UNIVERSITÉ DU QUÉBEC À MONTRÉAL

MIGRATION D'UN SYSTÈME ORIENTÉ OBJETS VERS UN SYSTÈME ORIENTÉ SERVICES WEB DE TYPE REST

RAPPORT DE PROJET

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN GÉNIE LOGICIEL

PAR

MOUSTAPHA BOULGOUDAN

SEPTEMBRE 2017

UNIVERSITÉ DU QUÉBEC À MONTRÉAL
Service des bibliothèques

Avertissement

La diffusion de ce document diplômant se fait dans le respect des droits de son auteur, qui a signé le formulaire *Autorisation de reproduire et de diffuser un travail de recherche de cycles supérieurs* (SDU-522 – Rév.10-2015). Cette autorisation stipule que «conformément à l'article 11 du Règlement no 8 des études de cycles supérieurs, [l'auteur] concède à l'Université du Québec à Montréal une licence non exclusive d'utilisation et de publication de la totalité ou d'une partie importante de [son] travail de recherche pour des fins pédagogiques et non commerciales. Plus précisément, [l'auteur] autorise l'Université du Québec à Montréal à reproduire, diffuser, prêter, distribuer ou vendre des copies de [son] travail de recherche à des fins non commerciales sur quelque support que ce soit, y compris l'Internet. Cette licence et cette autorisation n'entraînent pas une renonciation de [la] part [de l'auteur] à [ses] droits moraux ni à [ses] droits de propriété intellectuelle. Sauf entente contraire, [l'auteur] conserve la liberté de diffuser et de commercialiser ou non ce travail dont [il] possède un exemplaire.»

REMERCIEMENTS

Je tiens tout d'abord à remercier mes directrices de projet, Prof. Naouel Moha et Prof. Ghizlane El Boussaidi pour leur soutien et leurs critiques constructives tout au long de ce projet.

Je remercie aussi Francis Palma pour son aide précieuse à la compréhension du système existant et pour sa disponibilité à répondre à mes questions.

Enfin, je remercie ma femme Chihaz qui m'a toujours soutenu malgré mes absences prolongées et mes enfants Ayman, Aya et Basma pour leur compréhension et leur patience. Je leur promets de rattraper le temps perdu et de passer beaucoup plus de moments agréables avec eux.

DÉDICACES

À

Mon père,

qui m'a tout donné et m'a toujours encouragé à aller de l'avant et réaliser mes rêves

Ma mère,

qui a été toujours là pour moi avec son amour indéfectible et inconditionnel.

TABLE DES MATIÈRES

LISTE DES TABLEAUX 7

LISTE DES FIGURES 8

RÉSUMÉ 10

CHAPITRE I
INTRODUCTION 11

1.1 Problématique 11

1.2 Objectifs 13

1.3 Méthodologie 14

1.3.1 Analyse du système patrimonial orienté objets 14

1.3.2 Migration du système patrimonial vers le système cible 15

1.4 Plan du document 18

CHAPITRE II
CONCEPTS/ÉTAT DE L'ART 20

2.1 Les approches d'évolution des systèmes patrimoniaux 20

2.1.1 La maintenance 21

2.1.2 La modernisation 23

2.1.3 Le remplacement 23

2.2 Les techniques de modernisation 24

2.2.1 Le rhabillage 24

2.2.2 La réingénierie 25

2.2.3 La migration 25

2.3 L'architecture logique en couches 26

2.3.1 Définition 26

2.3.2 Le niveau de décomposition d'un système 27

2.4 Le style architectural REST 28

2.4.1 Définition 28

2.4.2 L'interface uniforme 30

2.4.3 Les patrons et anti-patrons REST 34

2.5 L'architecture SCA 39

2.5.1 Définition 39

2.5.2 Le composant 39

CHAPITRE III
COMPRÉHENSION DU SYSTÈME PATRIMONIAL 41

3.1 Introduction 41

3.2 Analyse du système patrimonial 42

3.2.1 Le modèle des cas d'utilisation 42

3.2.2 Les concepts du domaine 42

3.2.3 Le diagramme des classes 44

3.3 Ajout manuel d'une nouvelle API au système 46

3.3.1 S'enregistrer auprès du fournisseur de l'API 48

3.3.2 Sélectionner un échantillon de ressources de l'API 49

3.3.3 Créer le composant SCA 50

CHAPITRE IV
MIGRATION DU SYSTÈME PATRIMONIAL VERS LE SYSTÈME CIBLE 54

4.1 L'architecture cible 54

4.1.1 La vue physique 54

4.1.2 Les vues logiques 56

4.1.3 Structure de nommage des packages 60

4.1.4 Choix techniques 62

4.2 La stratégie de migration 63

4.3 Implémentation en couches 64

4.4 Migration vers le système cible 66

4.5 Migration de la couche d'accès aux données 66

4.6 Migration de la couche logique métier 69

4.7 Migration de la couche présentation : l'API REST 71

4.7.1 Modélisation des ressources 72

4.7.2 Implémentation des services Web de l'API REST 74

4.8 Migration de la couche présentation : l'application Web 75

4.9 Ajouter une nouvelle API au système 75

4.9.1 Mustache 75

4.9.2 Les gabarits et le hash 77

CONCLUSION 78

ANNEXE A
EXTRAITS DE CODE 80

BIBLIOGRAPHIE 96

LISTE DES TABLEAUX

Tableau Page

3.1 Description du cas d'utilisation CU1. 43

4.1 Nomenclature de l'API REST. 73

LISTE DES FIGURES

Figure Page

1.1 L'approche API en premier. 17

2.1 Cycle de vie d'un système d'information. Extrait de (Seacord *et al.*, 2003). 22

2.2 Architecture logique de l'application. 27

2.3 Niveaux de décomposition d'un système. 28

2.4 Exemple de composant SCA. 40

3.1 Diagramme des cas d'utilisation. 43

3.2 Diagramme des classes qui participent à la détection des (anti)patrons. 45

3.3 Diagramme des classes qui participent à l'affichage des résultats. . 46

3.4 Le fichier composite FraSCAti pour l'API opendata. 51

4.1 Architecture physique de l'application. 55

4.2 Utilisation de l'API par un client REST. 56

4.3 Utilisation de l'API par l'application Web. 57

4.4 Architecture multi-niveau de l'application. 59

4.5 Architecture en couche et patron architectural MVC. 61

4.6 Structure de nommage des packages. 62

4.7 Modernisation d'une couche d'abstraction. 63

4.8 Stratégie de migration en couches. 64

4.9 Implémentation en couches. 65

4.10 Le modèle Entité-Relation (ER). 67

4.11 Ajout d'une nouvelle API (opendata) au système. 76

A.1 L'interface définissant les méthodes d'utilisation de l'API. 81

A.2 La classe client qui va accéder aux services Web REST. 82

A.3 Implémentation de l'objet du domaine `ApiEntity`. 83

A.4 `ApiDao.java` - Interface de l'objet d'accès aux données. 84

A.5 `ApiDaoJpaImpl.java` - Implémentation JPA de l'interface ApiDao. 85

A.6 `ApiDaoJdbcImpl.java` - Implémentation JDBC de l'interface Apidao. 86

A.7 `ApiBlo.java` - Interface de la couche logique métier. 87

A.8 `ApiBloImpl.java` - Implémentation de la couche logique métier. . 88

A.9 Représentation d'une ressource de base : l'API `alchemy`. 89

A.10 Représentation d'une collection de ressources : liste des APIs. . . 89

A.11 Représentation d'un processus : détection des patrons de l'API `bitly`. 90

A.12 Code pour récupérer la ressource `/apis` : Liste des APIs. 91

A.13 Le gabarit Mustache du fichier composite FraSCAti. 92

A.14 Le gabarit Mustache pour la création de l'interface associée à une API. 93

A.15 Le gabarit Mustache pour la création du client qui implémente l'interface associée à l'API. 94

A.16 Le Hash qui spécifie l'échantillon de ressources pour l'ensemble de ressources `energy-usage-2010`. 95

RÉSUMÉ

Le but de ce projet est de migrer un système orienté objet s'exécutant sur une machine locale vers un système orienté services Web de type REST (*REpresentational State Transfer*) accessible à n'importe quel utilisateur connecté à Internet. Nous proposons une stratégie de migration en deux étapes. La première étape consiste à comprendre le système patrimonial en étudiant les scénarios possibles d'exécution des cas d'utilisation et ensuite en analysant le code qui implémente ces cas d'utilisation. La deuxième étape consiste à définir l'architecture cible, à décrire la migration des trois couches d'abstraction (présentation, logique métier, accès aux données) et enfin à automatiser l'ajout d'une nouvelle API (*Application Programming Interface*) au système en utilisant un moteur de gabarit qui générera automatiquement le code lié à cette API.

CHAPITRE I

INTRODUCTION

Ce rapport est le résultat de mon projet de maîtrise en génie logiciel qui a été effectué au sein du laboratoire de recherche LATECE à l'UQAM.

1.1 Problématique

Le style architectural REST (*REpresentational State Transfer*) (Fielding, 2000) est largement utilisé pour implémenter des services Web en utilisant uniquement les mécanismes inhérents du protocole de communication HTTP (*HyperText Transfer Protocol*) qui a fait le succès de l'Internet. Les services Web qui sont créés conformément au style architectural REST sont communément appelés API REST (API : *Application Programming Interface*).

La création d'un système orienté services Web de type REST passe par les phases suivantes (Pautasso, 2009) :

1. Identifier les ressources qui vont être exposées comme services Web; une ressource est une information nommée qui s'identifie de façon unique via une URL;
2. Modéliser les relations entre les ressources avec des liens qui peuvent être suivis pour obtenir plus d'information;

3. Définir les URLs qui identifient les ressources ;
4. Comprendre l'association des méthodes HTTP (GET, PUT, POST, DELETE) avec chaque ressource ;
5. Documenter et concevoir les représentations de chaque ressource ;
6. Implémenter les services Web ;
7. Déployer sur le serveur Web ;
8. Tester avec un client REST.

Le but de ce projet est de mettre en œuvre une stratégie de migration d'un système orienté objets vers un système orienté services Web de type REST afin de réaliser les étapes précédentes. Cette migration permettra la réutilisation des fonctionnalités clés par les développeurs et les partenaires d'affaires.

Pour mener à bien cette migration, on a besoin de comprendre le fonctionnement interne du système patrimonial afin d'extraire les parties du code qui vont être incorporées dans les services Web de la nouvelle API. Cette migration va nécessiter une refactorisation des parties du code afin de régler le problème de dépendances des nouveaux modules.

Le système patrimonial orienté objet est une application Java de bureau qui permet la détection des patrons et anti-patrons REST des APIs HTTP publiques. Ce système utilise, entre autres, la technologie SCA (*Service Component Architecture*) pour communiquer avec les systèmes distants hébérgeant les APIs.

Le système patrimonial souffre des limitations suivantes :

1. L'utilisation de l'application est limitée à l'utilisateur qui l'installe localement sur sa machine et n'est pas accessible au plus grand nombre via le Web.

2. La création du composant SCA se fait en ajoutant manuellement du code dans l'application, ce qui est sujet aux erreurs et demande un effort non négligeable si l'échantillon de ressources sélectionnées est grand.
3. Les traces d'exécution qui affichent les messages HTTP des URLs, objet de détection :
   - sont stockées dans des fichiers CSV et ne sont pas accessibles directement dans l'application ;
   - ne caractérisent pas de façon précise les patrons et anti-patrons associées à ces URLs.
4. La détection se fait en bloc pour tous les patrons et anti-patrons et ne permet pas de choisir seulement les patrons ou les anti-patrons ou bien un sous-ensemble des patrons et des anti-patrons.
5. Les rapports de synthèse pour l'aggrégation et la comparaison des données se font manuellement dans un chiffrier.

1.2 Objectifs

L'objectif de ce projet consiste à pallier aux limitations du système patrimonial en mettant en œuvre une stratégie de migration d'un système orienté objets vers un système orienté services Web de type REST. Le nouveau système sera composé d'une application Web et d'un ensemble de services Web de type REST accessibles via une API Web publique. Cette API va être exploitée principalement par des développeurs qui vont pouvoir l'intégrer dans leurs applications. L'application Web va être exploitée par des utilisateurs finaux qui vont pouvoir tester la qualité des APIs publiques qu'ils utilisent, mais aussi par les fournisseurs d'API pour soumettre leurs APIs pour qu'elles soient intégrées au système.

Pour atteindre l'objectif de migration, il faudra comprendre le système patrimonial

et notamment l'utilisation de la technologie SCA dans le processus de détection des patrons et anti-patrons des APIs.

1.3 Méthodologie

La méthodologie adoptée se base sur deux étapes successives à accomplir :

— Analyse du système patrimonial orienté objets ;
— Migration du système patrimonial orienté objets vers le système cible orienté services.

Les sections suivantes apportent des précisions quant à l'analyse et la migration du système patrimonial.

1.3.1 Analyse du système patrimonial orienté objets

L'analyse du système patrimonial permet de comprendre le fonctionnement interne du système patrimonial pour pouvoir l'améliorer. Cette compréhension doit se faire d'abord d'un point de vue externe ou utilisateur en considérant le système comme une boîte noire. Ensuite, une analyse du fonctionnement interne du système s'impose pour déterminer quelles sont les composantes du système qui participent aux cas d'utilisation.

L'analyse du système va se faire en trois étapes :

1. Utiliser le système en mode opératoire, en exécutant les différents scénarios possibles, pour en déduire le modèle des cas d'utilisation ;
2. Identifier les concepts du domaine et leurs relations à partir du modèle des cas d'utilisation ;
3. Localiser les composants du système qui implémentent les cas d'utilisation.

Cette analyse est faite au niveau de l'interface utilisateur, au niveau applicatif et au niveau des données. Au niveau de l'interface utilisateur, elle consiste à reproduire les cas d'utilisation du système. Au niveau applicatif, elle consiste entre autres à extraire ses composants (classes) réutilisables et ses interfaces. Le comportement de l'application en termes d'interaction entre les objets doit aussi être extrait en partie pour pouvoir le reproduire sur le système cible. Enfin, au niveau des données, elle consiste à reconstruire la structure des données.

1.3.2 Migration du système patrimonial vers le système cible

La migration consiste à proposer une architecture cible et à l'implémenter. Nous allons opérer cette migration en quatre étapes :

1. Définition de l'architecture cible ;
2. Choix de la stratégie de migration ;
3. Implémentation du système cible ;
4. Déploiement sur la nouvelle plateforme.

Définition de l'architecture cible

Durant cette étape, nous allons concevoir l'architecture du système cible afin qu'elle soit évolutive et maintenable. La définition de cette architecture cible consiste à présenter :

1. une vue physique du système montrant les machines qui supportent l'application ;
2. plusieurs vues logiques du système pour mieux cerner l'interaction entre les différents composants logiciels du système ;
3. une structure de nommage des packages ;

4. les choix techniques pour implémenter la persistance des données, les services Web de type REST ainsi que l'application Web.

Choix de la stratégie de migration

La stratégie de migration consiste à choisir quelle approche de modernisation est plus appropriée au niveau de chacune des trois couches d'abstraction logicielle (présentation, métier, données) du système patrimonial. Le choix de ces approches de modernisation se fera après l'analyse du système patrimonial.

Implémentation du système cible

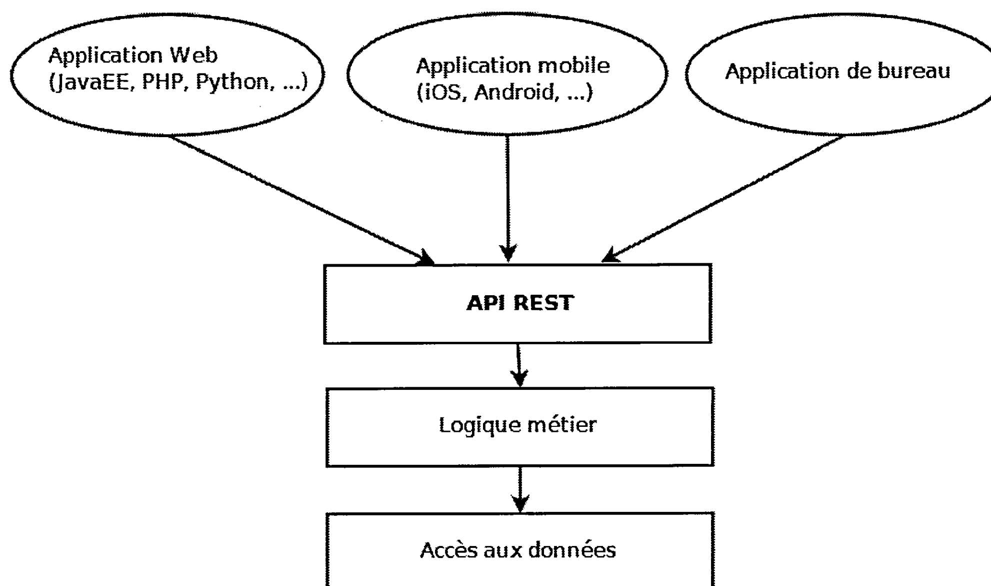
La solution proposée doit s'intégrer avec l'existant. C'est pourquoi nous allons utiliser pour les services Web REST le cadre de référence d'implémentation de la spécification JAX-RS, Jersey et pour l'application Web les technologies Java EE dédiées aux environnements Web.

