Surveillance des interfaces logicielles

Robin Jarry

le cnam
eicnam

Conservatoire National des Arts et Métiers

Département Informatique : Architecture et Intégration des Systèmes Logiciels

Mémoire d'ingénieur

Robin Jarry

Surveillance des interfaces logicielles

Date de soutenance	12 février 2013
Directeurs de mémoire	M. Pierre Courtieu M. Ivan Boule
Jury	M. Yann Pollet (président) M. Pierre Courtieu M. Ivan Boule M. Jérôme Vacher M. Nicolas Lethellier

Table des matières

Index des illustrations

Index des tables

Résumé – Abstract

L'intégration d'applications multi-composants est une discipline complexe et très coûteuse en temps et en ressources humaines. L'instabilité des interfaces des composants logiciels est au centre du problème. Aujourd'hui, la surveillance de l'évolution de ces interfaces et l'évaluation du risque impliqué par celle-ci se fait manuellement, opération à faible valeur ajoutée et sujette aux erreurs et aux oublis. Ce document présente une solution technique permettant d'automatiser la détection d'évolutions dans les interfaces de programmation et l'évaluation du risque.

§

The integration of multi-component applications is a complex and expensive task. The instability of the application programming interfaces (*API*) of software components is directly responsible for this complexity. For now, all *API* evolutions and their impact on the application stability can only be detected manually. This operation is therefore error prone and has a low added value. In this document, we will present a solution that allows to automate the detection of *API* modifications and the evaluation of their potential impact on the application stability.

Remerciements

Ce mémoire est dédié à ***Jérôme Vacher*** de la société *Thales Global Services*. Sans lui, ce projet – et bien d'autres que j'ai menés durant ces dernières années – n'aurait pas vu le jour.

Je tiens à remercier mes directeurs de mémoire : ***Pierre Courtieu*** et ***Ivan Boule*** professeurs au *Conservatoire National des Arts et Métiers* ; et mes relecteurs : ***Nicolas Lethellier*** ainsi que ***Guillaume Brocard***, ***Alexandre Sanches***, ***Jean-Baptiste Garidel*** et ***Alexia Even*** pour leurs précieux conseils et leurs encouragements.

Je souhaite également exprimer ma sincère gratitude aux sociétés *Thales Global Services* et *ABlogiX* pour m'avoir accordé leur confiance et m'avoir soutenu dans la réalisation de l'application *APIWatch*.

Enfin, une pensée émue va à ma grand-mère qui m'a offert – il y a bien longtemps – *mon premier ordinateur*.

Introduction

L'essor des communautés *open-source* et la facilité de diffusion de leur production via internet permettent aujourd'hui de mettre à disposition un nombre inégalé et toujours croissant de composants logiciels réutilisables, à une vitesse de production elle aussi croissante. De ce fait, la complexité de la production de logiciels réside de plus en plus dans l'assemblage de ces fragments. Cette logique modulaire, évolutive et très souple donne accès à une grande richesse de fonctionnalités à des coûts de développement réduits.

Cette richesse présente cependant un revers : la bonne orchestration de tous ces fragments nécessite de s'assurer que leurs interactions se font de manière cohérente. Indépendants dans leur production, les développeurs/intégrateurs sont ainsi interdépendants du fait de l'évolution libre et non contrôlée des composants. Le risque attaché à ces changements est à l'heure actuelle évalué de façon empirique ou constaté *a posteriori*. La multiplicité des sources et la rapidité de l'évolution des composants nécessitent aujourd'hui une veille technique constante pour pouvoir garantir la stabilité et la fiabilité des logiciels.

Afin de préserver la richesse de fonctionnalités et la rapidité de développement, tout en maintenant la fiabilité des produits, nous nous attacherons à présenter une solution pour automatiser l'évaluation du risque attaché à l'évolution des composants.

