à l'utilisation de langages dynamiques est ironiquement liée aux libertés données au programmeur. Groovy est un langage dynamique mais permissif. Il devient incroyablement facile de faire « la mauvaise chose ». Par « mauvaise chose », nous entendons, entre autres, le bris de l'encapsulation, l'utilisation abusive de l'héritage et le recours systématique à des techniques de métaprogrammation. Ce dernier faux-pas a été la source d'un retard considérable lors de l'un des projets. La possibilité de « métaprogrammer » certains comportements de classes a entraîné un développeur dans un dédale informatique duquel il n'a pu se sortir seul. Groovy peut-il être blâmé pour cette expérience désastreuse? Dans ce cas précis, l'inexpérience du développeur est la source véritable du problème. Avant de pouvoir utiliser un langage dynamique, une bonne connaissance des techniques de programmation « objet » de base est nécessaire. Programmer avec un langage dynamique n'est pas une excuse pour écrire du mauvais code. Avant de militer pour l'utilisation de langages statiques, rappelons cette citation merveilleuse de Glenn Vanderburg : « *Bad developers will move heaven and earth to do the wrong thing.* »

Un effet direct remarqué est une réduction dramatique du nombre de lignes de code produites. En comparant des formulaires écrits en Java et les mêmes formulaires écrits avec notre DSL, on remarque que le nombre de lignes de code n'est jamais réduit de moins que 50%. Dans un cas isolé, le nombre de lignes de code est passé de 192 à 28 et ce, sans en sacrifier la lisibilité. Mais la réduction du nombre de lignes de code est-il un objectif

Langage	Niveau	Énoncés par PF
Assembleur	**1**	**320**
C	2,5	128
Lisp	5	64
C++	6	53
Java	6	53
Haskell	8,5	38
Perl	15	21
SmallTalk	15	21
Shell Script Unix, AWK et MAKE	15	21
SQL	25	13

Source : (Jones, 1995)

Tab. 4.1 Niveau approximatif des langages de programmation

valable ? Peut-on justifier l'adoption d'un langage dynamique et/ou la construction de DSL en prétextant que ces techniques réduiront la taille du code produit ?

Plusieurs études tendent à montrer que la productivité d'un développeur ne varie pas en fonction du langage qu'il utilise (Boehm, 1981; Putnam et Myers, 1992; Prechelt, 2000). Ceci signifie que le langage de programmation choisi n'affecte pas le nombre de lignes de code produit pour une période donnée.

Capers Jones a compilé une table comparant le niveau de différents langages de programmation (Jones, 1995). On y indiquent le nombre d'énoncés moyen nécessaires à la réalisation d'un point de fonction. Le tableau 4.1 illustre quelques uns des langages répertoriés. On peut en déduire que, toutes choses étant égales par ailleurs, une fonctionnalité écrite en Perl nécessitera deux fois moins de lignes de code que si elle était écrite en Java ou en C++. Évidemment, la précision absolue de ces chiffres peut être débattue longuement. On ne cherche pas ici à donner des rapports exacts mais simplement un ordre de grandeur. Au moment où ces données ont été obtenues, les langages Groovy et Ruby n'y étaient pas répertoriés. Toutefois, de part nos observations, notre expérience et la nature du langage, ont peut estimer que Groovy se situerait dans le tableau entre Java et Perl. Ruby étant fortement inspiré par Smalltalk et Perl serait aussi positionné

Taille du projet	Densité d'erreur
moins de 2K lignes de code	0-25 bogues par 1000 lignes (KLOC)
2K-16K	0-40 bogues par KLOC
16K-64K	0,5-50 bogues par KLOC
64K-512K	2-70 bogues par KLOC
512K et plus	4-100 bogues par KLOC

Sources : (Jones, 1977; Jones, 1998; McConnell, 1996)

Tab. 4.2 Densité de bogues par KLOC

dans le bas du tableau.