La transformation du système patrimonial vers le système cible consiste à migrer l'application sur la nouvelle plateforme, en l'occurence de l'environnement de bureau Java vers l'environnement Web Java EE. Quoique les deux environnements sont basés sur le même langage de programmation Java, les technologies utilisées ne sont pas les mêmes. En environnement Java, le programme est un processus interactif en mode console qui intéragit avec les commandes de l'utilisateur localement. En environnement Java EE, l'application tourne sur un serveur d'application (Tomcat) qui incorpore un serveur Web et qui s'attend donc à recevoir des requêtes HTTP d'un client distant (navigateur Web).

Comme l'objectif du projet est de créer à la fois une API avec des services Web RESTful et une application Web, nous avons opté pour une architecture multiniveaux qui supporte les deux technologies. L'architecture est multi-niveaux dans le sens où les trois couches d'abstraction logicielle peuvent résider chacune sur un

serveur différent. Nous allons, donc, développer l'API en premier et réutiliser ses services Web dans l'application Web. Quand on exécute une fonctionnalité dans l'application Web, celle ci fait appel à l'API qui va invoquer la logique métier et lui retourner le résultat (Figure 1.1).

Figure 1.1 L'approche API en premier.



Cette approche de développement, dite *API-first*, qui consiste à bâtir l'API (ou les services Web de type REST) en premier et ensuite développer les applications (Web, mobiles ou de bureau) autour de cette API a plusieurs avantages, parmi lesquels :

1. Utilisation de l'API par des clients de différentes plateformes : Une même API peut être utilisée pour développer des applications à travers plusieurs plateformes en l'occurence Java EE, .Net, PHP, Pyhton, Ruby, Go, ... La documentation de l'API est suffisante pour l'utiliser.
2. La logique métier, qui peut être complexe, est appelée uniquement via l'API : Les développeurs des applications Web et mobiles n'ont pas à com-

prendre cette complexité. Cette approche évite de dupliquer le code métier au niveau de l'API et de l'application Web quoique techniquement l'application Web a aussi accès à la logique métier.

3. Les clients REST et les navigateurs Web auront accès aux mêmes fonctionnalités du nouveau système.

Cette approche exige, néanmoins, de la rigueur au niveau de la gestion de l'API. Par exemple, une amélioration d'une fonctionnalité ne devrait pas casser le code des applications Web et mobiles, mais plutôt garantir la rétro-compatibilité avec les applications existantes.

Déploiement sur la nouvelle plateforme

Le déploiement consiste à installer l'API et l'application Web sur la nouvelle plateforme. Ceci va se faire en deux étapes :

1. Installation et configuration du serveur d'application Tomcat ;
2. Installation et configuration du serveur de bases de données MySQL.

1.4 Plan du document

La suite de ce rapport est organisé comme suivant :

Dans le chapitre II, nous allons décrire l'état de l'art et les concepts indispensables pour la réalisation du projet.

Dans le chapitre III, nous allons analyser le système patrimonial existant pour déterminer la nature des changements à apporter avant de procéder à la migration vers le système cible.

Dans le chapitre IV, nous allons implémenter la stratégie de migration en propo-

sant une architecture pour le nouveau système cible et en développant les composants logiciels qui composent cette architecture.

Enfin, nous allons terminer avec une conclusion.

CHAPITRE II

CONCEPTS/ÉTAT DE L'ART

Dans ce chapitre, nous allons présenter les concepts qui sont nécessaires pour la compréhension des chapitres suivants, mais également l'état de l'art sur les approches d'évolution des systèmes patrimoniaux, les techniques de modernisation, l'architecture logique en couches, le style architectural REST et l'architecture SCA (*Service Component Architecture*).

2.1 Les approches d'évolution des systèmes patrimoniaux

Un système patrimonial peut être défini comme étant "n'importe quel système d'information qui résiste aux modifications et à l'évolution" (Brodie et Stonebraker, 1995).

Un système peut être considéré comme patrimonial s'il est soumis à une ou plusieurs des contraintes suivantes :

- il ne peut plus évoluer pour répondre aux nouveaux besoins ;
- l'ajout de nouvelles fonctionnalités requiert sa ré-architecture ou sa migration ;
- sa webification requiert sa migration vers une plateforme Web et le redéveloppement, entre autres, de la couche présentation.

Un système patrimonial nécessite une approche d'évolution qui permet de s'af-

franchir des contraintes énumérées précédemment et de réutiliser les connaissances acquises au fil du temps.

Les activités d'évolution d'un système patrimonial peuvent être divisées en trois catégories : la maintenance, la modernisation et le remplacement (Seacord *et al.*, 2003). Comme le montre la figure 2.1, ces activités participent au cycle de vie d'un système d'information. Les besoins d'affaires (lignes pointillées) évoluent constamment et determinent quelle activité d'évolution est la plus appropriée pour un système d'information. L'activité de maintenance commence une fois que le système (*System 1*) a été construit. Dès que les besoins d'affaires surpassent les tâches de maintenance, on passe à l'activité de modernisation qui apportera un second souffle au système d'information. Suivra alors une seconde période de maintenance. Enfin, quand la modernisation n'arrive plus à répondre aux besoins d'affaires, on passe à l'activité de remplacement du système pour avoir un nouveau système (*System 2*). Un nouveau cycle commencera alors pour le nouveau système avec une activité de maintenance.

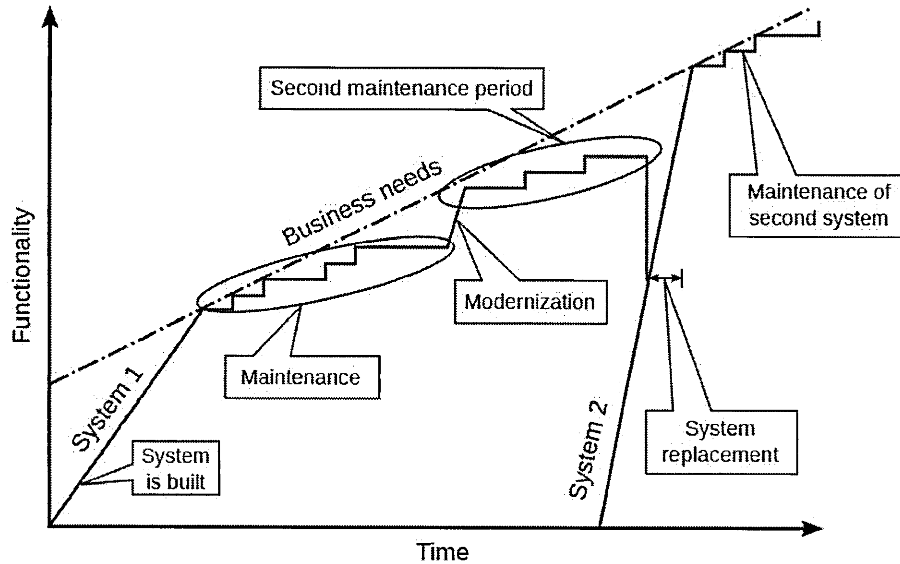
Dans ce qui suit, nous allons décrire ces trois approches ou activités d'évolution d'un système patrimonial.

2.1.1 La maintenance

La maintenance est un processus incrémental et itératif dans lequel de petits changements sont effectués dans le système. Ces changements concernent principalement la correction des défauts et des améliorations mineures et ne requièrent jamais des changements structurels majeurs (Seacord *et al.*, 2003).

Pour définir la nature et l'étendu des changements apportés à un système, l'IEEE (IEEE Std 1219-1998) définit quatre types de maintenance :

Figure 2.1 Cycle de vie d'un système d'information. Extrait de (Seacord *et al.*, 2003).



— *la maintenance corrective* : on effectue des changements pour corriger des défauts dans le système ;

— *la maintenance perfective* : on effectue des changements pour améliorer le système en ajoutant ou modifiant des fonctionnalités ;

— *la maintenance adaptative* : on effectue des changements pour s'adapter à un nouvel environnement (nouvelle plateforme matérielle ou logicielle, changement des règles d'affaires, ...) ;

— *la maintenance préventive* : on effectue des changements pour améliorer la maintenabilité future du système afin de prévenir sa détérioration face aux changements.

Tous ces types de maintenance ne permettent pas d'implémenter de nouvelles fonctionnalités qui exigent des modifications à l'architecture globale du système.

2.1.2 La modernisation

La modernisation est un ensemble de techniques qui permettent de résoudre les problèmes engendrés par les systèmes patrimoniaux en procédant à des changements importants dans le code. Elle peut se faire au niveau de l'interface utilisateur, de la logique fonctionnelle ou des données.

Il existe deux types de modernisation : modernisation de type "boîte noire" et modernisation de type "boîte blanche".

La modernisation de type "boîte noire" consiste à examiner les entrées et sorties du système patrimonial dans un contexte opératoire pour comprendre les interfaces du système. Elle ne comprend pas le fonctionnement interne du système et ne permet donc pas de l'étendre, de le restructurer ou de l'améliorer.

La modernisation de type "boîte blanche" consiste à restructurer le code en préservant le comportement externe du système. Pour ce faire, une compréhension du système patrimonial est nécessaire, notamment en dévoilant son architecture et sa conception interne. Les méthodologies et les outils de rétro-ingénierie sont souvent utilisés dans ce type de modernisation.

2.1.3 Le remplacement

Le remplacement d'un système est la création d'un nouveau système en utilisant des méthodologies et technologies modernes ainsi que de nouvelles exigences (Bennet, 1995).

Le remplacement peut être considéré pour plusieurs raisons, parmi lesquelles :

- ni la maintenance, ni les techniques de modernisation ne sont appropriées pour répondre aux nouveaux besoins d'affaires ;

— la documentation du système est inexistante ou non à jour;
— il n'y a plus ou peu de personnel qualifié qui comprend le fonctionnement interne du système.

Il faudra aussi mentionner que le remplacement est un projet risqué puisqu'il permet de développer un nouveau système qui doit inclure toutes les fonctionnalités du système patrimonial, qui doit être testé et validé, et qui va exiger des ressources humaines et matérielles non négligeables ainsi que des coûts relativement élevés.

Dans ce qui suit, nous allons décrire les différentes techniques qui permettent de réaliser l'activité de modernisation : le rhabillage, la réingénierie et la migration.

2.2 Les techniques de modernisation

Les techniques de modernisation peuvent être classifiées en trois catégories : le rhabillage, le redéveloppement/réingénierie et la migration (Bisbal *et al.*, 1999).

2.2.1 Le rhabillage

Le rhabillage (*wrapping*) est une technique de modernisation de type "boîte noire" qui peut prendre plusieurs formes et qui est utilisée quand les coûts ne justifient pas la réécriture ou la migration du système patrimonial.

Voici quelques techniques de rhabillage, fréquement utilisées :

— Le grattage d'écran (*Screen Scraping*) se concentre uniquement sur l'interface utilisateur du système patrimonial en exploitant ces flux d'entrée/sortie pour offrir une nouvelle interface utilisateur conviviale et moderne (web par exemple). De ce fait, cette technique de rhabillage ne s'occupe guère du fonctionnement interne du système patrimonial et donc ne permet pas d'améliorer sa qualité.

— La *réutilisation de composants réputés stables* et qui forment le socle du système patrimonial. Cette réutilisation passe par la définition de nouvelles interfaces qui cachent ou rendent transparent le fonctionnement de ces composants.

2.2.2 La réingénierie

La réingénierie consiste en l'examen (compréhension) et la modification d'un système pour le reconstituer dans une nouvelle forme puis l'implémentation subséquente de la nouvelle forme (Chikofsky et Cross, 1990).

La réingénierie peut inclure des activités comme l'ingénierie inverse, la restructuration, la re-conception et la ré-implémentation logicielle (Almonaies et al, 2010).

La réingénierie peut consister en une réécriture quasi-complète du système patrimonial et elle est censée améliorer sa qualité et sa maintenablité en utilisant les dernières avancées technologiques et les meilleures pratiques.

Il est à noter aussi que le nouveau système ne s'exécutera pas nécessairement sur un environnement différent de celui du système patrimonial.

2.2.3 La migration

La migration consiste à déplacer un système vers une nouvelle plateforme en conservant les fonctionnalités du système patrimonial et en minimisant autant que possible l'impact sur l'environnement opérationnel. La finalité distinctive de la migration est d'éviter un re-développement complet du système patrimonial en réutilisant autant que possible des parties de ce système. En plus, le système cible résultant du processus de migration s'exécute sur un environnement différent (nouveau langage ou bien nouvelle architecture et nouvelle technologie) (Bisbal

et al., 1999).

La migration concerne, donc, aussi bien l'application que la plateforme qui va l'héberger. La migration au niveau de l'application peut être faite au niveau de la présentation, de la logique métier ou de la couche d'accès aux données. La migration au niveau de la plateforme consiste à déterminer les composants matériels/logiciels (serveurs/services) sur lesquels cette application va être portée.

En décomposant le système patrimonial en couches (présentation, logique métier, accès aux données), la migration de l'application peut utiliser une combinaison des approches précédentes (rhabillage, réingénierie) pour construire le système cible.

2.3 L'architecture logique en couches

2.3.1 Définition

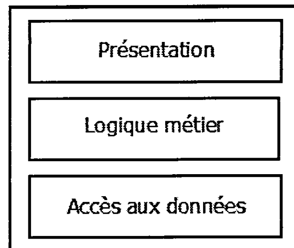
Une application se compose principalement de trois couches :

- *la couche présentation* qui permet à l'utilisateur final d'interagir avec le système ;
- *la couche logique métier* qui regroupe une implémentation de l'ensemble des règles d'affaires auxquelles l'application doit se conformer ;
- *la couche d'accès aux données* qui permet à l'application d'accéder aux sources de données externes (fichiers CSV, bases de données relationnelles, NoSQL, ...).

La communication entre les trois couches doit respecter les contraintes suivantes :

- *la couche présentation* communique avec *la couche logique métier* pour récupérer les informations à afficher à l'écran et pour permettre aux utilisateurs de réaliser les tâches programmées dans l'application ;

Figure 2.2 Architecture logique de l'application.

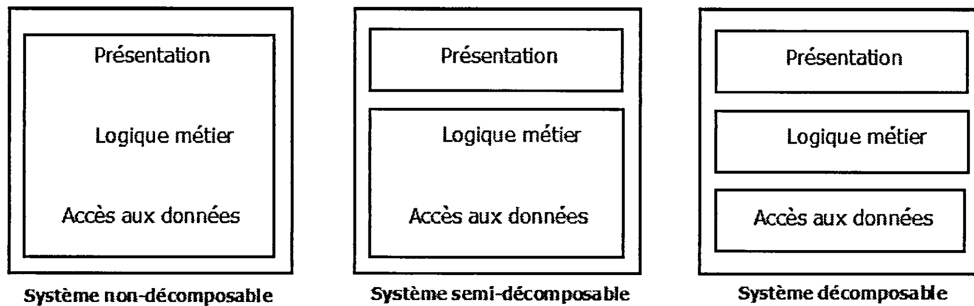


— *la couche logique métier* demande à *la couche d'accès aux données* de persister des données sur disque ou bien de retrouver des données du disque ;
— pour ne pas court-circuiter les règles d'affaires qui sont implémentées dans *la couche logique métier*, la *couche présentation* ne peut communiquer directement avec *la couche d'accès aux données.*

2.3.2 Le niveau de décomposition d'un système

La migration d'un système dépend de sa décomposition (Brodie et Stonebraker, 1995). En fonction du niveau de séparation entre les couches présentation, logique métier et accès aux données, un système patrimonial peut être classifié comme décomposable, semi-décomposable ou non-décomposable. Dans un système non-décomposable, les couches présentation, logique métier et accès aux données sont inséparables. Dans un système semi-décomposable, la couche présentation est séparée des couches logique métier et accès aux données qui elles sont inséparables. Dans un système décomposable, les trois couches présentation, logique métier et accès aux données sont séparables. La figure 2.3 illustre les trois niveaux de décomposition d'un système.