La première partie présentera et explicitera les différentes notions clés de l'architecture à base de composants, puis la seconde partie abordera les enjeux de ce type d'architecture au travers des pratiques et contraintes qu'elle amène. Enfin, dans une troisième partie, nous développerons une proposition de solution facilement extensible et utilisable dans la plupart des environnements pour permettre la surveillance et le contrôle des évolutions d'interfaces logicielles.

1. Notions de base

1.1. Architecture logicielle à base de composants

L'ingénierie logicielle à base de composants met l'accent sur la séparation des rôles dans un système logiciel donné. C'est une approche fondée sur la réutilisation de modules indépendants à couplage lâche : les composants.

Un composant logiciel est une unité logique destinée à être réutilisée en tant que « pièce détachée » dans des applications. Il fournit le plus souvent un ensemble de services que l'on peut invoquer par le biais d'interfaces définies. Un des bénéfices fondamentaux de cette approche modulaire est que l'on peut substituer un composant par un autre du moment qu'il respecte les mêmes interfaces.

Dans l'Illustration 1, on peut voir un exemple d'application de gestion de commandes pour un site de vente en ligne. Une telle application doit rendre de nombreux services : stockage de l'historique des commandes, gestion de la comptabilité, envoi d'*e-mails* aux clients, etc. Il serait difficile de la concevoir de manière monolithique. Pour simplifier son développement et ses évolutions, on préférera une approche modulaire.

Illustration 1: Exemple d'architecture à composants, application de gestion de commandes

Les différentes fonctionnalités de l'application sont gérées par des composants spécialisés comme montré dans la Table 1.

Composant	*Rôle / Service*
`OrderManagement`	Création, édition et annulation de commandes client.
`Accounting`	Gestion comptable, édition de factures, etc.
`PersistanceLayer`	Stockage des données de l'application.
`Notifications`	Notifications au client et aux équipes techniques.
`UserInterface`	Interface avec l'utilisateur.

Table 1: Rôle des différents composants de l'application

Comme dit plus haut, on peut substituer un composant par un autre respectant les mêmes interfaces. On pourra par exemple changer le composant `PersistanceLayer` pour qu'il stocke les données dans une base relationnelle ou directement dans un système de fichiers. Et ce, sans modifier le comportement global de l'application. L'autre avantage de cette approche est que – une fois les rôles correctement définis – chaque composant peut être développé par des équipes différentes.[1]

1.2. Interface de programmation

Pour exploiter les services proposés par un composant, ses congénères doivent « entrer en communication » avec celui-ci. Il leur est donc nécessaire de définir au préalable un « protocole » et de le respecter. En ingénierie logicielle, ce « protocole » est appelé « interface de programmation ». Dans la suite de ce document, on appellera interface de programmation par son terme anglais, plus communément utilisé : *Application Programming Interface* ou *API*.

Le terme « *API* » peut représenter deux choses :

- Par sa forme « instanciée » : une *API* est le « contrat » exposé par un composant logiciel dans lequel est stipulé la façon d'invoquer les services de celui-ci.
- Par sa forme « conceptuelle » : des « informations d'*API* » portent des données relatives à une *API*.

Une *API* inclut généralement des spécifications de routines ou de fonctions, de structures de données, de classes d'objets ou de variables que les composants doivent respecter, utiliser et/ou implémenter. Les formats des fichiers écrits et/ou lus, les protocoles de communication réseau, plus généralement tout

1 Dans son article *Component Based Software Engineering* [1], Wilhelm Hasselbring donne une description plus détaillée sur les architectures logicielles à base de composants.

échange de données entre composants à travers un support font également partie de l'*API*.

Reprenons l'exemple précédent ; le composant de gestion des commandes propose trois services comme indiqué dans la Table 2.

Service	***Description***
`createOrder`	Création d'une commande.
`modifyOrder`	Édition d'une commande.
`cancelOrder`	Annulation d'une commande.

Table 2: Services proposés par le composant `OrderManagement`

Pour fonctionner, le composant `OrderManagement` exploite des services proposés par d'autres composants comme décrit dans la Table 3.

Service	***Proposé par***	***Description***
`getBilling`	`Accounting`	Édition de la facture d'une commande.
`persistOrder`	`PersistanceLayer`	Stockage de l'état d'une commande.