Ainsi, le développement des applications avec notre DSL prend moins de temps tout simplement parce qu'il requiert moins de lignes de code. Des études confirment nos observations. Dans certains cas, il est possible de réduire l'effort de codage de 75% (Jones, 1995). Il est aussi possible de constater que l'effort de design est réduit du même rapport (Klepper et Bock, 1995). Une autre conséquence importante est la réduction de bogues. En effet, des études montrent que la densité de bogues (nombre de bogues par lignes de code) dans un programme n'est pas constante (Jones, 1977; Jones, 1998). Le tableau 4.2 montre que le nombre de bogues par 1000 lignes de code augmente avec la taille d'un projet.

Les DSLs ne sont pas une panacée en matière de développement de logiciels, pas plus que ne l'est la programmation « structurée » ou la programmation orientée-objets. Comme le disait Fred Brooks : « *There is no silver bullet* » (Brooks Jr, 1995). Il n'y aura plus d'avancement majeur qui permettra de réduire la complexité *essentielle* de l'art de la programmation. Depuis déjà quelques années, nous tentons d'en atténuer la complexité *accidentelle* i.e. celle qui n'est pas propre à la programmation en tant que telle.

L'évolution des langages de programmation passe par des hausses incrémentales du niveau d'abstraction. Nous sommes ainsi passés des mnémoniques de l'assembleur à des objets flottant dans un univers immatériel qui communiquent en s'échangeant des messages. Quoiqu'il est toujours important de savoir ce que le code source fait, le programmeur n'a plus besoin de s'en préoccuper autant. Les langages de programmation utilisés

en entreprises aujourd'hui, comme Java et C#, ont été conçus dans une optique de simplicité. Des choix de design ont été pris en faveur de la réduction de la complexité et de l'augmentation de la « sécurité ». Pourquoi se soucier de la déréférenciation de pointeurs ou de gestion de l'allocation de mémoire lorsque le langage et la machine virtuelle s'occupent de tout ? Ceci ne veut pas dire pour autant que la programmation est devenue plus aisée. Cette réduction de la complexité accidentelle ne rend pas les programmes plus faciles à écrire. Bien au contraire, elle permet de relever de nouveaux défis qui eux peuvent s'avérer être de véritables nids de vipères.

En même temps, les clients sont, à juste titre, de plus en plus exigeants. Un site web qui arbore un *look* « pré-1999 » n'aura tout simplement pas la cote. Les programmeurs web doivent donc rivaliser d'ingéniosité et utiliser de nouvelles techniques comme Ajax, Comet et autres pour rester compétitifs. Évidemment, dans la foulée du « Web 2.0 » (O'Reilly, 2005), le temps à la mise en marché est d'une importance cruciale. On y privilégie donc l'utilisation d'outils programmatiques simples ainsi qu'un style de livraison qualifié de « beta perpétuel » où le logiciel est modifié continuellement.

Nous avons vu comment les langages de programmation se sont adapté à ce nouvel écosystème économique. Une première adaptation est faite au niveau du système de types. Le typage statique et fort a tendance à être remplacé par un typage fort mais dynamique : le *Duck Typing*. Ce style de typage apporte une flexibilité importante aux méthodes et aux classes utilisées. Des concepts difficiles à implémenter avec un typage statique deviennent subitement beaucoup plus simples et ce, sans perdre l'avantage d'un typage fort. Une seconde adaptation se trouve au niveau d'une utilisation de plus en plus importante de la métaprogrammation. Des choses impossibles à réaliser avec les langages établis le deviennent. Par exemple, il est tout simplement impossible d'intercepter des appels vers des méthodes ou des attributs inexistants en Java. Ainsi, l'évolution des langages tend à donner une plus grande importance à l'exécution d'un programme plutôt qu'à sa compilation.

Nous avons ensuite montré comment ces nouveaux langages de programmation per-

mettent de déverrouiller la porte qui mène vers un « nouveau » niveau d'abstraction : les langages spécifiques à un domaine d'affaire. Loin d'être un nouveau concept, les DSLs sont remis au goût du jour par les membres des communautés Ruby, Python et Groovy. Un DSL est un langage qui s'adresse aux experts d'un domaine en particulier. Les domaines sont très variés : constructions de logiciels, design d'interfaces graphiques, mise-en-page HTML, élaborations de règles d'affaires, etc. Nous développons déjà avec plus d'un langage de programmation à la fois et plus d'un DSL dans un même projet. La plupart des *frameworks* utilisent un DSL pour exprimer leur configuration. Pensons simplement à Ant, Maven, Hibernate et Spring qui nécessitent tous des fichiers de configuration XML. Certains de ceux-ci utilisent aussi des DSLs internes basés sur les annotations Java pour arriver aux mêmes fins.