Figure 2.3 Niveaux de décomposition d'un système.



2.4 Le style architectural REST

REST (*REpresentational State of Transfer*) est un style architectural qui a été introduit par R. Fielding dans sa thèse de doctorat en l'année 2000 (Fielding, 2000). REST est largement adopté pour construire des APIs Web publiques basées sur le standard HTTP (*HyperText Transfer Protocol*).

2.4.1 Définition

Un système distribué à base de REST doit se conformer à un certain nombre de *contraintes* que nous allons décrire ci-après.

Client/Serveur

L'application est de type Client/Serveur où un client émet une requête et attend la réponse, et un serveur traite la requête et renvoie une réponse au client à travers le réseau de communications. Le client et le serveur utilisent le même *protocole* de communication pour échanger des données. Les applications Web qui reposent sur le protocole HTTP pour échanger des données sont de type Client/Serveur avec le navigateur Web comme client et le serveur d'applications/Web comme serveur. Cette contrainte renforce la *séparation des préoccupations* en permettant, entre autres, au client de gérer les interactions avec l'utilisateur final et de s'occuper de

l'affichage des données, et au serveur de s'occuper de la logique d'affaire.

Sans état

Les requêtes entre le client et le serveur sont indépendantes les unes des autres même si plusieurs requêtes sont nécessaires pour accomplir une tâche commune. Le serveur ne détient donc pas d'informations de session lorsqu'il communique avec un client. Chaque requête contient toutes les informations nécessaires et suffisantes pour que le serveur la traite et retourne une réponse. Cette contrainte renforce la *mise à l'échelle* de l'architecture sous-jacente en permettant d'ajouter des serveurs à la demande pour renforcer un trafic de plus en plus croissant. Elle permet aussi de renforcer la *disponibilité* de l'application en cas de panne d'un serveur puisque les autres serveurs vont pouvoir répondre aux requêtes subséquentes. Les applications Web reposent sur une grappe de serveurs Web qui sont derrière un répartiteur de charge et qui répondent aux requêtes émanant de clients distants.

Mise en cache

Les caches sont utilisés pour éviter qu'une même réponse soit renvoyée à un client plusieurs fois si les données contenues dans la réponse sont toujours valides. Ceci permet de conserver la bande passante mais aussi de soulager le serveur de traiter la même requête plusieurs fois. L'information mise en cache peut être stockée sur le serveur, sur le client ou sur un serveur intermidiaire qui sert de relais entre le client et le serveur. Les serveurs mandataires sont des exemples typiques de cache. Cette contrainte renforce donc la *mise à l'échelle* de l'application et aussi l'*amélioration des performances*. Les applications Web exposées à un niveau élevé de trafic ne peuvent ignorer cette contrainte.

Système en couches

Cette contrainte préconise que l'application soit organisée en couches pour renforcer la *séparation des préoccupations* et la *réutilisation du code*. Les couches les plus communément utilisées sont la présentation, la logique métier et la gestion des accès aux données.

Interface uniforme

Cette contrainte permet de faire une séparation nette entre les implémentations des APIs et des services qu'elles fournissent. Contrairement au protocole SOAP (*Simple Object Access Protocol*), la nomenclature de l'interface est indépendante des méthodes qui implémentent cette interface. Pour ce faire, cette contrainte consiste à utiliser un nombre restreint de méthodes standardisées et bien définies qui vont être utilisées par toutes les APIs. Comme l'utilisation de REST est intimement liée au protocol HTTP, les méthodes utilisées sont celles définies par HTTP et sont GET, POST, PUT, HEAD, OPTIONS et TRACE. Enfin, pour réaliser cette contrainte, REST spécifie quatre autres sous-contraintes à respecter dans une architecture. Ces sous-contraintes de l'interface uniforme sont traitées dans la section suivante.

2.4.2 L'interface uniforme

L'interface uniforme d'une API REST est elle aussi soumise à cinq sous-contraintes qui sont :

1. Identification des ressources par des URIs ;
2. Manipulation des ressources par des représentations ;
3. Messages auto-descriptifs ;
4. Utilisation d'un nombre restreint de méthodes ;

5. L'hypermédia comme moteur de l'état de l'application.

Identification des ressources

Chaque ressource est identifiée au moyen d'un URI unique sur le serveur. On énumère les types de ressources suivants :

— *ressource de base*

Une ressource de base désigne une seule ressource. Par exemple, l'URI `http://webrestpad.sofa.uqam.ca/ws/bestbuy` permet de référencer la ressource "bestbuy".

— *collection*

Une collection est un ensemble de ressources de même type. Par exemple, l'URI `http://webrestpad.sofa.uqam.ca/ws/apis` permet de référencer la ressource "liste de toutes les APIs publiques".

— *collection triée*

Une collection triée est une collection ordonnée selon des critères. Par exemple, l'URI `http://webrestpad.sofa.uqam.ca/ws/patterns?sort=asc` retourne la liste de patrons en ordre croissant.

— *sous-collection*

Une sous-collection est une collection à laquelle on applique des filtres. Par exemple, `http://webrestpad.sofa.uqam.ca/ws/patterns?type=P` permet de référencer la liste des patrons REST recensés dans le système.

— *processus*

Une ressource processus désigne une activité du système. Par exemple, l'URI `http://webrestpad.sofa.uqam.ca/ws/apis/bestbuy/patterns` permet de référencer l'activité ou l'algorithme de "détection des patrons d'une api". La représentation de la ressource correspond au résultat de l'exécution de l'algorithme.

Manipulation des ressources par des représentations

Une représentation correspond au contenu de la réponse HTTP et son format de données est encodé par un MIME-Type. Les formats de données les plus utilisés sont JSON, XML et (X)HTML ainsi que les formats d'images PNG et JPEG. Une même ressource peut alors avoir plusieurs représentations dépendamment du MIME-Type utilisé (JSON, XML, (X)HTML).

Messages auto-descriptifs

Le client et le serveur échangent des messages. Les messages correspondent à la requête du client et à la réponse du serveur. Dans HTTP, les messages correspondent aux méta-données définies dans l'entête de la requête ou de la réponse, plus le corps. Un message est auto-descriptif dans la mesure où il contient toutes les informations nécessaires et suffisantes pour son traitement.

Utilisation d'un nombre restreint de méthodes

Les méthodes utilisées sont celles du protocole HTTP et sont au nombre fixe de quatre. Une méthode est *sûre* si son invocation n'altère pas l'état de la ressource et elle est *idempotente* si on peut l'invoquer plusieurs fois et toujours recevoir le même état de la ressource.

— GET

La méthode GET permet de retrouver la représentation d'une ressource à partir du serveur. GET est sûre car elle n'a aucun effet sur l'état de la ressource, et elle est idempotente car on peut l'invoquer plusieurs fois et toujours produire le même état de la ressource.

— POST

La méthode POST permet, entre autres, de créer une nouvelle ressource sur le serveur, d'ajouter une sous-ressource à une collection de ressources

ou d'ajouter des données aux représentations d'une ressource. POST n'est ni sûre ni idempotente puisqu'elle crée une nouvelle ressource à chaque appel. À titre de comparaison avec l'orienté objet, cette méthode joue le rôle d'une fabrique d'objets.

— PUT

La méthode PUT permet de mettre à jour une ressource sur le serveur. PUT n'est pas sûre puisqu'elle change l'état de la ressource, mais elle est idempotente car on peut l'invoquer plusieurs fois et toujours produire le même état de la ressource.

— DELETE

La méthode DELETE permet de supprimer une ressource du serveur. DELETE n'est pas sûre puisqu'elle change l'état de la ressource, mais elle est idempotente car une fois la ressource supprimée elle n'a aucun effet.

L'hypermédia comme moteur de l'état de l'application (HATEOAS)

Le terme HATEOAS (*Hypermedia As Engine of Application State*) désigne une contrainte REST qui stipule que les échanges entre un client HTTP(S) et une application ou API Web doit se faire via des liens hypertextes qui sont fournis dynamiquement dans les réponses HTTP(S). L'état de l'application est donc déterminé par les représentations échangées entre le client et le serveur. Une représentation peut contenir des *liens* vers d'autres ressources et ces liens font passer l'application d'un état à un autre.

Dans une application Web (ou site Web), l'état de l'application est contrôlé par l'utilisateur en cliquant sur des liens hypertextes ou en soumettant un formulaire HTML. Dans ce cas, l'utilisateur est conscient des choix qui lui sont offerts et agit en conséquence pour faire passer l'application d'un état à un autre.

Dans une API REST qui respecte la contrainte HATEOAS, le client est un pro-

gramme qui doit faire ses choix en fonction de ce que le serveur lui retourne comme représentation. Dans ce cas, cette représentation doit inclure toutes les transitions (ou actions) possibles à partir de l'état actuel. Comment le client va t-il alors choisir le prochain état ?

Le client hypermédia doit connaitre tous les états possibles de l'application. En fonction de la réponse du serveur, le client doit être en mesure de proposer à l'utilisateur les prochains états possibles ainsi que le lien hypermédia à suivre pour atteindre cet état.

En résumé, Fielding définit REST comme étant un modèle architectural Client/-Serveur en couches avec une interface uniforme, un cache et qui est sans état.

Les APIs Web qui respectent la totalité de ces contraintes sont dites *RESTful.* Les APIs Web qui respectent toutes les contraintes sauf l'hypermédia sont dites *pragmatiques.*

2.4.3 Les patrons et anti-patrons REST

Les patrons REST sont des solutions communes qui permettent de respecter certaines contraintes alors que les anti-patrons sont de mauvaises solutions à certaines contraintes. La description détaillée des anti-patrons REST se trouve dans (Tilkov, 2008) et celle des patrons REST se trouve dans (Pautasso, 2009) et (Pautasso et Wilde, 2009). Les algorithmes de détection de ces patrons et anti-patrons sont décrits dans (Palma *et al.*, 2014).

Dans ce qui suit, nous allons présenter et définir ces patrons et anti-patrons REST.

Patrons REST

— *Entity Linking* Ce patron REST indique que la représentation retournée par

le serveur au client contient des liens hypertextes qui permettent au client de communiquer dynamiquement avec le serveur sans avoir à construire de nouvelles URIs lui même. Les liens hypertextes sont insérés dans l'entête du message via le paramètre `Location` ou bien dans le corps du message. Il existe des types MIME, notamment *HAL* (Kelly, 2016), qui formalisent l'utilisation des liens hypertextes dans le corps des messages HTTP.

— *Entity Endpoint* Un client a besoin de travailler avec deux identifiants : un identifiant global pour le service et un identifiant local pour l'entité ou la ressource gérée par ce service. En exposant chaque ressource avec un URI à l'intérieur d'un service, cette ressource sera adressable globalement avec un identifiant unique (Pautasso, 2009).

— *Content Negociation* Ce patron REST permet au client de négocier le format de la représentation qu'il préfère recevoir du serveur. Les APIs qui respectent ce patron implémentent plusieurs formats pour une même ressource afin d'être compatibles avec la pluspart des clients. Ceci assure un couplage faible et une interopérabilité accrue entre le client et le serveur (Pautasso, 2009).

— *Endpoint Redirection* Ce patron REST permet au serveur d'indiquer au client qu'une ressource n'est plus disponible dans son ancienne URI et qu'il existe une nouvelle URI pour référencer cette ressource. Pour ce faire, le serveur répond avec le code HTTP `301 Moved Permanently` et indique la nouvelle URI dans le paramètre `Location` de l'entête du message. Ceci permet au client de se rediriger automatiquement vers le nouveau URI et de prendre acte de ce changement sans modification du code pour s'adapter à la nouvelle nomenclature de l'API.

— *Response Caching* La mise en cache des ressources côté client est un patron REST qui permet de sauvegarder la bande passante et d'améliorer les performances. Le serveur peut utiliser dans l'entête du message les paramètres

`Cache-Control` et `Etag` pour activer la mise en cache. Les données restent valides jusqu'à ce que le serveur décide qu'elles ne le sont plus. Le code HTTP utilisé dans l'entête du message pour indiquer que les données de la ressource n'ont pas changé est `304 Not Modified`.

Anti-patrons REST

- *Tunneling everything Through GET* Au lieu d'utiliser la méthode GET pour retrouver la représentation d'une ressource comme sa sémantique HTTP l'exige, les développeurs l'utilisent pour des opérations qui modifient l'état de la ressource. Voici quelques exemples de cet anti-patron :
  1. `GET http://api.exemple.com/etudiants/45?action=supprimer`
  La méthode GET est utilisée pour supprimer l'étudiant 45.
  2. `GET http://api.exemple.com/etudiants/45?nom=Lajoie&action=modifier`
  La méthode GET est utilisée pour mettre à jour le nom de famille de l'étudiant 45.
  Dans les deux cas précédents, l'URI n'identifie pas la ressource mais encode une opération et ses paramètres.
- *Tunneling everything Through POST* Cet anti-patron est identique à l'anti-patron précédent à la difference que POST utilise le corps du message HTTP au lieu de l'URI pour encoder une opération et ses paramètres.
  1. `POST http://api.exemple.com/etudiants`
  `{ 'id' : 45, 'action' : 'retrouver' }`
  La méthode POST est utilisée pour retrouver l'étudiant 45.
  2. `POST http://api.exemple.com/etudiants`
  `{ 'id' : 45, 'action' : 'supprimer' }`
  La méthode POST est utilisée pour supprimer l'étudiant 45.
  Dans les deux cas précédents, la méthode POST n'est pas utilisée pour créer une nouvelle ressource comme l'exige sa sémantique HTTP, mais pour

encoder une opération et ses paramètres dans le corps du message HTTP.

- *Ignoring Caching* La mise en cache des ressources est une contrainte de REST qui améliore les performances et la mise à l'échelle des applications Web. Ignorer complétement cette contrainte est un anti-patron. Par exemple, si le serveur envoie cet entête dans la réponse HTTP :
  `Cache-Control: no-cache, no-store, must-revalidate`
  alors le client ne peut mettre la ressource en cache et doit la redemander au serveur à chaque fois.
- *Ignoring Status Code* HTTP dispose d'un ensemble de codes qu'il envoie dans l'entête de la réponse à une requête pour indiquer le résultat du traitement de cette requête. Ces codes sont groupés comme ceci :
  `2xx Succès`
  `3xx Redirection`
  `4xx Erreur du client`
  `5xx Erreur du serveur`
  Les exemples de code HTTP les plus utilisés par les développeurs sont les suivants :
  `200 OK` pour indiquer que le serveur a traité la requête avec succès
  `404 Not Found` pour indiquer que l'URI ne correspond à aucune ressource sur le serveur
  `500 Internal Server Error` pour indiquer que le serveur a rencontré une erreur suite au traitement de la requête.
- *Misusing Cookies* REST est un style srchitectural sans état dans la mesure où le client n'est pas lié à un serveur particulier pour exécuter une requête. De ce fait, un autre serveur peut répondre aux requêtes subséquentes au cas où le premier serveur est indisponible. L'utilisation des témoins (*cookies*) de session générés par le serveur lie le client avec le serveur et empêche donc la mise à l'échelle des applications Web. Ces témoins de session constituent

donc un anti-patron REST.

— *Forgetting Hypermédia* Cet anti-patron se manifeste quand le serveur ne retourne pas de liens dans les représentations des ressources. Le client doit donc toujours construire les liens hypermédias lui-même pour exécuter une action subséquente et faire passer une ressource d'un état à un autre. Le format *HAL* (Kelly, 2016), par exemple, peut être utilisé pour concevoir une API de type REST qui respecte la contrainte de l'hypermédia.

— *Ignoring MIME Types* Les types MIME (*Multipurpose Internet Mail Extensions*) sont les différents formats supportés par le serveur pour les différentes ressources retournées au client. Le mécanisme de négociation de contenu permet au client de prévilégier un format de représentation d'une ressource sur les autres que le serveur pourra retourner pour cette ressource. Un serveur qui offre plusieurs formats donne la possibilité au client de choisir son format préféré et de ce fait allège le travail des différents clients qui le sollicitent. Ignorer les types MIME constitue donc un anti-patron REST.

— *Breaking Self-descriptivness* Les entêtes, les formats d'échange et les protocoles utilisés dans les requêtes/réponses HTTP sont standardisés dans la spécification HTTP. Néanmoins, le protocole HTTP est flexible et permet de créer de nouveaux entêtes, formats ou protocoles spécifiques. En créant un nouveau protocole d'authentification, par exemple, tous les clients doivent s'assurer qu'ils ont le logiciel spécifique qui dialoguera avec ce protocole. Un client peut donc se retrouver à gérer plusieurs protocoles propriétaires pour s'authentifier aux différentes APIs avec lesquelles il communique. L'utilisation de tout ce qui est non standard ne respecte pas la contrainte des messages auto-descriptifs et constitue donc un anti-patron REST.