Table 3: Services exploités par le composant `OrderManagement`

Sur l'Illustration 2, on peut voir un schéma au format *UML*[2] représentant l'*API* du composant `OrderManagement`.

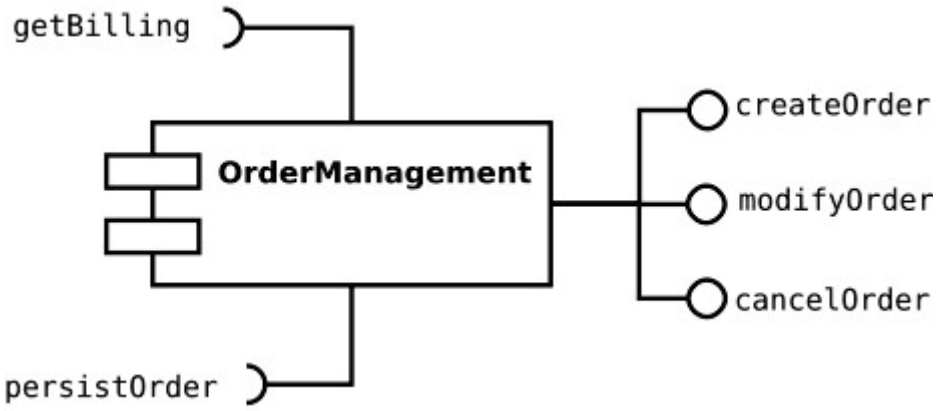


Illustration 2: API d'un composant logiciel : `OrderManagement`

2 *Unified Modeling Language* ou *UML* est un langage de modélisation utilisé dans l'ingénierie logicielle pour concevoir et décrire des applications.

1.3. Dépendances

Dès qu'un composant *A* exploite un service fourni par un composant *B*, on dit que *A* possède une dépendance directe vers *B*. Quand un composant dépend d'un autre, il hérite des dépendances de celui-ci. On parle alors de dépendances transitives.

Sur l'Illustration 3, on peut voir les dépendances entre les composants `OrderManagement`, `UserInterface` et `PersistanceLayer` vus dans l'exemple précédent.