L'objectif principal de ce travail était de démontrer comment il est simple et avantageux de construire et d'utiliser des DSLs *ad hoc* pour des projets de développement. Nous croyons fermement que chaque partie d'un programme doit communiquer l'*intention* du programmeur. Ce constat provient directement du fait que le code source est beaucoup plus souvent lu qu'il n'est écrit. Ce sont les mêmes raisons qui nous poussent à construire des interfaces programmatiques simples et intuitives à nos programmes et qui nous amènent à construire des DSLs.

Le cas de la construction de structures hiérarchiques met en lumière l'importance de l'intention du programmeur. Une structure hiérarchique comme un document XML ou une interface utilisateur graphique devrait être *définie* hiérarchiquement. Pourtant, le paradigme « objets/méthodes » ne permet pas d'exprimer cette intention clairement. En utilisant le patron *Builder* pour construire les formulaires web, il nous a été très simple de (1) construire les formulaires et (2) expliquer aux autres membres de l'équipe la mise en page de ceux-ci. L'imbrication des différentes composantes ainsi que la disposition verticale des appels de méthodes convoie un message très précis au lecteur. D'un seul coup d'œil, il est possible de reconnaître la forme du formulaire et, dans le cas échéant, l'emplacement d'un bogue éventuel. L'utilisation du *Builder* augmente aussi la cohésion du code source produit. En effet, à la pratique, on remarque que le code source servant

à générer les formulaires se « purifie », se spécialise. L'inclusion de code ne servant pas directement à bâtir l'interface graphique est effectivement beaucoup moins invitante. Aussi, la refactorisation d'un formulaire devient plus intuitive et a moins tendance à causer des erreurs de mise en page. Autre avantage intéressant, nous avons intégré dans le *Builder* des raccourcis pour les actions les plus souvent effectuées. Il nous est donc possible de définir certaines composantes graphiques non-élémentaires à partir de commandes spéciales. Ainsi, sans trop complexifier la grammaire du DSL, nous sauvons un temps considérable sur la construction des composantes répétitives.

Si la vie semble plus belle après l'ajout de DSL dans nos projets, tous n'est pas rose pour autant. Une première critique provient de développeurs apeurés par le côté « langage » d'un DSL. Ne risque-t-on pas de sombrer dans un déluge de mini-langages qui devront *tous* être maîtrisés avant d'être utilisés ? C'est effectivement le cas. Tous les DSL d'un projet doivent être compris avant d'être utilisés. Il en va de même pour les librairies externes, les fichiers de configuration XML et les *frameworks* utilisés lors d'un projet. Le temps nécessaire à l'apprentissage d'un nouveau DSL n'est pas supérieur à celui nécessaire à l'apprentissage d'un nouvel API. Seule la forme de l'interface est modifiée.

Nous avons aussi pu constater qu'un DSL n'obtient pas nécessairement sa forme finale dès le premier jet. En effet, le DSL servant à effectuer des appels RPC est le résultat d'une lente évolution. Plusieurs semaines se sont écoulées entre l'utilisation de l'API de base et celle du DSL. Toutefois, puisque le DSL est bâti *par-dessus* l'API existant, la rétro-compatibilité est assurée. Ceci permet la coexistence de code patrimonial pré-DSL avec le nouveau code source.