2.5 L'architecture SCA

2.5.1 Définition

SCA (*Service Oriented Architecture*) est un ensemble de spécifications qui permettent de créer des applications à base de composants. Les applications SCA sont donc un assemblage de composants. Chaque composant offre (ou expose) un certain nombre de services qui peuvent être utilisés par d'autres composants. Un composant peut donc aussi dépendre de services offerts par d'autres composants.

SCA définit la notion de *composite* pour configurer et connecter des composants qui composent une application SCA. En effet, **un composite est un fichier XML** de configuration décrit avec le langage SCDL (*Service Component Description Language*).

Les principales spécifications SCA sont les suivantes :

— *Le modèle d'assemblage SCA* pour spécifier comment configurer et connecter des composants ;
— *L'implémentation de composants* pour spécifier comment développer des applications dans des langages de programmation spécifiques ;
— Les *liaisons* pour spécifier comment accéder aux services Web distants.

2.5.2 Le composant

Un composant se défnit à partir des concepts suivants :

— Les *services* : ce sont les opérations accessibles au client du composant. La description des services d'un composant Java se fait dans des *interfaces* Java.

— Les *références* : ce sont des services offerts par d'autres composants qu'il utilise dans son implémentation. Une référence est une interface qui contient des opérations qui peuvent être invoquées par le composant.

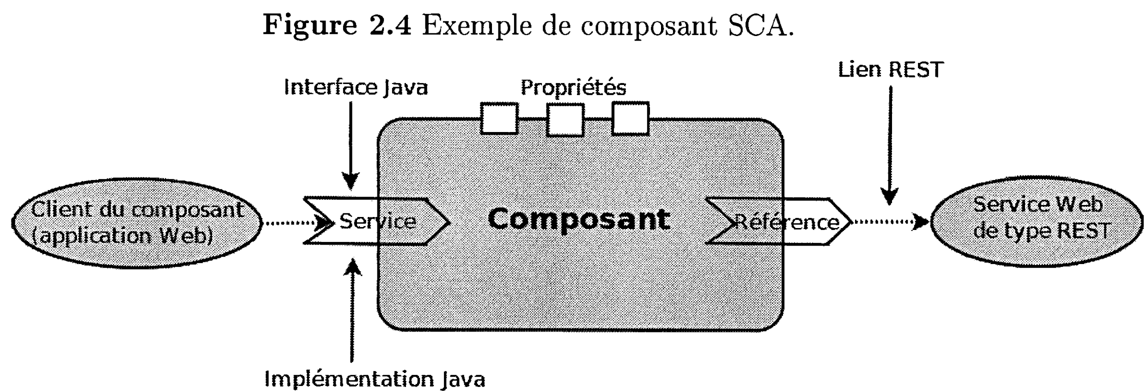
— Les *liens* : Un lien spécifie comment doit se faire une communication entre le composant SCA et un autre composant SCA ou application externe non conforme au standard SCA. Il définit un protocole particulier qui peut être utilisé pour communiquer avec un service ou une référence du composant (Chappell, 2007).

— Les *propriétés* : ce sont des variables qui contiennent des valeurs et qui sont définies dans le composite.

— L'*implémentation* : c'est le code correspondant à la logique métier du composant.

La figure 2.4 met en perspective les notions SCA présentées.

Figure 2.4 Exemple de composant SCA.



CHAPITRE III

COMPRÉHENSION DU SYSTÈME PATRIMONIAL

Dans ce chapitre, nous allons décrire la première étape du processus de migration qui est la compréhension du système patrimonial.

3.1 Introduction

Le système patrimonial est une application Java de bureau qui permet, via une interface de ligne de commande, de détecter les patrons et anti-patrons REST des APIs HTTP publiques. Cette détection se fait en quatre étapes successives (Palma *et al.*, 2014) :

1. Analyse des descriptions des patrons et anti-patrons REST pour identifier les propriétés qui les caractérisent ;
2. Définition des heuristiques de détection pour les patrons et anti-patrons REST, les heuristiques étant des propriétés qui caractérisent chaque (anti)patron ;
3. Extension du cadre de travail SOFA (*Software Oriented Framework For Analysis*) pour implémenter les algorithmes de détection associés aux heuristiques ;
4. Validation
   — Appliquer ces algorithmes sur des APIs REST en invoquant des mé-

thodes à l'intérieur de ces APIs ;
— Rapporter la liste des services REST détectés comme des patrons et ceux détectés comme des anti-patrons.

L'interface utilisateur de l'application affiche la liste des APIs parmi lesquelles l'utilisateur en choisit une. Une fois l'API sélectionnée, le système exécute l'algorithme de détection des (anti)patrons et affiche le résultat de cette détection, à savoir la liste des (anti)patrons détectés. Par ailleurs, les traces de cette exécution sont enregistrées dans un fichier séparé sur disque.

3.2 Analyse du système patrimonial

Tel qu'indiqué dans la section *Méthodologie* du chapitre 1, l'analyse du système patrimonial consiste à en déduire le modèle des cas d'utilisation, les concepts du domaines et leurs relations ainsi que le diagramme des classes.

3.2.1 Le modèle des cas d'utilisation

En exécutant les scénarios possibles de l'application, on a trouvé un seul cas d'utilisation du système illustré dans le diagramme de la figure 3.1. Ce diagramme donne une vue d'ensemble des fonctionnalités du système.

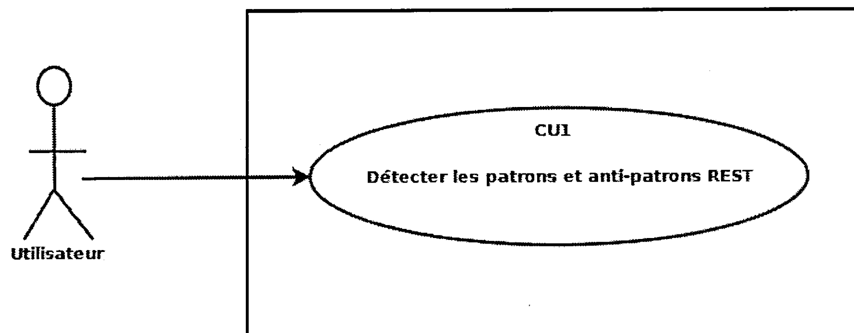
Le tableau 3.1 décrit textuellement le cas d'utilisation CU1.

3.2.2 Les concepts du domaine

À partir des cas d'utilisation, on peut en déduire les concepts du domaine suivants :

- api
- ressource

Figure 3.1 Diagramme des cas d'utilisation.



Cas d'utilisation	**UC1 : Détecter les patrons et anti-patrons REST d'une API**
Acteurs	Utilisateur
Description	L'utilisateur teste la qualité d'une API en détectant ses patrons et anti-patrons REST
Cas d'utilisation liés	
Préconditions	
Postconditions	
Scénario normal	**1.0 Détecter les patrons et anti-patrons d'une API Web publique** 1. L'utilisateur sélectionne l'API à tester 2. L'utilisateur demande au système d'effectuer la détection des patrons et d'anti-patrons 3. Le système affiche le résutat de la détection, en l'occurence la liste des patrons et anti-patrons détectés 4. Le système enregistre dans un fichier texte les traces de détection associées à chaque ressource (caractéristiques de chaque (anti)patron de cette ressource)
Scénarios alternatifs	
Exceptions	
Priorité	Haute

Tableau 3.1 Description du cas d'utilisation CU1.

— patron

— anti-patron

Les relations entre les concepts du domaine sont les suivantes :

— Une API a plusieurs ressources

— Une ressource peut avoir plusieurs patrons ou plusieurs anti-patrons

3.2.3 Le diagramme des classes

Nous allons présenter deux diagrammes de classes, un diagramme pour les classes importantes participant à la détection des (anti)patrons (Figure 3.2) et un diagramme des classes participant à l'affichage des résultats (Figure 3.3). Ces deux diagrammes ont été obtenus en analysant le code et en utilisant la fonctionnalité de rétro-conception de l'outil Visual Paradigm.

Les classes qui participent à la détection des (anti)patrons

— *Launcher*

Permet de lancer l'exécution du programme.

— *RESTPadManager*

Implémente la détection des (anti)patrons au niveau de l'API.

— *Motif*

Implémente les heuristiques qui caractérisent chaque patron et anti-patron.

— *AMotif*

Implémente la détection au niveau de la ressource.

— *LexicalResult*

Stocke les résultats de la détection au niveau de chaque ressource.

— *Reliability*

Définit les propriétés de validation de la détection des (anti)patrons.

Les classes qui participent à l'affichage des résultats

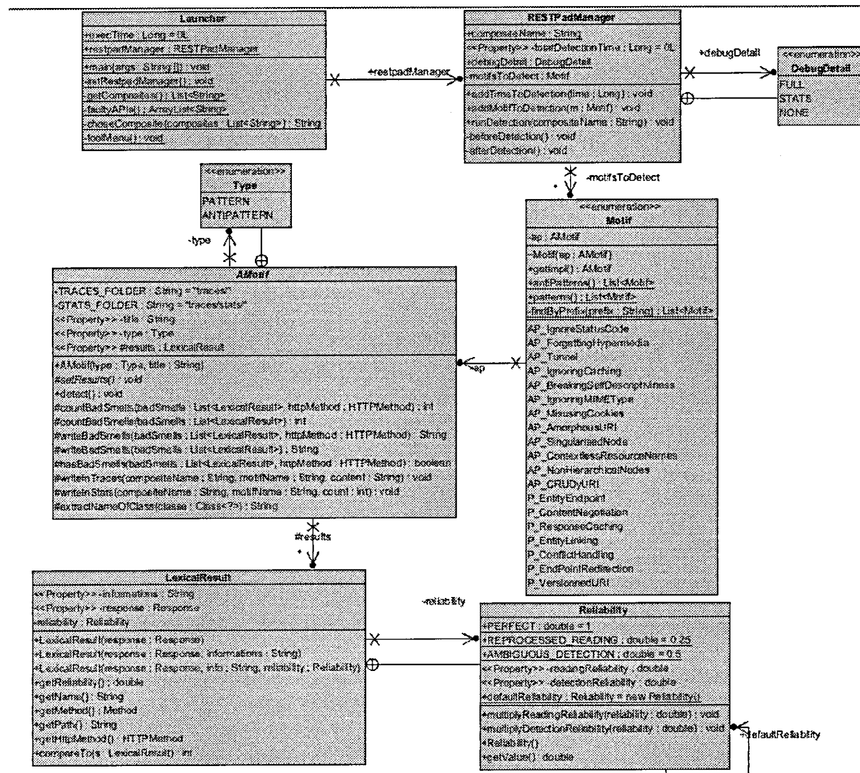
— *Pattern*

Définit un (anti)patron (entité patron).

— *Call*

Enregistre pour chaque ressource les statistiques de validation de l'algorithme de détection des (anti)patrons.

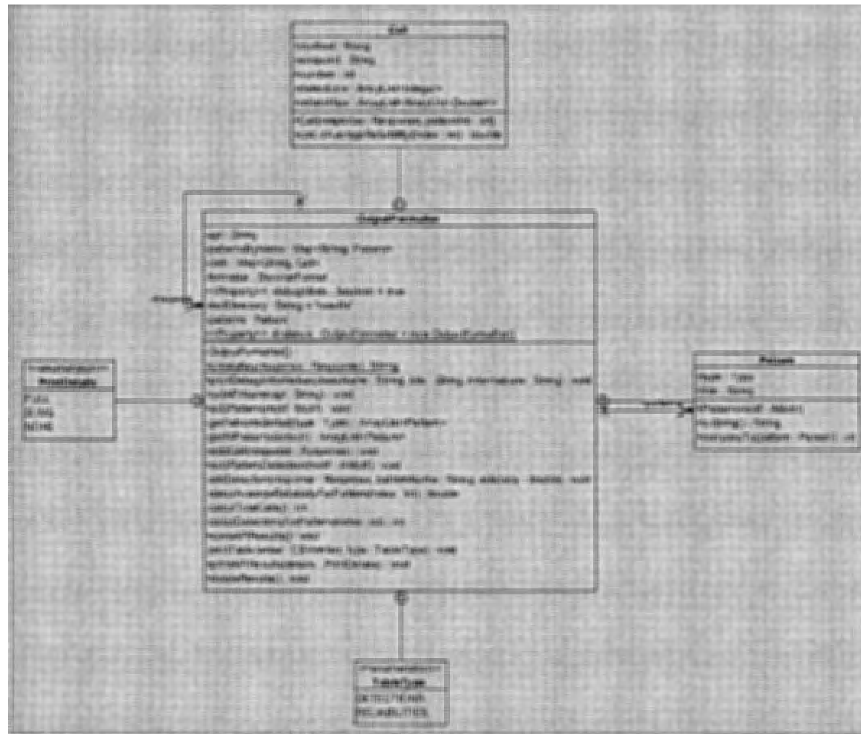
Figure 3.2 Diagramme des classes qui participent à la détection des (anti)patrons.



— *OutputFormatter*

Récupère les résultats de détection, les formate et les affiche à l'écran.

Figure 3.3 Diagramme des classes qui participent à l'affichage des résultats.



3.3 Ajout manuel d'une nouvelle API au système

Pour ajouter (ou prendre en compte) une nouvelle API dans le système, il faut passer par les trois étapes suivantes :

— S'enregistrer auprès du fournisseur de l'API, via son site Web, pour recevoir les jetons de connexion à cette API ;

— Sélectionner un échantillon de ressources de l'API en consultant sa documentation en ligne ;

— Créer un composant SCA pour détecter les (anti)patrons des ressources sélectionnées de cette API.

Pour illustrer ces étapes, on va ajouter une nouvelle API qui traite des données ouvertes pour la ville de Chicago en utilisant la plateforme Socrata.

Cette API fournit plusieurs ensembles de ressources de type collection dits *datasets* sur lesquels on peut faire du filtrage et de la pagination. L'URI de chaque ensemble de ressources est :

`https://data.cityofchicago.org/resource/`$\{DATASET\}$`.json`

Nous allons utiliser trois ensembles de ressources pour construire notre échantillon de ressources :

1. *energy-usage-2010*

   Cet ensemble de ressources contient des données concernant l'utilisation de l'énergie en 2010 par les maisons résidentielles, les entreprises et les industries.

   La documentation d'utilisation des ressources se trouve à l'adresse suivante :

   `https://data.cityofchicago.org/Environment-Sustainable-Development/Energy-Usage-2010/8yq3-m6wp`

2. *historical-traffic-congestion-region*

   Cet ensemble de ressources contient des données concernant l'historique des estimations de congestion pour les 29 zones de circulation.

   La documentation d'utilisation des ressources se trouve à l'adresse suivante :

   `https://data.cityofchicago.org/Transportation/Chicago-Traffic-Tracker-Historical-Congestion-Esti/emtn-qqdi`

3. *alternative-fuel-locations*

   Cet ensemble de ressources contient des données concernant les stations de carburants alternatifs (biodiesel, gaz naturel compressé, éthanol, chargement électrique, hydrogène, propane).

La documentation d'utilisation des ressources se trouve à l'adresse suivante :

https://data.cityofchicago.org/Environment-Sustainable-Development/Alternative-Fuel-Locations/f7f2-ggz5/data

3.3.1 S'enregistrer auprès du fournisseur de l'API

Pour utiliser les ressources de l'API, une application a besoin de s'enregistrer auprès du fournisseur de l'API. Une fois enregistrée, l'accès à l'API peut être fait via un jeton applicatif ou via un mécanisme d'authentification.

L'accès par jeton applicatif est utilisé uniquement pour des opérations de lecture en utilisant exclusivement une requête GET qui n'altère pas l'état de la ressource en question. Les opérations de création (POST), de modification (PUT, PATCH) ou de suppression (DELETE) ne sont pas autorisées. Ce type d'accès est fourni par des APIs qui exposent des données publiques et qui ne permettent pas à un usager de modifier l'état des ressources de cette API.

Quand l'application veut modifier l'état des ressources d'une API, une authentification au nom d'un utilisateur est alors exigée par le fournisseur d'API. Les mécanismes d'authentification à une API Web utilisent alors l'authentification de base HTTP, HMAC, oAuth 1.0a ou oAuth 2.0.

Comme nous n'allons pas modifier l'API Socrata, nous allons utiliser le mécanisme du jeton applicatif pour récupérer les différentes ressources faisant partie de l'échantillon sélectionné. Socrata nous permet d'utiliser le même jeton d'application pour accéder à tous les ensembles de ressources et on peut faire jusqu'à 1000 requêtes par heure.