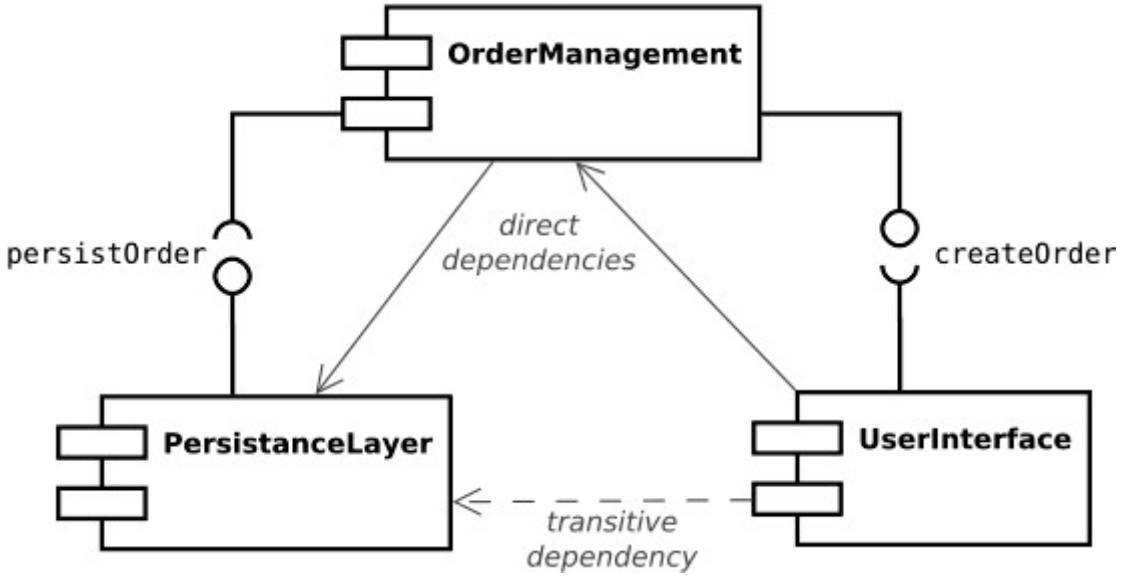


Illustration 3: Dépendances entre composants

2. Contexte et enjeux

2.1. Pratiques et contraintes du développement logiciel

Le génie logiciel respecte la même logique de conception et de livraison par étapes que la plupart des autres domaines d'ingénierie. Ces étapes sont décrites dans l'Illustration 4.

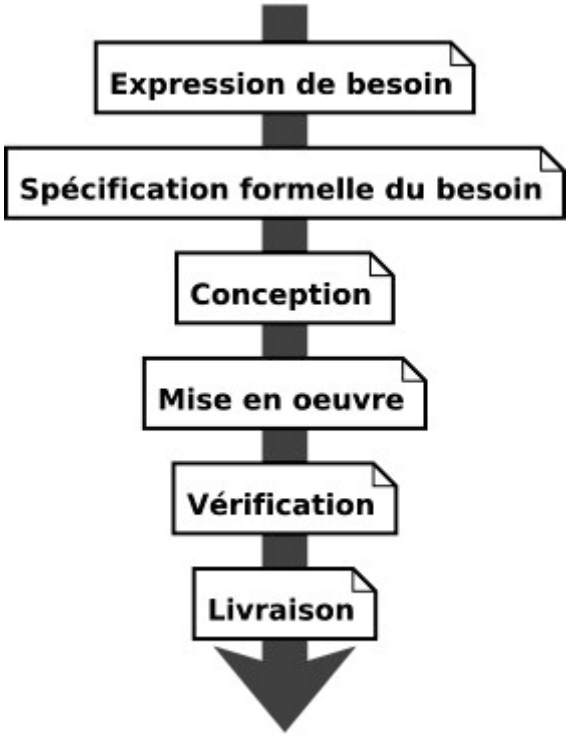


Illustration 4: Étapes de la construction d'un logiciel

Dans un monde idéal, le logiciel livré au client ne contiendrait aucun *bug* et répondrait parfaitement au besoin exprimé. Il n'y aurait pas besoin de changer un comportement ou de corriger un problème : il n'y aurait pas de retour en arrière dans la succession des étapes décrites dans l'Illustration 4, et il n'y aurait qu'une seule livraison dudit logiciel.

Néanmoins, le produit livré n'est jamais exempt de défauts ; les besoins du client et les contraintes d'exploitations peuvent également amener à devoir

modifier le logiciel. Il est donc inévitable de repasser par les étapes décrites ci-dessus (voir Illustration 5), et donc, de livrer plusieurs fois un même logiciel.

Illustration 5: Cycles dans les étapes de la construction d'un logiciel

Dans tous les autres domaines d'ingénierie (mécanique, bâtiment, etc.), la livraison d'une révision du produit remplace généralement l'ancienne. Les anciennes révisions d'un logiciel, elles, ne sont pas rendues caduques par la livraison de nouvelles.

Comme il a été dit au chapitre 1.2, les composants communiquent entre eux. Le bon déroulement de ces communications repose sur une entente mutuelle au sujet d'une même *API*. Or, les composants évoluent, chaque évolution pouvant changer leur comportement mais surtout leur *API*. Au chapitre 1.3, on a également vu que certains composants dépendent des services fournis par d'autres. On doit pouvoir exprimer cette dépendance en prenant en compte l'évolutivité des *API*. Malheureusement, les données d'*API* sont beaucoup trop complexes pour être exploitables pour spécifier une dépendance. Il faut trouver une notion plus facile à manipuler.

On commencera généralement par identifier clairement et de manière unique chaque livraison d'un même composant. Cet identifiant sera appelé « version », l'évolution de cette version permettant de rendre compte de l'évolution de l'*API*.