Grâce aux succès encourus par l'implémentation des DSLs, plusieurs pistes d'amélioration et d'approfondissement s'ouvrent à nous. Les applications web construites dans le présent cadre font toutes partie d'une même famille. Elles sont basées sur le principe d'une machine à états où chacun de ceux-ci sont caractérisés par une page et un formulaire HTML. Les transitions se font via la soumission de formulaires. Les données entrées par l'utilisateur le dirigent à travers le graphe de pages. La forme de ces applica-

tions étant structurante et implémentée de façon relativement uniforme, il est possible de développer un DSL qui décrit le graphe de pages ainsi que les transitions. Ce DSL permettrait de communiquer plus efficacement avec les clients-utilisateurs principaux du système. Nous ne pensons toutefois pas que ce DSL serait en mesure de définir l'application en entier. Il servirait principalement à définir la *coquille* de l'application web. Cette coquille serait alors transformée en application web Click/Groovy grâce à un compilateur, un traducteur, sur-mesure. Il s'agirait donc d'un DSL externe dont le code source servirait d'intrant à un générateur de code.

Le *Builder* de formulaires HTML peut aussi être amélioré de diverses façons. Il serait premièrement intéressant de pouvoir insérer du code source Javascript à l'intérieur même de la définition des contrôles HTML. Ce code serait exécuté dans le fureteur lors du déclenchement d'un événement HTML. Ceci permettrait d'inclure des comportements dynamiques et asynchrones plus naturellement. Présentement, le seul événement pouvant être capturé est « `onclick` ». Afin d'inclure tous les autres événements HTML, une réimplémentation de l'API de contrôles Click devrait être faite. Une autre piste d'amélioration serait de permettre la spécialisation d'un *Builder* soit par l'héritage ou le patron « Stratégie ». Il est présentement possible de modifier la présentation d'un formulaire par les paramètres passés lors de la construction. Tous les formulaires d'une même application sont généralement semblables. Le code de personnalisation de la présentation est donc souvent obligatoirement recopié d'une classe à une autre. Il serait intéressant de pouvoir encapsuler les paramètres de personnalisation dans une entité logicielle séparée et de l'utiliser pour l'application en entier. Cette amélioration permettrait d'alléger grandement la définition des formulaires et éliminerait beaucoup de répétitions.

Bien que ce ne soit pas une technique nouvelle, les nouveaux langages de programmation favorisent l'utilisation de DSL afin d'améliorer la communication de l'intention du programmeur. Comme tout outil puissant, la définition et le maniement de langages spécialisés à un domaine d'affaire requiert professionnalisme, sérieux, adresse et jugement. Une utilisation judicieuse de DSL permet d'améliorer la communication entre les développeurs et les clients, de simplifier la rédaction et la compréhension du code source

et ainsi d'en faciliter la maintenance, le « refactoring » et le débogage.

RÉFÉRENCES

Aahz. 2003. Typing : Strong vs. Weak, Static vs. Dynamic. http ://www.artima.com/weblogs/viewpost.jsp ? thread=7590 Consultée le 2008-09-12.

Appel, A. et J. Palsberg. 2002. *Modern Compiler Implementation in Java*. Cambridge University Press, second édition.

Beck, K. 2008. *Implementation Patterns*. Addison-Wesley Professional.

Bini, O. 2008. Don't overuse instance_eval and instance_exec. http ://ola-bini.com/blog/ 2008/09/ dont-overuse-instance_eval-and-instance_exec/ Consultée le 2008-10-27.

Boehm, B. 1981. *Software engineering economics*. Prentice-Hall Englewood Cliffs, NJ.

Brooks Jr, F. 1995. *The mythical man-month (anniversary ed.)*. Addison-Wesley.

Davidson, J. 2004. Ant and XML Build Files. http ://web.archive.org/web/20040602210721/ x180.net/Articles/ Java/AntAndXML.html Consultée le 2008-09-12.

Eckel, B. 2008. Java : Evolutionary Dead End. http ://www.artima.com/ weblogs/viewpost.jsp ? thread=221903 Consultée le 2008-09-12.

Flanagan, D. 2005. *Java in a Nutshell*. O'Reilly Media, Inc., fifth édition.

Ford, N. 2006. Polyglot Programming. http ://memeagora.blogspot.com/2006/12/polyglot-programming.html Consultée le 2008-09-12.