Le jeton de l'API peut être passé dans l'entête HTTP(S) de la requête ou bien comme paramètre d'URL :

— si le jeton est passé dans l'entête HTTP(S) de la requête, alors il correspond à la valeur du paramètre **X-App-Token**

```
Host: data.cityofchicago.org
Accept: application/json
X-App-Token: Fo2vflcqEPRJNAVYO8iEkNZoU
```

— si le jeton est passé comme paramètre d'URL, alors il correspond à la valeur du paramètre **$$app_token**

```
data.cityofchicago.org?$$app_token=Fo2vflcqEPRJNAVYO8iEkNZoU
```

Le token utilisé dans chacun des exemples précédents est **Fo2vflcqEPRJNAVYO8iEkNZoU**.

3.3.2 Sélectionner un échantillon de ressources de l'API

Voici l'échantillon de ressources sélectionnées à partir des trois ensembles précédents :

Ressource 1 :

```
https://data.cityofchicago.org/resource/energy-usage-2010.json?
building_type=Residential
```

Cette ressource est une collection de ressources.

Ressource 2 :

```
https://data.cityofchicago.org/resource/energy-usage-2010.json?
building_type=Residential&building_subtype=Single%20Family&%24
where=kwh_january_2010%20%3E%200&%24limit=10&%24offset=20
```

Les paramètres `limit` et `offset` indiquent que cette ressource est une collection paginée de ressources.

Ressource 3 :

`https://data.cityofchicago.org/resource/historical-traffic-congestion-region.json?`
`$$app_token=Fo2vflcqEPRJNAVY08iEkNZoU&time=2015-01-12T21%3A20%3A55`
`®ion_id=29&bus_count=20`

Cette ressource est une collection de ressources.

Ressource 4 :

`https://data.cityofchicago.org/resource/alternative-fuel-locations.json?`
`$$app_token=Fo2vflcqEPRJNAVY08iEkNZoU&fuel_type_code=LPG`

Cette ressource est une collection de ressources.

3.3.3 Créer le composant SCA

La création d'un nouveau composant SCA associé à l'API consiste à créer :

1. un fichier XML de configuration dit *composite* qui est un descripteur d'architecture pour cette API ;
2. une *interface* Java définissant les méthodes d'utilisation de cette API, chaque méthode étant un service Web REST ;
3. une classe *client* qui implémente les méthodes de l'interface Java et qui exécute les services Web REST.

Créer le fichier composite FraSCAti

La composition de services est faite via le fichier composite XML qui utilise un langage de description architecturale pour SCA nommé SCDL (*Service Component Definition Language*). La figure 3.4 montre le fichier composite de l'API Socrata.

Dans tous les exemples qui suivent, nous utiliserons *opendata* pour référencer l'API Socrata.

Figure 3.4 Le fichier composite FraSCAti pour l'API opendata.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <sca:composite xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
3     xmlns:frascati="http://frascati.ow2.org/xmlns/sca/1.1" name="
        opendata">
4      <sca:service name="r" promote="client/Runnable" />
5      <sca:component name="client">
6          <sca:implementation.java class="ca.uqam.sofa.opendata.
             lib.Client"/>
7          <sca:reference name="opendata" promote="client/opendata
             ">
8              <sca:interface.java interface="ca.uqam.sofa.opendata
                 .api.Opendata"/>
9              <frascati:binding.rest uri="https://data.
                 cityofchicago.org/resource"/>
10         </sca:reference>
11         <sca:property name="base_uri">https://data.cityofchicago
             .org/resource</sca:property>
12         <sca:property name="app_token">Fo2vflcqEPRJNAVY08iEkNZoU
             </sca:property>
13         <sca:property name="secret_token">tdqaPemQ5rtYJYVgMrM-
             kk0vBElgCCXik9hh</sca:property>
14     </sca:component>
15 </sca:composite>
```

La description de ce fichier composite est la suivante :

— Le nom du composite est opendata.

```
2 <sca:composite ... name="opendata">
```

- Ce composite définit un seul service nommé `r` via l'élément XML **sca :service**. Ce service `r` promeut le service `Runnable` du composant `client` pour être un service visible dans ce composite.

  ```
  4 <sca:service name="r" promote="client/Runnable" />
  ```

- Ce composite définit un seul composant nommé `client`.

  ```
  5 <sca:component name="client" >
  ```

Le composant nommé `client` :

- indique via l'élément XML **sca :implementation.java** que la technologie utilisée pour l'implémentation de ce composant est Java et que cette implémentation se trouve dans la classe `ca.uqam.sofa.opendata.lib.Client` ;
- définit une référence nommée `opendata` qu'il va utiliser pour invoquer des services distants. Cette référence `opendata` promeut le service `opendata` du composant `client` pour être un service visible dans ce composant. Les opérations utilisées pour invoquer les services distants sont définies dans une interface Java nommée `ca.uqam.sofa.opendata.api.Opendata` :

  ```
  8 <sca:interface.java interface="ca.uqam.sofa.opendata.api.Opendata"/>
  ```

  Pour décrire le protocole utilisé pour invoquer les services distants, le composant définit un lien REST via l'élèment XML **frascati :binding.rest** et indique l'URL racine comme ceci :

  ```
  9 <frascati:binding.rest uri="https://data.cityofchicago.org/resource"/>
  ```

- définit trois propriétés, via l'élément XML **sca :property**, nommées `base_uri`, `app_token` et `secret_token` avec une valeur pour chacune d'elles.