Il existe de nombreuses manières[3] de noter la version des logiciels. Une des plus courantes consiste à utiliser une suite de nombres ordonnés du plus significatif au moins significatif. À chaque nouvelle version, on incrémente un

3 Le procédé de « versionnage » (de l'anglais *versioning*) consiste à assigner un identifiant à chaque état d'une pièce logicielle http://en.wikipedia.org/wiki/Software_versioning

ou plusieurs des nombres en fonction de l'importance des modifications apportées au logiciel. L'Illustration 6 montre un exemple d'une version à trois nombres.

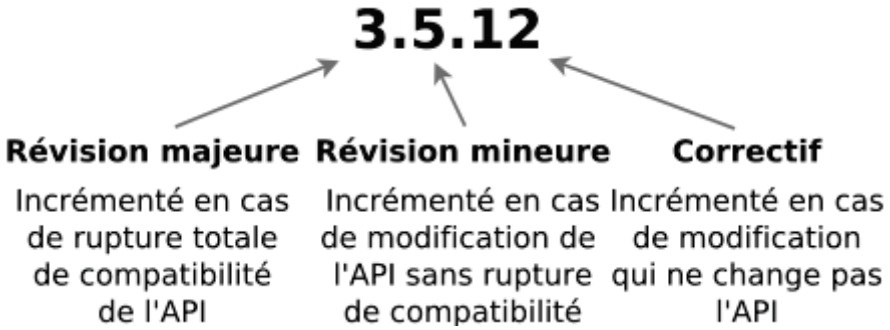


Illustration 6: Signification du numéro de version d'un logiciel

Grâce à cette notion de version, on pourra identifier les composants logiciels en prenant en compte la dimension temporelle et son influence sur l'état de leur *API*. Ceci permettra également d'exprimer les dépendances de manière plus précise.

2.2. Modification d'une API : risques et conséquences

Un changement d'*API* peut survenir à plusieurs niveaux :

- Changement de format d'un fichier (entrée ou sortie)
- Changement du type d'une variable
- Modification de la spécification d'une structure de données
- Changement du nombre ou des types de paramètres nécessaires à l'invocation d'une fonction.
- etc.

Si lors de l'exécution, un composant fait appel à un service d'un autre sans prendre en compte une modification de l'*API*, il est très difficile de prévoir le comportement du programme. Celui-ci va dans la plupart des cas s'arrêter de fonctionner ou pire, continuer à fonctionner de manière erratique.

On a vu précédemment que la version d'un composant logiciel est supposée refléter les évolutions de son *API*. Or, la modification de la version est une opération manuelle. Il est donc de la responsabilité des équipes de développement de faire évoluer la version en corrélation avec la nature des changements apportés au composant. Comme toute opération manuelle, celle-ci est sujette aux erreurs et aux oublis. Il n'existe pas de moyen simple d'assurer que l'état de l'*API* d'un composant logiciel soit reflété par sa version.

2.3. Solutions existantes

La plupart du temps, on a recours à des tests d'intégration[4] pour vérifier le bon fonctionnement de tous les composants d'un même système avant de le livrer.

Néanmoins, l'écriture et la maintenance de ces tests est une tâche très coûteuse en temps et ressources humaines, à faible valeur ajoutée et qui permet difficilement d'assurer une couverture totale des *API*. Aussi, l'exécution de ces tests doit souvent être supervisée par un être humain, l'interprétation des résultats pouvant être très complexe. En cas de développement de composants réutilisables comme des bibliothèques de fonctions ou des systèmes d'exploitation, il est impossible de faire des tests d'intégration car les autres modules susceptibles d'exploiter les *API* fournies ne peuvent être connus par avance.