———. 2007. Developer Productivity Mean vs. Median. http ://memeagora.blogspot.com/2007/10/developer-productivity-mean-vs-median.html Consultée le 2008-09-12.

Fowler, M. 2003. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional.

———. 2004. SoftwareDevelopmentAttitude. http ://martinfowler.com/bliki/SoftwareDevelopmentAttitude.html Consultée le 2008-09-12.

———. 2008. Domain-Specific Languages. WIP. http ://martinfowler.com/dslwip/ Consultée le 2008-09-12.

Gagnon, E. et L. Hendren. 1998. « SableCC, an object-oriented compiler framework », *In Proceedings of TOOLS 1998-26th International Conference and Exhibition*, p. 140–154.

Gamma, E., R. Helm, R. Johnson, et J. Vlissides. 1995. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley.

Jones, C. 1977. *Program Quality and Programmer Productivity*. IBM General Products Division, Santa Teresa Laboratory.

———. 1995. « Programming Languages Table », *Burlington, MA : Software Productivity Research, March*. www.mindspring.com/ dway/smalltalk/docs/0langtbl.pdf Consultée le 2008-10-19.

Jones, T. 1998. *Estimating software costs*. McGraw-Hill, Inc. Hightstown, NJ, USA.

Kelly, S. 2006. Recovery from Addiction. http ://www.archive.org/details/SeanKelly RecoveryfromAddiction Consultée le 2008-09-12.

Khun, T. 1983. *La structure des révolutions scientifiques*. Paris, Flammarion.

Klepper, R. et D. Bock. 1995. « Third and fourth generation language productivity differences », *Communications of the ACM*, vol. 38, no. 9, p. 69–79.

Laddad, R. 2003. *AspectJ in Action : Practical Aspect-Oriented Programming*. Manning.

Lilly, J. 1972. *Programming and Metaprogramming in the Human Biocomputer : Theory and Experiments*. Julian Press.

McConnell, S. 1996. *Rapid Development : Taming Wild Software Schedules*. Microsoft Press.

———. 2004. *Code complete*. Microsoft Press, second édition.

Mernik, M. et A. Sloane. 2005. « When and how to develop domain-specific languages », *ACM Computing Surveys (CSUR)*, vol. 37, no. 4, p. 316–344.

O'Reilly, T. 2005. « What is Web 2.0 : Design Patterns and Business Models for the Next Generation of Software ».

Parr, T. 2007. *The Complete Antlr Reference Guide*. Pragmatic Bookshelf.

Pillai, P. 2004. Introduction to Static and Dynamic Typing. http ://www.sitepoint.com/article/typing-versus-dynamic-typing Consultée le 2008-09-12.

Prechelt, L. 2000. « An empirical comparison of C, C++, Java, Perl, Python, Rexx and Tcl », *IEEE Computer*, vol. 33, no. 10, p. 23–29.

Putnam, L. et W. Myers. 1992. « Measures for Excellence : Reliable Software on Time, on Budget », *Yourdon, Englewood Cliffs, NJ*.

Sammet, J. E. 1981. « The early history of cobol », p. 199–243.

the Lucky Stiff, W. 2008. Mixing Our Way Out Of Instance Eval? http ://hackety.org/2008/10/06/ mixingOurWayOutOfInstanceEval.html Consultée le 2008-10-26.

Thomas, D. et A. Hunt. 2001. *Programming Ruby*. Addison-Wesley.

Tolksdorf, R. 2008. Programming Languages for the Java Virtual Machine. http ://www.is-research.de/info/vmlanguages/ Consultée le 2008-09-12.

Wall, L., T. Christiansen, et J. Orwant. 2000. *Programming Perl*. O'Reilly.

Ward, M. 1994. « Language-Oriented Programming », *Software - Concepts and Tools*, vol. 15, no. 4, p. 147–161.

Whorf, B., J. Carroll, et S. Chase. 1956. *Language, Thought, and Reality : Selected Writings of Benjamin Lee Whorf*. Technology Press of Massachusetts Institute of Technology.

Wittgenstein, L. 1981. *Tractatus Logico-Philosophicus*. Routledge.