  ```
  11 <sca:property name="base_uri">https://data.cityofchicago.org/resource</sca:property>
  12 <sca:property name="app_token">Fo2vflcqEPRJNAVYO8iEkNZoU</sca:property>
  13 <sca:property name="secret_token">tdqaPemQ5rtYJYVgMrM-kkOvBElgCCXik9hh</sca:property>
  ```

Créer l'interface définissant les méthodes d'utilisation de l'API

L'interface de l'API Socrata est décrite dans la figure A.1 (p. 81).

Les méthodes définies dans l'interface décrivent les ressources et les actions sur ces ressources. Elles sont annotées par `@GET`, `@Path`, `@Produces` et `@QueryParam`.

La description de ces annotations est la suivante :

- `@GET` annote une méthode Java qui va traiter des requêtes HTTP GET ;
- `@Path` indique l'URI relative de la ressource ;
- `@Produces` spécifie le type MIME qu'une méthode peut produire ;
- `@QueryParam` permet d'extraire les paramètres de l'URI.

Les annotations sont exploitées à l'exécution pour générer les "*helper classes*" d'une ressource.

Créer la classe Client

La classe Client (Fig. A.2, p. 82) permet d'instancier les appels (requêtes HTTP) vers l'API. Ces appels correspondent à l'échantillon des quatre ressources sélectionnées auparavant parmi toutes les ressources disponibles.

CHAPITRE IV

MIGRATION DU SYSTÈME PATRIMONIAL VERS LE SYSTÈME CIBLE

Dans ce chapitre, nous allons décrire la deuxième étape du processus de migration qui consiste à :

— définir l'*architecture cible* ;

— mettre en œuvre de la *migration* des données, de la logique métier et de la présentation du système patrimonial vers le système cible ;

— et automatiser l'*ajout d'une nouvelle API* au système en utilisant un système de gabarits qui va permettre de générer automatiquement le code lié à cette API.

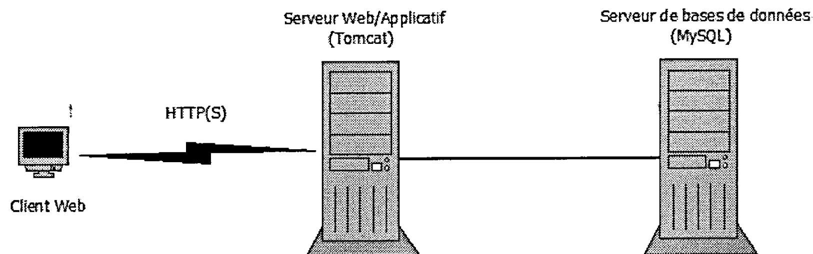
4.1 L'architecture cible

Dans cette section, nous allons décrire l'architecture du système cible selon plusieurs vues : une vue physique et plusieurs vues logiques.

4.1.1 La vue physique

La vue physique (Figure 4.1) permet de présenter les machines qui font partie du contexte d'exécution de l'application.

Figure 4.1 Architecture physique de l'application.



Dans cette architecture dite à trois niveaux, on a trois systèmes qui entrent en jeu pour faire aboutir une fonctionnalité de l'application :

— un client Web

— un serveur Web applicatif

— un serveur de bases de données relationnelles

Le client Web est la machine utilisée par l'utilisateur pour exécuter une tâche dans l'application. Cette machine peut être un ordinateur différent du serveur, une tablette ou un tétéphone mobile mais le plus important est qu'elle contient un logiciel qui va agir en tant que client HTTP.

Le serveur Web applicatif doit intercepter les requêtes des clients Web et leur retourner des pages Web. En tant que serveur Web, il analyse la requête pour déterminer quelle est la partie du code qui va la prendre en charge. Et en tant que serveur applicatif, il exécute les traitements et peut être amené à déléguer une partie de ces traitements à un système externe comme un serveur de bases de données.

Le serveur de bases de données relationnelles s'occupe de la gestion des données. Il garantit l'intégrité des données et leur disponibilité. Il permet aussi d'exécuter des requêtes SQL et de retouner les résultats aux clients. On peut interroger

le serveur via une commande SQL (*Structured Query Language*), via l'interface de programmation JDBC (*Java Database Connectivity*) ou via une interface de programmation ORM (*Object-Relational Mapping*).

4.1.2 Les vues logiques

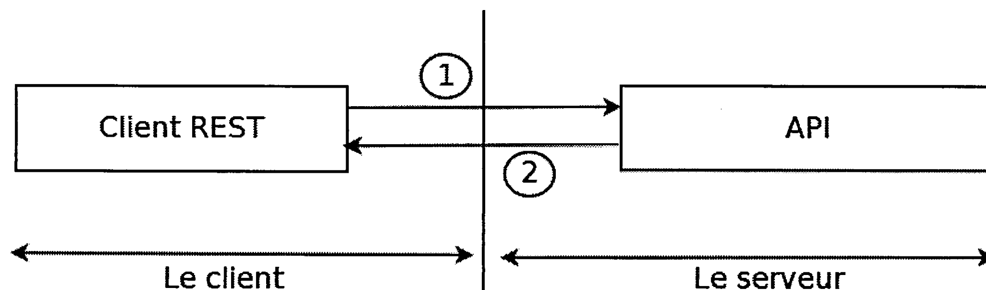
Une vue logique permet de présenter l'application selon le modèle en trois couches d'abstraction introduit au chapitre 2 :

— la couche présentation
— la couche logique métier
— la couche d'accès aux données

Vue logique 1 : Utilisation de l'API par un client REST

La figure 4.2 montre l'utilisation de l'API par un client REST.

Figure 4.2 Utilisation de l'API par un client REST.

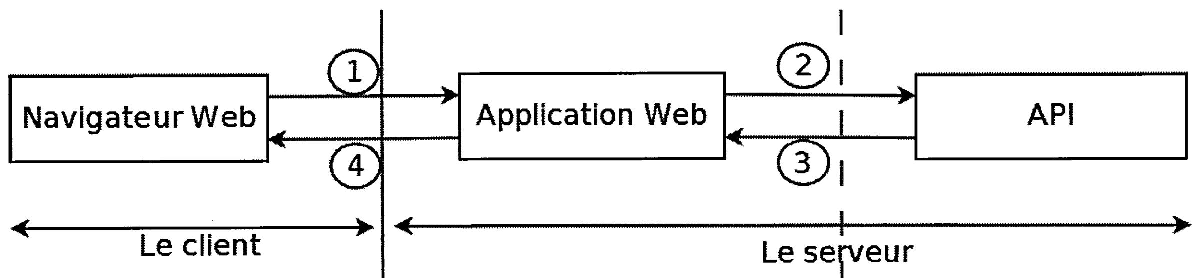


1. Le client REST fait une requête HTTP à l'API en spécifiant l'URI de la ressource ;
2. L'API décode l'URI, exécute la logique métier et retourne une représentation de la ressource au client REST.

Vue logique 2 : Utilisation de l'API par l'application Web

La figure 4.3 montre l'utilisation de l'API par l'application Web en invoquant des appels AJAX (*Asynchronous Javascript and XML*) asynchrones.

Figure 4.3 Utilisation de l'API par l'application Web.



1. Le navigateur Web interroge l'application Web en utilisant un contrôle HTML (lien hypertexte, bouton de formulaire ou appel Ajax asynchrone) ;
2. L'application Web fait appel à l'API pour exécuter la logique métier ;
3. L'API exécute la logique métier et retourne le résultat sous format JSON ;
4. L'application Web construit une page HTML avec les données JSON et la retourne au navigateur Web pour affichage.

Vue logique 3 : Architecture multi-niveau de l'application

La figure 4.4 montre une vue de l'architecture multi-niveau de l'application.

La couche présentation consiste en un ensemble de pages Web dans lesquelles on trouve du texte formaté, des formulaires de saisie ainsi que des contrôles HTML (liens hypertextes, boutons) pour accéder à d'autres pages.

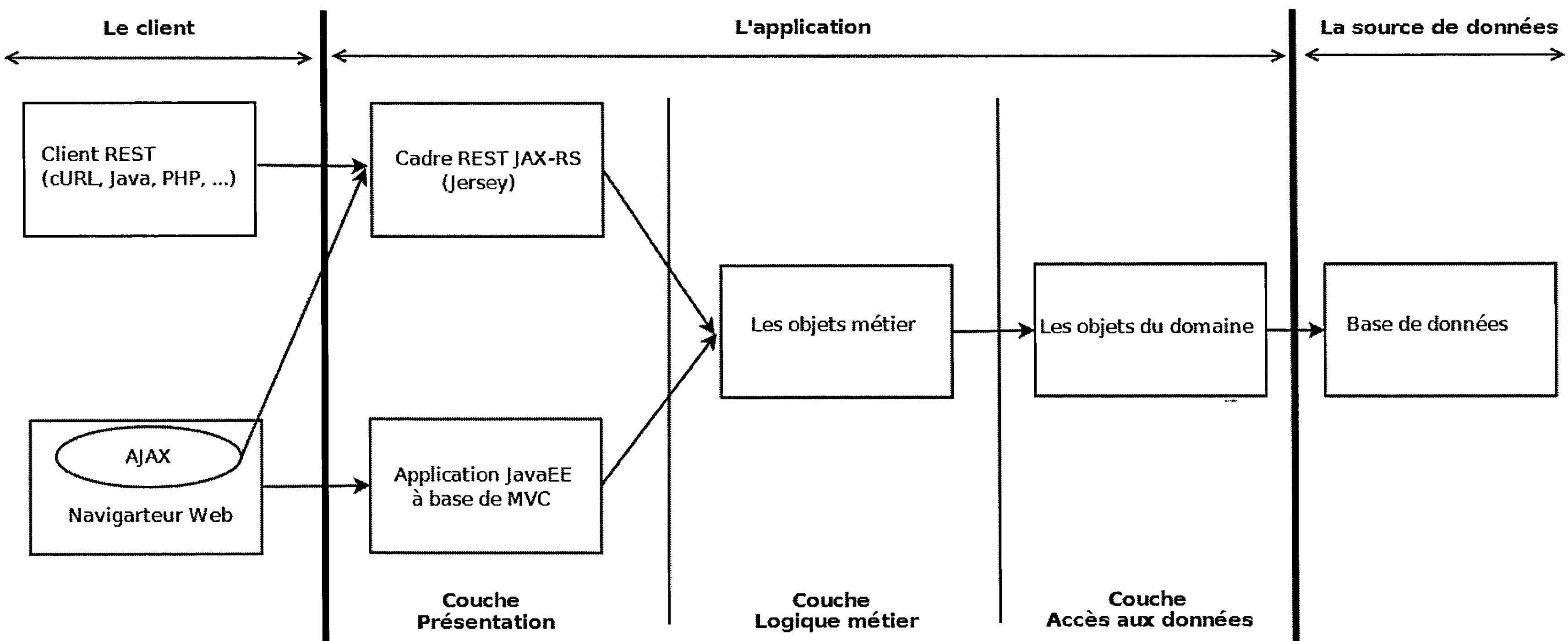
La logique de présentation est implémentée côté serveur et côté client (navigateur). En effet, en accédant à une URL de l'application, le client télécharge une page Web HTML depuis le serveur. Cette page Web contient une logique de présentation

pour le formattage de la page (CSS) ainsi que la validation des champs de saisie des formulaires (Javascript). La validation des données se fait d'abord côté client pour éviter des interactions réseau inutiles avec le serveur, et ensuite côté serveur pour se conformer aux règles d'affaires implémentées dans la logique métier.

La couche logique métier implémente les règles d'affaires sous forme d'objets métier qui seront invoqués soit dans la couche présentation, soit par d'autres objets métier, soit par un client distant (service Web). Les objets métier ne sont pas persistants.

La couche d'accès aux données implémente les objets du domaine. Ces objets sont persistants dans une source de données et ils sont invoqués par la couche métier.

Figure 4.4 Architecture multi-niveau de l'application.



Vue logique 4 : Architecture en couche et patron architectural MVC

Les applications Web Java EE sont basées sur le patron architectural MVC (Modèle-Vue-Contrôleur) :

- Le modèle s'occupe de la logique métier et gère les données de l'application. Il n'a aucune interaction avec l'utilisateur et n'affiche aucun résultat à l'écran ;
- La vue gère la partie visuelle de l'application. Elle affiche les résultats qu'elle récupère du contrôleur ou du modèle (selon les implémentations). Dans les applications Web, la visualisation est réalisée en utilisant HTML, CSS et JavaScript ;
- Le contrôleur (1) gère l'interaction avec l'utilisateur, (2) mandate le modèle d'appliquer la logique métier puis (3) passe les résultats à la vue pour affichage.

Dans une architecture MVC, le modèle englobe à la fois la couche métier et la couche d'accès aux données. Aussi, le contrôleur et la vue font partie de la couche présentation du modèle en couches.

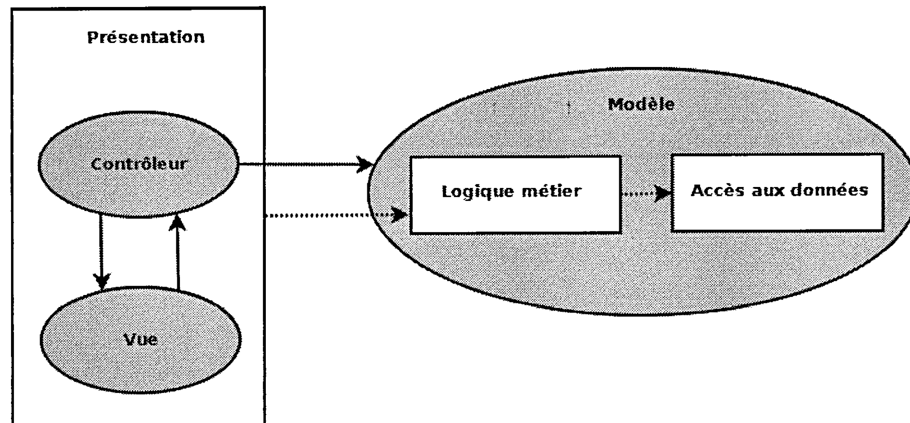
La figure 4.5 montre une vue de l'architecture en couches en contraste avec le patron architectural MVC.

4.1.3 Structure de nommage des packages

Un package est un espace de noms qui permet d'organiser un ensemble de classes qui sont en relation les unes avec les autres. À ce package correspond une structure hiérarchique dans le système de fichiers.

Il existe principalement deux approches pour structurer les packages, par couche ou par fonctionnalité. L'approche par couche permet de structurer les packages

Figure 4.5 Architecture en couche et patron architectural MVC.



par couche d'abord puis par fonctionnalité ensuite. À l'inverse, l'approche par fonctionnalité permet d'abord de structurer les packages par fonctionnalité puis par couche ensuite.

Dans l'approche par couche, toutes les fonctionnalités relatives à une couche du modèle architectural seront dans le package associé à cette couche. C'est cette approche que nous allons adopter.

Voici donc un extrait de la structure des packages de notre application :

Par couche d'abord

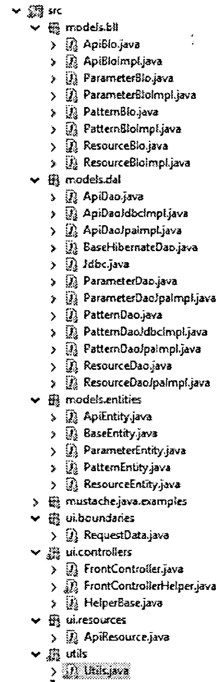
`models.entities`	Les objets du domaine
`models.dal`	Les objets de la couche d'accès aux données
`models.bll`	Les objets de la couche logique métier
`ui.boundaries`	Les objets de transfert de données
`ui.controllers`	Les objets contrôleurs Web
`ui.resources`	Les objets ressources REST

Puis en fonctionnalité ensuite

La figure 4.6 montre la structure des packages en couches d'abord puis en fonc-

tionnalité ensuite.

Figure 4.6 Structure de nommage des packages.



4.1.4 Choix techniques

Pour créer l'application Web et les services Web de type REST, nous avons opté pour les technologies Java EE suivantes :

- les *servlets* pour implémenter les contrôleurs ;
- les *JSP/JSTL* pour implémenter les vues ;
- le cadriciel *Jersey* pour implémenter l'API de type REST ;
- l'interface *JDBC* ainsi que le cadriciel *Hibernate* pour implémenter la persistance des données.

Le cadriciel *Jersey* est une implémentation du standard JAX-RS (*Java API for RESTful Web Services*) pour les services Web de type REST.

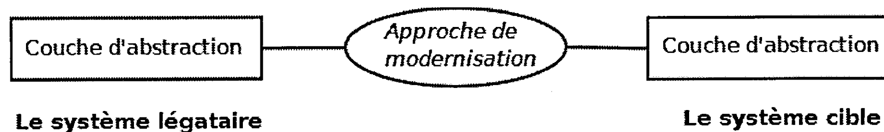
Le cadriciel *Hibernate* est une implémentation du standard JPA (*Java Persistence API*).

4.2 La stratégie de migration

La migration d'un système dépend de sa décomposition (Brodie et Stonebraker, 1995).

En fonction du niveau de décomposition (*decomposability*) du système patrimonial, la stratégie de migration consiste à choisir quelle approche de modernisation (rhabillage, ré-ingénierie, migration) à utiliser pour chacune des couches de l'application (présentation, logique métier, accès aux données) (Figure 4.7).

Figure 4.7 Modernisation d'une couche d'abstraction.



Les techniques de modernisation opérées au niveau de chaque couche sont les suivantes (Figure 4.8) :

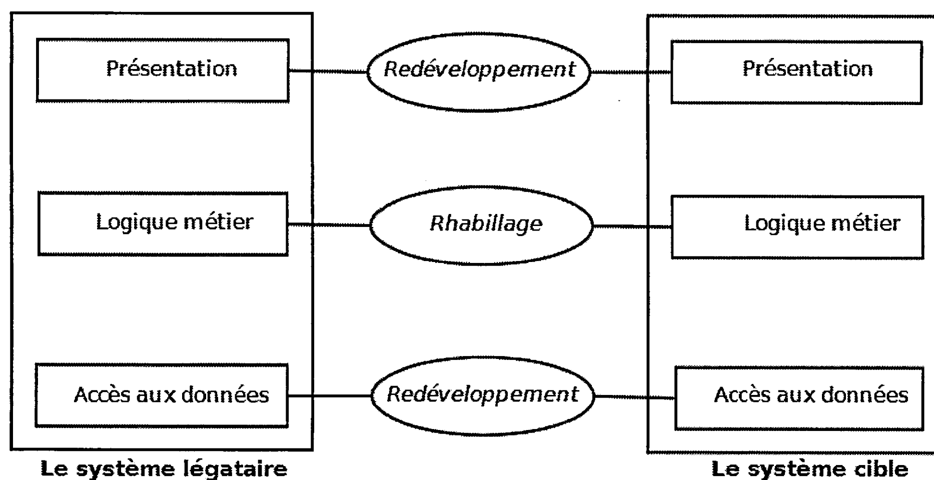
- Présentation : *Redéveloppement*
  Le système patrimonial est une application de bureau basée sur les technologies Java, alors que le système cible est une application Client/Serveur basée sur les technologies Web Java EE. L'environnement d'exécution est donc différent. Les applications Web utilisent HTML comme format standard d'affichage des pages Web et le rendu est fait par un navigateur Web. Les APIs de type REST utilisent JSON ou XML comme représentation dans les réponses HTTP.
- Logique métier : *Rhabillage*

Dans cette couche, nous avons développé la couche métier propre au nouveau système et faire appel au code métier du système patrimonial qui permet la détection des patrons et anti-patrons REST.

— Données : *Redéveloppement*

Dans cette couche, nous avons utilisé une base de données au lieu d'un fichier plat au format CSV (*Comma-Separated Values*). Nous avons aussi développé la couche d'accès aux données et l'isoler de la couche métier.

Figure 4.8 Stratégie de migration en couches.



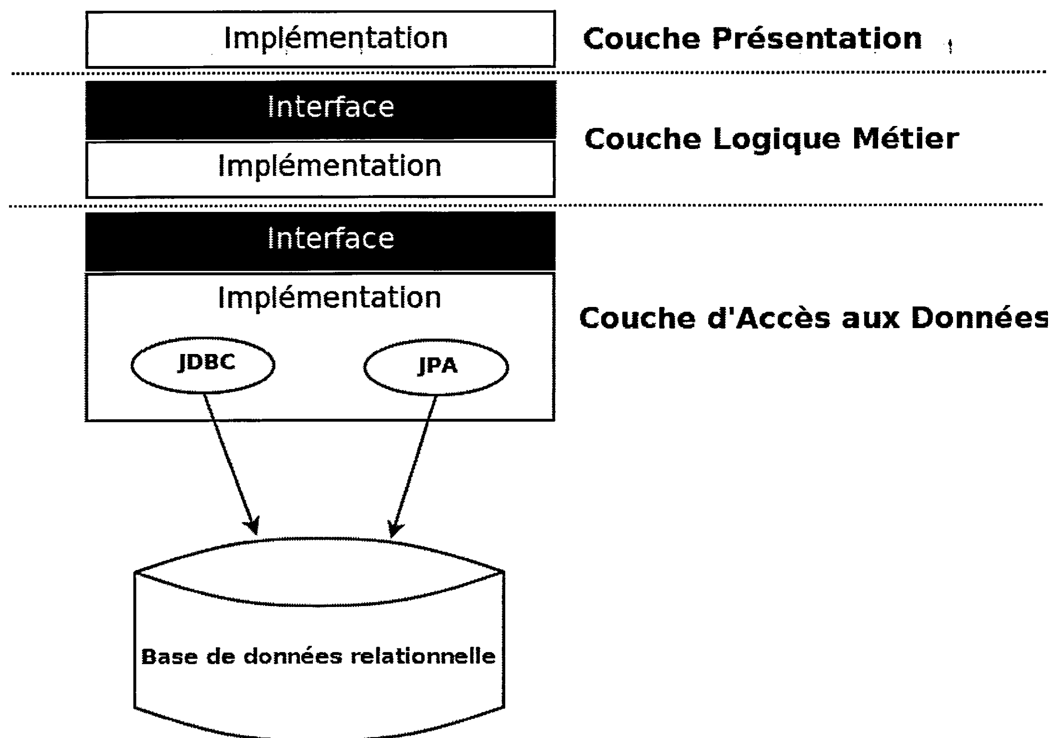
Nous avons commencé par migrer l'application vers une API sous forme de services Web de type REST. Ensuite, nous avons utilisé cette API pour développer une application Web ayant une interface HTML conviviale pour consommer les services REST de cette API.

4.3 Implémentation en couches

Dans une architecture en couches, chaque couche utilise les services de la couche inférieure adjacente. Pour cela, la couche inférieure propose une interface commune pour utiliser ses services indépendamment des autres couches inférieures. La figure

4.9 montre l'implémentation en couches que nous avons adopté.

Figure 4.9 Implémentation en couches.



La couche présentation invoque des services de la couche logique métier via son interface. De même la couche logique métier invoque les services de la couche d'accès aux données via son interface. La couche d'accès aux données dispose de deux implémentations de son interface pour persister les données : JDBC et JPA.

Plusieurs avantages découlent de cette architecture :

1. Un service de la couche logique métier peut choisir l'une ou l'autre des implémentations de persistance des données ;
2. Chaque couche peut être modifiée indépendamment des autres en respectant la signature de l'interface ;
3. Les préoccupations sont bien séparées entre les couches.

4.4 Migration vers le système cible

Les fonctionnalités du système cible à implémenter sont les suivantes :

1. les services Web REST :
   — récupération de la liste des APIs publiques gérées par le système
   — récupération de la liste des ressources d'une API Web publique
   — détection des patrons et anti-patrons d'une API Web publique
2. l'application Web
3. l'ajout d'une nouvelle API publique externe au système

Nous avons utilisé une approche itérative et incrémentale pour développer le nouveau système modernisé.

À chaque étape (incrément), une nouvelle version du système modernisé a été déployée et le système patrimonial restant opérationnel.

Voici le plan des activités de transformation sous forme d'incréments :

Incrément 1 : Migrer la couche d'accès aux données

Incrément 2 : Migrer la couche logique métier

Incrément 3 : Migrer la couche présentation

4.5 Migration de la couche d'accès aux données

Dans le système patrimonial, on utilise des fichiers textes pour stocker les traces d'exécution des algorithmes de détection ainsi que des fichiers CSV (*Comma-Separated Values*) pour stocker des statistiques. Pour faciliter l'accès et la recherche d'informations, nous avons utilisé une base de données (BD) relationnelle MySQL. Cette BD stocke les informations relatives aux différentes APIs publiques

qui sont gérées par le système et permet de générer des rapports de comparaison des différentes APIs en terme du nombre de patrons et d'anti-patrons détectés.

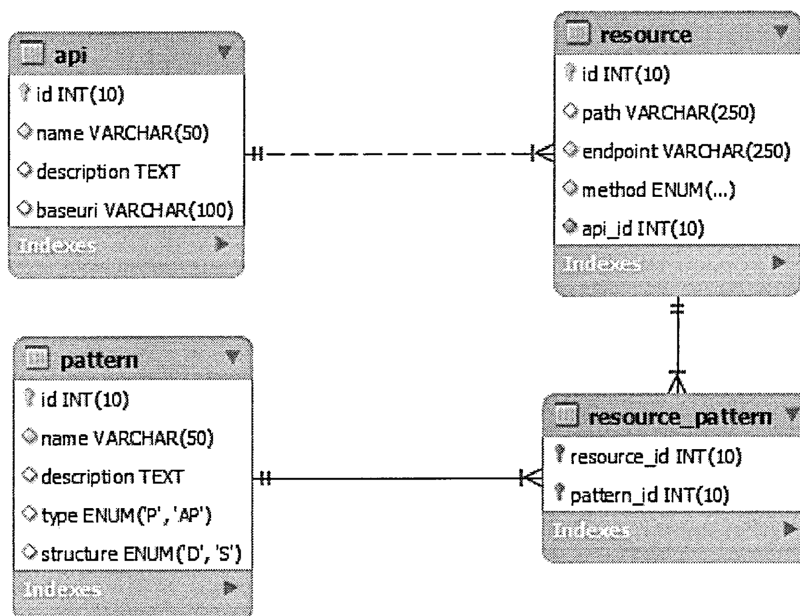
La migration des données consiste à :

- créer un schéma Entité-Relation pour modéliser les données ;
- créer une base de données relationnelle qui correspond au schéma Entité-Relation ;
- créer les objets du domaine ;
- implémenter l'accès aux données.

Le modèle Entité-Relation

La figure 4.10 montre le modèle Entité-Relation pour la base de données.

Figure 4.10 Le modèle Entité-Relation (ER).



La base de données

Pour créer une base de données, il faut l'instancier et ensuite lui ajouter des

éléments (tables, vues, ...). La manipulation d'une base de données est faite en utilisant des commandes SQL.

Les principales commandes SQL sont les suivantes :

- `CREATE DATABASE` pour créer une nouvelle base de données
- `CREATE TABLE` pour créer une nouvelle table
- `ALTER TABLE` pour mettre à jour le schéma d'une table
- `INSERT INTO TABLE` pour insérer des données dans une table
- `UPDATE TABLE` pour mettre à jour les données d'une table

On peut exécuter plusieurs commandes SQL à la fois en les groupant dans un fichier. Par exemple, le fichier SQL suivant permet de créer la base de données `soda-r` ainsi que le schéma de la table `api` :

```
1 CREATE DATABASE 'soda-r';
2 USE 'soda-r';
3 CREATE TABLE 'api' (
4     'id' int(10) NOT NULL AUTO\_INCREMENT,
5     'name' varchar(50) NOT NULL,
6     'description' text,\linebreak
7     'baseuri' varchar(100) DEFAULT NULL,
8     PRIMARY KEY ('id')
9 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Les objets du domaine

Les objets du domaine ou les objets "entité" sont implémentés dans la partie modèle (M) du style architectural MVC et ils servent simplement à stocker les données. Les classes "entité" ont une relation un-à-un avec les tables d'une base de données relationnelle.