Il existe cependant des solutions qui permettent de contrôler la stabilité et la compatibilité ascendante de composants logiciels sans utiliser de tests d'intégration. Ces solutions se présentent souvent comme des outils en ligne de commande facilement automatisables :

2.3.1. *ABI Compliance Checker*

ABI Compliance Checker est un outil de contrôle de compatibilité ascendante au niveau binaire et au niveau source pour les bibliothèques *C* et *C++*. L'outil compare les fichiers d'en-tête et les fichiers binaires des versions anciennes avec ceux des nouvelles et analyse les changements dans les *API* et *ABI*[5] susceptibles de rompre la compatibilité : changements de pile d'appel, symboles supprimés, champs renommés, etc.

Une description complète de l'outil est donnée sur leur site : http://ispras.linuxbase.org/index.php/ABI_compliance_checker

4 Un test d'intégration consiste à vérifier que tous les composants d'un même système fonctionnent de manière cohérente. On effectue les tests d'intégration après que chaque composant ait été testé unitairement.

5 Une *Application Binary Interface* (interface binaire-programme), décrit une interface bas niveau entre les applications et le système d'exploitation, entre une application et une bibliothèque ou bien entre différentes parties d'une application. Une *ABI* diffère d'une *API*, qui elle, définit une interface entre du code source et une bibliothèque.

2.3.2. *PkgDiff*

Package Changes Analyzer (pkgdiff) est un outil permettant d'analyser les changements dans les logiciels *Linux*. L'outil est destiné aux mainteneurs de *Linux* qui doivent assurer la compatibilité des anciennes et nouvelles versions des paquets (aux formats *RPM*, *DEB*, *TAR.GZ*, etc.).

L'outil est disponible sur la forge logicielle *GitHub* : http://pkgdiff.github.com/pkgdiff/

§

Ces solutions ont été conçues pour des technologies ou langages de programmation bien spécifiques (*C/C++*, *Linux*, etc.). De plus, chacun d'entre eux est développé sur la base d'une technologie qui lui est propre (langage, *framework*, plate-forme, etc.). De ce fait, il est techniquement difficile de mutualiser les fonctionnalités de chacun. Étendre le fonctionnement de ces outils à d'autres langages serait très complexe car ceux-ci n'ont pas été conçus dans l'optique d'être évolutifs.

A ce jour, il n'existe pas de solution qui permette une gestion unifiée des contrôles de cohérence d'*API* qui soit indépendante du langage de programmation analysé.

Pour pallier ce besoin, il faudrait disposer d'un outil facilement extensible à de nouveaux langages. Dans cette optique, cet outil devrait fournir une abstraction totale du langage de programmation pour ne conserver que les informations d'*API*. Cette abstraction permettrait notamment de mutualiser les processus de traitement des ces informations.

Dans le chapitre suivant, on décrira la conception et la mise en œuvre d'une solution technique à ce problème.

3. Proposition de solution

La solution qui sera décrite par la suite doit permettre la surveillance et le contrôle des évolutions d'interfaces logicielles. Étant donné qu'il existe un nombre indéterminé de langages de programmation et de plate-formes d'exécution, la solution proposée se doit d'être facilement extensible et utilisable dans la plupart des environnements.

Pour répondre à ce problème, j'ai imaginé, conçu et développé une application qui permet de maîtriser les évolutions des interfaces logicielles : *APIWatch*.

Illustration 7 : Logo de l'application APIWatch

APIWatch est une application ***libre*** distribuée sous la licence *BSD*[6] à deux clauses. C'est une licence logicielle simple et peu contraignante développée à l'*Université de Californie à Berkeley* (UCB). Les seules restrictions qu'elle impose aux utilisateurs sont les suivantes :

- Quiconque est autorisé à modifier, réutiliser tout ou partie de l'application pour produire des travaux dérivés sous réserve d'en citer la provenance. Il est impossible de prétendre avoir écrit/créé l'application si ce n'est pas le cas. Le droit d'auteur est inaliénable. Toute redistribution de l'application – sous quelque forme que ce soit – doit comporter la licence originale.
- Le ou les auteurs ne peuvent pas être tenus responsables si l'application ne fonctionne pas de la manière attendue, en cas de dommages et/ou perte de données.