La figure A.3 (p. 83) montre l'implémentation de l'objet du domaine `ApiEntity`.

Les objets d'accès aux données

Dans cette couche, nous avons définir une interface commune d'accès aux données et cette interface a été implémentée par JDBC ou JPA (Hibernate). Nous pouvons donc choisir l'une ou l'autre des implémentations pour accéder aux données.

La figure A.4 (p. 84) montre l'interface `ApiDao` d'accès aux données. À travers cette interface, nous pouvons persister les objets du domaine `ApiEnity` et aussi les retrouver à partir de la base de données.

La figure A.5 (p. 85) montre l'implémentation JPA (Hibernate) de l'interface `ApiDao`.

La figure A.6 (p. 86) montre l'implémentation JDBC de l'interface `ApiDao`.

4.6 Migration de la couche logique métier

La migration de la logique métier consiste à :

— implémenter la logique métier du nouveau système ;

— emballer la fonctionnalité de détection des patrons et anti-patrons du système patrimonial ;

— réusiner le code pour caractériser de façon précise les patrons et les anti-patrons associés à chaque ressource lors de l'exécution de l'algorithme de détection.

Logique métier du nouveau système

La figure A.7 (p. 87) montre l'interface de la logique métier de la classe *api* et la figure A.8 (p. 88) montre son implémentation.

Cette couche utilise les objets, préalablement instanciés, de la couche d'accès aux données. Les deux exemples suivants montrent comment une même fonctionnalité

métier peut être réalisée en faisant appel à deux technologies d'accès aux données différentes.

Exemple 1 : Retrouver la liste des APIs en utilisant JPA

```
1 ApiDao apiDao = new ApiDaoJpaImpl();
2
3 ApiBlo apiBlo = new ApiBloImpl();
4 apiBlo.setApiDao(apiDao);
5
6 List<ApiEntity> apis = apiBlo.retrieveAll();
```

Exemple 2 : Retrouver la liste des APIs en utilisant JDBC

```
1 ApiDao apiDao = new ApiDaoJdbcImpl();
2
3 ApiBlo apiBlo = new ApiBloImpl();
4 apiBlo.setApiDao(apiDao);
5
6 List<ApiEntity> apis = apiBlo.retrieveAll();
```

Emballage de la fonctionnalité de détection des patrons et anti-patrons

Le code suivant fait appel au système patrimonial pour la détection des patrons REST et retourne un tableau d'éléments JSON.

```
1 public JSONArray detectPatternsJson(String apiName) throws
     ClassNotFoundException, SQLException {
2         JSONArray apiPatternsArray = new JSONArray();
3
4         RESTPadManager restpadManager = new RESTPadManager();
5         try {
6             for (Motif m : Motif.patterns()) {
7                 restpadManager.addMotifToDetection(m);
8             }
```

```
9               apiPatternsArray = restpadManager.runDynamicDetection
                    (apiName);
10          } catch (Exception e) {
11              e.printStackTrace();
12          }
13
14          return apiPatternsArray;
15  }
```

Réusinage du code pour caractériser de façon précise les patrons et les anti-patrons associés à chaque ressource

Un (anti-)patron peut être caractérisé par plusieurs combinaisons d'informations renvoyées dans les réponses HTTP. Dans le système patrimonial, l'algorithme de détection ne rapportait pas pourquoi un patron ou anti-patron est associé à une ressource. On renvoie la valeur 1 pour indiquer que le patron est associé à une ressource et on renvoie la valeur 0 pour indiquer que le patron n'est pas associé à la ressource.

Nous avons réusiné le code pour caractériser un patron ou un anti-patron de façon précise.

4.7 Migration de la couche présentation : l'API REST

L'implémentation de l'API REST a été faite en deux étapes :

1. Modélisation des ressources REST
2. Implémentation des services Web de l'API REST

4.7.1 Modélisation des ressources

Comme indiqué dans le chapitre «Introduction», la modélisation d'une API REST passe par les étapes suivantes :

1. Identifier les ressources qui vont être exposées comme services Web ;
2. Modéliser les relations entre les ressources ;
3. Définir les URIs qui identifient les ressources ;
4. Comprendre l'association de chaque méthode HTTP avec chaque ressource ;
5. Concevoir les représentations de chaque ressource.

Identification des ressources

Les ressources correspondent aux concepts du domaine que nous avons identifiés à partir des cas d'utilisation au chapitre III. Pour rappel, ces ressources sont les suivantes :

- api
- ressource
- patron
- anti-patron

et les relations entre ces concepts du domaine sont les suivants :

- Une api a plusieurs ressources
- Une ressource peut avoir plusieurs patrons et plusieurs anti-patrons

Les patrons et anti-patrons d'une API correspondent à l'ensemble des patrons et anti-patrons de toutes les ressources de cette API.

Aussi, à chaque *ressource* est associée une ressource de type *collection* qui contient toutes les instances de cette ressource.

Définition des URIs et les relations entre les ressources

Le tableau 4.1 définit les ressources, les relations entre les ressources, les URIs qui identifient les ressources, les méthodes HTTP appliquées à chaque ressource ainsi que la description de chaque ressource (étapes 2 à 4 de la modélisation).

Tableau 4.1 Nomenclature de l'API REST.

RESSOURCE	URI	MÉTHODE	DESCRIPTION
apis	/apis	GET	Obtenir toutes les APIs
		POST	Créer une nouvelle API
api	/apis/{*name*}	GET	Obtenir une API identifiée par son nom
		DELETE	Supprimer une API identifiée par son nom
resources	/apis/{*name*}/resources	GET	Lister toutes les ressources d'une API identifiée par son nom
patterns	/apis/{*name*}/patterns	GET	Détecter et lister tous les patrons d'une API identifiée par son nom
antipatterns	/apis/{*name*}/antipatterns	GET	Détecter et lister tous les anti-patrons d'une API identifiée par son nom
patterns	/patterns	GET	Obtenir tous les patrons
pattern	/patterns/{*pname*}	GET	Obtenir un patron identifié par son nom
antipatterns	/antipatterns	GET	Obtenir tous les anti-patrons
antipattern	/antipatterns/{*apname*}	GET	Obtenir un anti-patron identifié par son nom

Voici quelques exemples de relations entre les ressources :

- La ressource */apis* est la collection de toutes les ressources *api*
- La ressource */apis/{name}* est la ressource api identifiée par le nom *name*
- La ressource */apis/{name}/patterns* est la collection de toutes les ressources *pattern* de la ressource *api* identifiée par le nom *name*

Conception des représentations de chaque ressource

Toutes les ressources de l'API ont la même URL de base qui est la suivante : `http://webrestpad.sofa.uqam.ca`.

Représentation d'une ressource de base : l'API alchemy

URL : `http://webrestpad.sofa.uqam.ca/apis/alchemy`

La figure A.9 (p. 89) montre la représentation de cette ressource.

Représentation d'une collection de ressources : liste des APIs

URL : `http://webrestpad.sofa.uqam.ca/apis`

La figure A.10 (p. 89) montre la représentation de cette ressource.

Représentation d'un processus : détection des patrons de l'API bitly

URL : `http://webrestpad.sofa.uqam.ca/apis/bitly/patterns`

La figure A.11 (p. 90) montre la représentation de cette ressource.

4.7.2 Implémentation des services Web de l'API REST

Configuration de Jersey

Le support de JSON est basé sur la librairie MOXy qui permet de convertir des objets JAVA POJO à JSON et vice-versa.

L'installation de cette librairie consiste à déclarer le module `jersey-media-moxy` comme dépendance dans le fichier `pom.xml` de MAVEN.

```
<dependency>
   <groupId>org.glassfish.jersey.media</groupId>
   <artifactId>jersey-media-moxy</artifactId>
</dependency>
```

Programmation du serveur

La programmation des services Web, associés aux ressources de l'API, utilise des annotations du standard JAX-RS telles qu'implémetées dans le cadriciel Jersey. La figure A.12 (p. 91) montre le code pour la ressource `/apis`.

4.8 Migration de la couche présentation : l'application Web

L'application Web est implémentée comme frontal (*front-end*) pour l'API REST décrite dans la section précédente.

4.9 Ajouter une nouvelle API au système

Pour ajouter une nouvelle API au système, nous avons utilisé un moteur de gabarit nommé *Mustache* qui génére automatiquement le code source lié à cette API pour le fichier composite, l'interface et la classe client.

4.9.1 Mustache

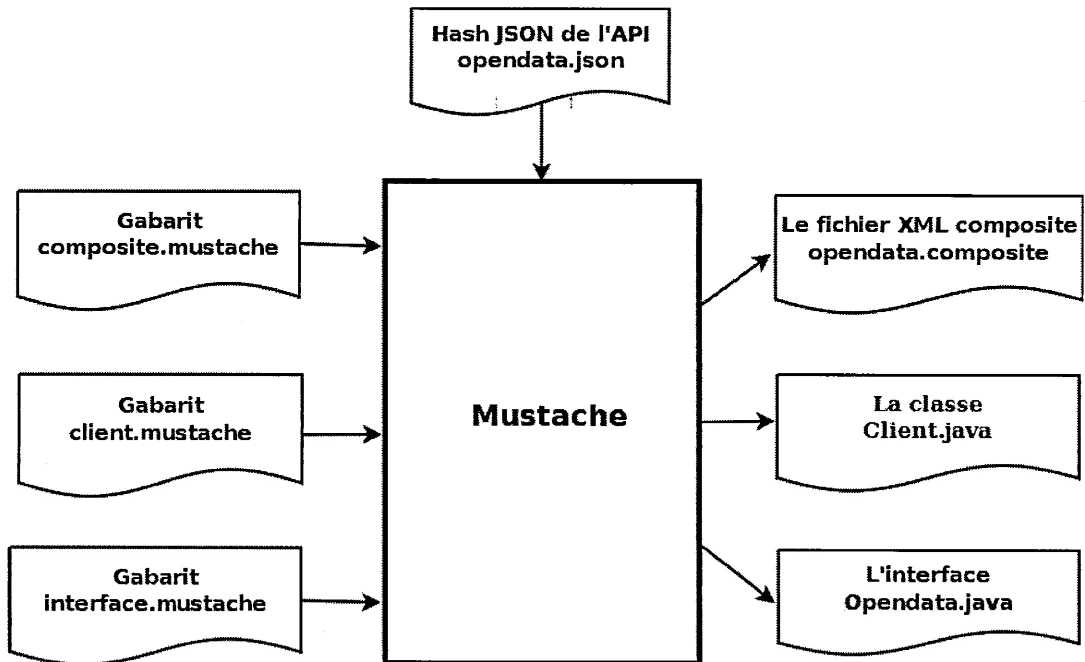
Mustache (Wanstrath, 2009) est un moteur de gabarit basé uniquement sur des balises (*tags*) et il n'y a pas d'instructions conditionnelles (if, else) ni de boucles (while, do while, for). Le fichier gabarit contient alors du texte statique et des balises qui doivent être remplacées par des valeurs fournies dans un fichier appelé *hash*. La figure 4.11 (p. 76) illustre ce mécanisme pour l'API `opendata`. On a alors trois gabarits, combinés au *hash*, pour générer le code source du fichier composite, de la classe client et de l'interface.

Les balises

Les balises correspondent aux éléments variables dans un gabarit et elles sont déclarées entre {{ et }}. Dans le cas du gabarit du fichier composite SCA, par exemple, les éléments variables sont les suivants :

```
{{api_name}}
{{base_uri}}
{{api_token}}
```

Figure 4.11 Ajout d'une nouvelle API (opendata) au système.




```
{{secret_token}}
```

Ce sont les seuls éléments qui changent, d'une API à l'autre, dans le gabarit du fichier composite.

Les sections

Pour simuler les boucles, Mustache utilise la notion de *section* qui commence par un dièse (#) et se termine par une barre oblique (/). Dans le cas du gabarit de l'interface SCA pour l'API, par exemple, on retrouve la section "resources" suivante :

```
{{#resources}}
@GET
{{#resource_path}}@Path("{{resource_path}}"){{/resource_path}}
```

```
@Produces("application/{{format}}")
...
{{/resources}}
```

Dans le hash, la section `resources` correspond à un ensemble d'éléments JSON qui vont être parsés l'un après l'autre. Dans chacun de ces éléments JSON, les balises `resource_path` et `format` vont être remplacées par leurs valeurs correspondantes.

Les commentaires

Pour documenter les gabarits, Mustache utilise le point d'exclamation (!). Par exemple :

```
{{!API Interface template}}
```

est considéré comme un commentaire et va être ignoré.

4.9.2 Les gabarits et le hash

Les trois gabarits sont illustrés respectivement dans les figures A.13 (p. 92), A.14 (p. 93) et A.15 (p. 94). Le hash de l'API opendata, qui correspond à l'échantillon de ressources sélectionnées, est illustré dans la figure A.16 (p. 95).

CONCLUSION

Le but de ce projet consistait à migrer un système patrimonial orienté objets vers un système orienté services Web de type REST. Le système patrimonial était une application Java qui s'exécute sur une machine locale et le système cible s'exécute sur un environnement Web Java EE. La stratégie de migration a consisté à utiliser une technique de modernisation (redéveloppement, rhabillage, ...) au niveau de chaque couche d'abstraction de l'application (présentation, logique métier, accès aux données). La nouvelle application permet aux fournisseurs d'APIs de tester la qualité de leurs APIs en termes de patrons de d'anti-patrons REST.

Le processus de migration s'est fait en deux étapes. La première étape a consisté à comprendre le fonctionnement interne du système patrimonial et la deuxième étape a consisté à définir une architecture cible, à migrer les trois couches d'abstraction de l'application et à automatiser l'ajout d'une nouvelle API su système.

Le projet présente quelques limitations à améliorer dans des travaux futurs, notamment :

- l'ajout d'un mécanisme d'autentification pour pouvoir utiliser les services Web REST ;
- la création du hash et la génération du code d'une nouvelle API à partir de l'interface de l'application Web ;
- la suppression et la mise à jour du code et des entrées dans la base de données d'une API existante ;
- la sauvegarde des traces d'exécution et la génération de rapports de synthèse sur la détection de patrons et anti-patrons d'une API.

La réalisation de ce projet m'a permis d'approfondir mes connaissances dans les techniques de migration des systèmes patrimoniaux et dans la conception et la réalisation des architectures Web multi-niveau et des services Web de type REST. Cependant, il y a eu quelques défis à relever. Il fallait comprendre l'architecture SCA (*Service Component Architecture*), les caractéristiques des patrons et anti-patrons REST et surtout le code existant pour pouvoir le modifier.