§

On peut consulter le code source, lire la documentation ou télécharger une version compilée de l'application à l'adresse suivante : http://www.apiwatch.org/.

6 *Berkeley Software Distribution* http://www.linfo.org/bsdlicense.html

3.1. Spécifications

APIWatch est une application dont le but est de surveiller les évolutions des interfaces d'un composant logiciel. Elle doit permettre une détection automatisée des modifications d'API. Ceci afin de faciliter les prises de décision que ces dernières impliquent :

- Ajustement du numéro de version
- Modification du *changelog*[7]
- Annulation des modifications
- etc.

3.1.1. Analyse

APIWatch devra pouvoir extraire les informations essentielles d'*API* depuis un code source donné.

- L'application devra être capable de supporter la plupart des paradigmes de programmation[8].
- Les informations d'*API* extraites devront être modélisées de manière formelle.
  - Ce modèle devra être indépendant du langage et du paradigme de programmation.
  - Ce modèle unifié pourra contenir des informations provenant d'un ou plusieurs fichiers de code source.
  - Ce modèle unifié devra être *sérialisable*[9] dans les formats textuels les plus communs : *XML*, *JSON*, etc.

7 Un *changelog* (littéralement « Journal des modifications ») est un document qui permet de lister les changements significatifs apportés à une version d'un composant logiciel.

8 Il existe quatre principaux paradigmes de programmation informatique. La programmation *impérative*, *fonctionnelle*, *orientée-objet*, et *logique*. La plupart des langages utilisent une combinaison de ces paradigmes. http://en.wikipedia.org/wiki/Programming_paradigm

9 Néologisme emprunté au mot anglais *serializable* relatif à la *serialization*. Action qui consiste à transformer une structure de donnée en mémoire vers un format qui peut être stocké sur un support ou transmis par un protocole de communication réseau.

3.1.2. Persistance

APIWatch devra conserver une trace de ces informations d'*API*.

- Les informations stockées devront être organisées et classées par composant et version de composant.
- L'utilisateur devra être en mesure de visualiser de manière informelle l'état de l'*API* d'une version d'un composant.

3.1.3. Surveillance

APIWatch devra être en mesure de détecter des modifications de l'*API* d'un composant logiciel.

- Les modifications devront être classées en fonction de leur importance.
- L'utilisateur devra être en mesure de spécifier en détail l'importance de chaque type de modification pour un projet/composant donné.

3.2. Conception et modélisation

3.2.1. Interface logicielle

Le travail de Terrence Parr dans son livre *Language Implementation Patterns* [2] a servi de fondations pour modéliser ce qu'est une *API* de manière formelle. Il était important que le modèle soit aussi « générique » que possible. Ceci afin qu'il s'adapte à tous les paradigmes de programmation.

Le modèle est constitué d'un ensemble de classes simples :

Visibility

C'est un type énuméré permettant de représenter de manière générique la portée ou visibilité d'un élément d'*API*. On retrouvera les différentes valeurs du type énuméré dans la Table 4.

Valeur	*Description*
`PRIVATE`	Visibilité restreinte à la classe dans laquelle l'élément est défini (uniquement applicable pour les langages objet).
`SCOPE`	Visibilité restreinte au contexte courant (fichier, classe, module, foncteur, fonction, procédure, etc.).
`PROTECTED`	Visibilité restreinte aux sous-classes de celle où l'élément est défini (uniquement applicable pour les langages objet).
`PUBLIC`	Visibilité maximale.

Table 4: Visibilité des éléments d'API

La plupart des langages de programmation n'utiliseront pas toutes les valeurs du type énuméré. Par exemple, le langage C n'a que deux niveaux de visibilité possibles : `SCOPE` pour une variable ou fonction déclarée dans un fichier `.c`, et `PUBLIC` pour une variable ou fonction déclarée dans un fichier `.h`.

La visibilité d'un élément sera cruciale pour déterminer l'importance d'une modification d'*API*. Plus l'élément affecté par la modification est visible, plus le risque engendré par celle-ci est élevé.