ANNEXE A

EXTRAITS DE CODE

Figure A.1 L'interface définissant les méthodes d'utilisation de l'API.

```
1  package ca.uqam.sofa.opendata.api;
2  import javax.ws.rs.GET;
3  import javax.ws.rs.Path;
4  import javax.ws.rs.Produces;
5  import javax.ws.rs.QueryParam;
6  import javax.ws.rs.core.Response;
7
8  public interface Opendata {
9          @GET
10         @Path("energy-usage-2010.json")
11         @Produces("application/json")
12         Response getByBuildingType(
13                 @QueryParam("$$app_token") String app_token,
14                 @QueryParam("building_type") String building_type
15                 );
16         @GET
17         @Path("energy-usage-2010.json")
18         @Produces("application/json")
19         Response getBySubBuildingType(
20                 @QueryParam("$$app_token") String app_token,
21                 @QueryParam("building_type") String building_type
                       ,
22                 @QueryParam("building_subtype") String
                       building_subtype,
23                 @QueryParam("$where") String where,
24                 @QueryParam("$limit") String limit,
25                 @QueryParam("$offset") String offset
26                 );
27         ...
28 }
```

Figure A.2 La classe client qui va accéder aux services Web REST.

```
1  package ca.uqam.sofa.opendata.lib;
2  import org.osoa.sca.annotations.Property;
3  import org.osoa.sca.annotations.Reference;
4  import org.osoa.sca.annotations.Service;
5  import ca.uqam.sofa.opendata.api.Opendata;
6  import com.sofa.metric.lexical.Dictionaries;
7
8  @Service(Runnable.class)
9  public class Client implements Runnable {
10         /** Reference to the Opendata services. */
11         @Reference
12         private Opendata opendata;
13         @Property
14         private String app_token = "Fo2vflcqEPRJNAVY08iEkNZoU";
15         @Property
16         private String base_uri;
17         @Override
18         public void run() {
19                 // set the base URI into dictionaries
20                 Dictionaries.getInstance().setBaseURI(base_uri);
21                 this.callOpendataMethods();
22         }
23         private void callOpendataMethods() {
24                 opendata.getByBuildingType(app_token, "
                      Residential");
25                 opendata.getBySubBuildingType(app_token, "
                      Residential", "Single Family", "
                      kwh_january_2010 > 0", "10", "20");
26                 opendata.getByTimeRegionBus(app_token,
                      "2015-01-12T21:20:55", "29", "20");
27                 opendata.getPropaneLocations(app_token, "LPG");
28         }
29 }
```

Figure A.3 Implémentation de l'objet du domaine `ApiEntity`.

```
1  package models.entities;
2  public class ApiEntity {
3          private Integer id;
4          private String  name;
5          private String  description;
6          private String  baseuri;
7          public ApiEntity() {
8          }
9          // Getters
10         public Integer getId() {
11                 return this.id;
12         }
13         public String getName() {
14                 return this.name;
15         }
16         public String getDescription() {
17                 return this.description;
18         }
19         public String getBaseuri() {
20                 return this.baseuri;
21         }
22         // Setters
23         public void setId(Integer id) {
24                 this.id = id;
25         }
26         public void setName(String name) {
27                 this.name = name;
28         }
29         public void setDescription(String description) {
30                 this.description = description;
31         }
32         public void setBaseuri(String baseuri) {
33                 this.baseuri = baseuri;
34         }
35 }
```

Figure A.4 `ApiDao.java` - Interface de l'objet d'accès aux données.

```
1  package models.dal;
2
3  import java.sql.SQLException;
4  import java.util.List;
5
6  import models.entities.ApiEntity;
7
8  public interface ApiDao {
9          // Create a new api entity
10         public void create(ApiEntity api) throws
               ClassNotFoundException, SQLException;
11
12         // Retrieve an api entity by Id
13         public ApiEntity retrieveById(Integer apiId) throws
               ClassNotFoundException, SQLException;
14
15         // Retrieve all api entities
16         public List<ApiEntity> retrieveAll() throws
               ClassNotFoundException, SQLException;
17
18         // Update an existing api entity
19         public void update(ApiEntity api) throws
               ClassNotFoundException, SQLException;
20
21         // Delete an existing api entity
22         public void delete(Integer apiId) throws
               ClassNotFoundException, SQLException;
23 }
```

Figure A.5 ApiDaoJpaImpl.java - Implémentation JPA de l'interface ApiDao.

```
1  package models.dal; import models.entities.ApiEntity;
2  import org.apache.commons.lang.Validate; import org.hibernate.*;
3  import java.sql.SQLException; import java.util.List;
4  @SuppressWarnings("rawtypes")
5  public class ApiDaoJpaImpl extends BaseHibernateDao implements
      ApiDao {
6          private static final long serialVersionUID = 1L;
7          public void create(ApiEntity api) {
8                  Validate.notNull(api, "Null api not allowed");
9                  this.getSession().save(api);
10         }
11         public ApiEntity retrieveById(Integer apiId) {
12                 Validate.notNull(apiId, "Null apiId not allowed")
                      ;
13                 return (ApiEntity)((BaseHibernateDao) this.
                      getSession()).retrieve(apiId);
14         }
15         @SuppressWarnings("unchecked")
16         public List<ApiEntity> retrieveAll() {
17                 Criteria criteria = this.getSession().
                      createCriteria(ApiEntity.class);
18                 return criteria.list();
19         }
20         public void update(ApiEntity api) {
21                 Validate.notNull(api, "Null api not allowed");
22                 this.getSession().update(api);
23         }
24         public void delete(Integer apiId) {
25                 Validate.notNull(apiId, "Null apiId not allowed")
                      ;
26                 ApiEntity api = this.retrieveById(apiId);
27                 if(api != null) { this.getSession().delete(api);
                      }
28         }
29 }
```

Figure A.6 `ApiDaoJdbcImpl.java` - Implémentation JDBC de l'interface Apidao.

```
1  package models.dal;
2  import models.entities.ApiEntity;
3  import java.sql.PreparedStatement;
4  import java.sql.ResultSet;
5  import java.sql.SQLException;
6  import java.sql.Statement;
7  import java.util.ArrayList;
8  import java.util.List;
9
10 public class ApiDaoJdbcImpl implements ApiDao {
11         // Create a new api entity
12         public void create(ApiEntity api) throws
              ClassNotFoundException, SQLException
13         {
14                 String sql = "INSERT INTO api (id, name,
                      description, baseuri) VALUES (?, ?, ?, ?)";
15                 PreparedStatement stmt = (new Jdbc()).Connect().
                      prepareStatement(sql);
16                 stmt.setInt(1, api.getId());
17                 stmt.setString(2, api.getName());
18                 stmt.setString(3, api.getDescription());
19                 stmt.setString(4, api.getBaseuri());
20                 stmt.executeUpdate();
21         }
22
23         // Retrieve an api entity by name
24         public ApiEntity retrieveByName(String apiName) throws
              ClassNotFoundException, SQLException
25         {
26                 ...
27         }
28         ...
29 }
```

Figure A.7 `ApiBlo.java` - Interface de la couche logique métier.

```
1  package models.bll;
2  import java.sql.SQLException;
3  import java.util.List;
4  import models.dal.ApiDao;
5  import models.entities.ApiEntity;
6
7  public interface ApiBlo {
8
9          // Create a new api entity
10         public void create(ApiEntity api) throws
               ClassNotFoundException, SQLException;
11
12         // Retrieve an api entity by Id
13         public ApiEntity retrieveById(Integer apiId) throws
               ClassNotFoundException, SQLException;
14
15         // Retrieve all api entities
16         public List<ApiEntity> retrieveAll() throws
               ClassNotFoundException, SQLException;
17
18         // Update an existing api entity
19         public void update(ApiEntity api) throws
               ClassNotFoundException, SQLException;
20
21         // Delete an existing api entity
22         public void delete(Integer apiId) throws
               ClassNotFoundException, SQLException;
23
24         // Get api DAO
25         public ApiDao getApiDao();
26
27         // Set api DAO
28         public void setApiDao(ApiDao apiDao);
29 }
```

Figure A.8 `ApiBloImpl.java` - Implémentation de la couche logique métier.

```
1  package models.bll;
2  import java.sql.SQLException;
3  import java.util.List;
4  import models.dal.ApiDao;
5  import models.entities.ApiEntity;
6
7  public class ApiBloImpl implements ApiBlo {
8          protected ApiDao apiDao;
9
10         public void create(ApiEntity api) {
11                 this.apiDao.create(api);
12         }
13         public void update(ApiEntity api) {
14                 this.apiDao.update(api);
15         }
16         public ApiEntity retrieveById(Integer apiId) {
17                 return this.apiDao.retrieveById(apiId);
18         }
19         public List<ApiEntity> retrieveAll() {
20                 return this.apiDao.retrieveAll();
21         }
22         public void delete(Integer apiId) {
23                 this.apiDao.delete(apiId);
24         }
25         public ApiDao getApiDao() {
26                 return this.apiDao;
27         }
28         public void setApiDao(ApiDao apiDao) {
29                 this.apiDao = apiDao;
30         }
31 }
```

Figure A.9 Représentation d'une ressource de base : l'API `alchemy`.

```
1 {
2         "id": 1,
3         "name": "alchemy",
4         "description": null,
5         "baseuri": "http://access.alchemyapi.com"
6 }
```

Figure A.10 Représentation d'une collection de ressources : liste des APIs.

```
1  [
2  {
3          "id": 1,
4          "name": "alchemy",
5          "description": null,
6          "baseuri": "http://access.alchemyapi.com"
7  },
8  {
9          "id": 2,
10         "name": "bestbuy",
11         "description": null,
12         "baseuri": "https://api.remix.bestbuy.com/v1"
13 },
14 {
15         "id": 3,
16         "name": "bitly",
17         "description": null,
18         "baseuri": "https://api-ssl.bitly.com/v3"
19 },
20 ...
```

Figure A.11 Représentation d'un processus : détection des patrons de l'API bitly.

```
1  {
2        "endpoint": "/v3/link/encoders_count?access_token=
            b2ced19b1b9215918af1adfbd74f1bc686bb0d58&link=http%3A/
            bit.ly/ze6poY&format=json",
3        "trace": "...",
4        "method": "GET",
5        "patterns":
6        [
7          {
8            "name": "Contextualised Resource Names",
9            "information": "Context chain : \nContext detected
                between [link] and [encode, rs, by, count]\n1.0
                context relations were detected out of 1.0",
10           "type": "P",
11           "detections": "1"
12         },
13         {
14           "name": "Verbless URI",
15           "information": "No CRUDy word detected : ",
16           "type": "P",
17           "detections": "1"
18         }
19       ]
20 },
21 {
22       "endpoint": "/v3/shorten?access_token=
            b2ced19b1b9215918af1adfbd74f1bc686bb0d58aaa&longUrl=http
            %3A%2F%2Fsofa.uqam.ca&format=json",
23       "trace": "...",
24       "method": "GET",
25       "patterns": "..."
26 },
27 ...
```

Figure A.12 Code pour récupérer la ressource /apis : Liste des APIs.

```
1  @Path("/apis")
2  public class ApiResource {
3      /**
4       * Method handling HTTP GET requests. The returned object
          will be
5       * sent to the client as "application/json" media type.
6       * @return JSONArray that will be returned as a application/
          json response.
7       * @throws IOException
8       * @throws JsonMappingException
9       * @throws JsonGenerationException
10      */
11
12     @GET
13     @Produces(MediaType.APPLICATION_JSON)
14     public JSONArray retrieveAllApisJson() throws
         ClassNotFoundException, SQLException,
         JsonGenerationException, JsonMappingException, IOException
          {
15         ApiBlo apiBlo = new ApiBloImpl();
16         ApiDao apiDao = new ApiDaoJpaImpl();
17         apiBlo.setApiDao(apiDao);
18
19         List<ApiEntity> apis = apiBlo.retrieveAll();
20
21         JSONArray apisArray = new JSONArray();
22         for (ApiEntity api : apis) {
23                 ObjectMapper mapper = new ObjectMapper();
24             String json = mapper.writeValueAsString(api);
25             apisArray.put(json);
26         }
27
28         return apisArray;
29     }
```

Figure A.13 Le gabarit Mustache du fichier composite FraSCAti.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <sca:composite xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
3      xmlns:frascati="http://frascati.ow2.org/xmlns/sca/1.1" name
          ="{{api_name}}">
4      <sca:service name="r" promote="client/Runnable" />
5      <sca:component name="client"  >
6          <sca:implementation.java class="ca.uqam.sofa.{{api_name
              }}.lib.Client"/>
7          <sca:reference name="{{api_name}}" promote="client/{{
              api_name}}">
8              <sca:interface.java interface="ca.uqam.sofa.{{
                  api_name}}.api.I{{api_name}}"/>
9              <frascati:binding.rest uri="{{base_uri}}"/>
10         </sca:reference>
11         <sca:property name="base_uri">{{base_uri}}</sca:property>
12         <sca:property name="app_token">{{api_token}}</sca:
              property>
13         <sca:property name="secret_token">{{secret_token}}</sca:
              property>
14     </sca:component>
15 </sca:composite>
```

Figure A.14 Le gabarit Mustache pour la création de l'interface associée à une API.

```
1  {{!API Interface template}}
2
3  package ca.uqam.sofa.{{api_name}}.api;
4
5  import javax.ws.rs.GET;
6  import javax.ws.rs.Path;
7  import javax.ws.rs.PathParam;
8  import javax.ws.rs.Produces;
9  import javax.ws.rs.QueryParam;
10 import javax.ws.rs.core.Response;
11
12 public interface I{{api_name}} {
13
14 {{#resources}}
15     @GET
16     {{#resource_path}}@Path("{{resource_path}}"){{/resource_path
           }}
17     @Produces("application/{{format}}")
18     Response getByBuildingType(
19             {{#query_params}}
20                 @QueryParam("{{param_name}}") String {{var_name
                       }},
21             {{/query_params}}
22             @QueryParam("$$app_token") String app_token
23             );
24 {{/resources}}
25
26 }
```

Figure A.15 Le gabarit Mustache pour la création du client qui implémente l'interface associée à l'API.

```
1  package ca.uqam.sofa.{{api_name}}.lib;
2  import org.osoa.sca.annotations.Property;
3  import org.osoa.sca.annotations.Reference;
4  import org.osoa.sca.annotations.Service;
5  import ca.uqam.sofa.{{api_name}}.api.I{{api_name}};
6  {{#api_auth}}import ca.uqam.sofa.{{api_name}}.util.Authentication
       ;{{/api_auth}}
7  import com.sofa.metric.lexical.Dictionaries;
8
9  @Service(Runnable.class)
10 public class Client implements Runnable {
11         /** Reference to the I{{api_name}} services. */
12         @Reference
13         private I{{api_name}} {{api_name}};
14         @Property
15         private String app_token;
16         @Property
17         private String base_uri;
18         @Override
19         public void run() {
20                 // set the base URI into dictionaries
21                 Dictionaries.getInstance().setBaseURI(base_uri);
22                 this.callI{{api_name}}Methods();
23         }
24         private void callI{{api_name}}Methods() {
25     {{#resources}}
26       {{api_name}}.{{resource_method}}F{{resource_desc}}({{#
            query_params}}{{var_value}}, {{/query_params}}app_token)
            ;
27     {{/resources}}
28         }
29 }
```

Figure A.16 Le Hash qui spécifie l'échantillon de ressources pour l'ensemble de ressources energy-usage-2010.

```
1  {
2    "api_name": "opendata",
3    "app_token": "Fo2vflcqEPRJNAVY08iEkNZoU",
4    "base_uri": "https://data.cityofchicago.org/resource",
5    "resources": [
6      {
7        "resource_path": "energy-usage-2010.json",
8        "format": "json",
9        "query_params": [
10         { "param_name": "building_type", "var_name": "
              building_type", "var_value": "Residential" },
11       ]
12     },
13     {
14       "resource_path": "energy-usage-2010.json",
15       "format": "json",
16       "query_params": [
17         { "param_name": "building_type", "var_name": "
              building_type", "var_value": "Residential" },
18         { "param_name": "building_subtype", "var_name": "
              building_subtype", "var_value": "Single Family" },
19         { "param_name": "$where", "var_name": "where", "var_value
              ": "kwh_january_2010 > 0" },
20         { "param_name": "$limit", "var_name": "limit", "var_value
              ": "10" },
21         { "param_name": "$offset", "var_name": "offset", "
              var_value": "20" }
22       ]
23     }
24 }
```

BIBLIOGRAPHIE

Bennet, K. (1995). Legacy systems : Coping with success. *IEEE Software*, *12*(1), 19–23.

Bisbal, J., Lawless, D., Wu, B. et Grimson, J. (1999). Legacy information systems migration : A brief review of problems, solutions, and research issues. *IEEE Software*, *6*(5).

Brodie, M. L. et Stonebraker, M. (1995). *Migrating legacy systems : gateways, interfaces and the incremental approach*. Morgan Kaufmann Publishers.

Chappell, D. (2007). INTRODUCING SCA. `http://www.davidchappell.com/writing/Introducing_SCA.pdf`. [Récupéré le 26 avril 2017].

Chikofsky, E. J. et Cross, J. H. (1990). Reverse engineering and design recovery : A taxonomy. *IEEE Software*, *7*(1), 13–17.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures.* (Thèse de doctorat). University of California, Irvine.

Kelly, M. (2016). JSON Hypertext Application Language. `https://tools.ietf.org/html/draft-kelly-json-hal-08`. [Récupéré le 18 mai 2017].

Palma, F., Dubois, J., Moha, N. et Guéhéneuc, Y.-G. (2014). Detection of rest patterns and antipatterns : A heuristics-based approach. *X. Franch, A. Ghose, G. Lewis et S. Bhiri, éditeurs, Service-Oriented Computing. Springer Berlin Heidelberg*, *2*(8831 de Lecture Notes in Computer Science), 230–244.

Pautasso, C. (2009). Some REST Design Patterns (and Anti-Patterns). `http://www.jopera.org/node/442`. [Récupéré le 26 avril 2017].

Pautasso, C. et Wilde, E. (2009). *SOA with REST.* WWW.

Seacord, R. C., Plakosh, D. et Lewis, G. A. (2003). *Modernizing Legacy Systems.* Addion-Wesley.

Tilkov, S. (2008). REST Anti-Patterns. `https://www.infoq.com/articles/rest-anti-patterns`. [Récupéré le 26 avril 2017].

Wanstrath, C. (2009). mustache - Logic-less templates. https://mustache.github.io/mustache.5.html. [Récupéré le 26 avril 2017].