APIElement

C'est l'élément de base du modèle. C'est une classe abstraite dont toutes les classes du modèle vont hériter. On retrouvera une description de ses attributs et méthodes dans les tables 5 et 6 :

Attribut	*Type*	*Description*
`name`	**`String`**	Nom de l'élément.
`language`	**`String`**	Langage dans lequel l'élément a été défini. Un champ libre est utilisé ici plutôt qu'un type énuméré pour permettre une extensibilité des langages supportés par l'application.
`sourceFile`	**`String`**	Chemin du fichier dans lequel l'élément est défini. Il pourra éventuellement contenir une indication de numéro de ligne dans le fichier en respectant la syntaxe : `<chemin>:<n° de ligne>`
`visibility`	**`Visibility`**	Visibilité de l'élément. Voir paragraphe précédent.
`parent`	**`APIElement`**	Référence vers l'élément parent de celui-ci.

Table 5: Attributs de la classe `APIElement`

Méthode	*Type de retour*	*Description*
`ident()`	**`String`**	Renvoie un identifiant unique pour cet élément. L'implémentation par défaut renvoie le type effectif et le nom de l'élément : `<type>:<nom>`
`path()`	**`String`**	Renvoie un « chemin » permettant de retrouver l'élément dans la structure d'une *API*. Cette méthode sera utilisée par l'algorithme de comparaison. Elle utilise l'attribut `parent` pour calculer le chemin.

Table 6: Méthodes de la classe `APIElement`

Symbol

Un symbole est la brique principale d'une *API*. Cela peut être une variable, une fonction ou même une définition de type (simple ou complexe). La classe `Symbol` est abstraite et hérite de la classe `APIElement`. Ses attributs propres sont décrits dans la Table 7.

Attribut	*Type*	*Description*
`modifiers`	**`Set<String>`**	Ensemble de mots clés qui permettent de qualifier ou modifier le comportement d'un symbole. Ils sont dépendants du langage source. Par exemple en Java, le mot clé `static` permet de définir une variable ou méthode attachée directement à une classe et non pas aux instances de cette classe.

Table 7: Attributs de la classe `Symbol`

Variable

Une variable est un espace de stockage pour une valeur auquel est attaché un nom. La plupart du temps cette variable permet de stocker des valeurs d'un certain type. Dans certains langages (comme C) on peut définir des variables avec des contraintes. Par exemple, on peut définir une structure de données avec des champs suffixés par une taille en bits[10] :

```
typedef struct DISK_REGISTER {
     unsigned track         :9; /* valeurs (0, 511) */
     unsigned sector        :5; /* valeurs (0, 31)  */
     unsigned write_protect :1; /* valeurs (0, 1)   */
} disk_register_t;
```

Ici, les champs sont déclarés comme des entiers non-signés mais leur valeur est limitée à leur taille en bits.

On utilisera des objets de type `Variable` pour représenter les variables, les attributs de classe et les arguments de fonction. La classe `Variable` hérite de `Symbol`. Ses attributs propres sont décrits dans la Table 8.

Attribut	*Type*	*Description*
`type`	**`String`**	Nom du type de la variable tel qu'il est défini dans le code source.
`constraints`	**`String`**	Optionnel. Cet attribut est un champ libre car sa valeur est trop dépendante du langage de programmation et donc trop incertaine.

Table 8: Attributs de la classe `Variable`

10 http://publications.gbdirect.co.uk/c_book/chapter6/bitfields.html

Function

On utilisera la classe `Function` pour représenter à la fois les procédures, les fonctions et les méthodes de classe. Une fonction possède un type de retour, et une liste d'arguments. `Function` hérite également de `Symbol`. On retrouvera une description de ses attributs et méthodes dans les tables 9 et 10, respectivement.

Attribut	*Type*	*Description*
`returnType`	**`String`**	Nom du type de retour de la fonction tel qu'il est défini dans le code source.
`arguments`	**`List<Variable>`**	Arguments de la fonction. Peut être vide.
`hasVarArgs`	**`boolean`**	Vrai si la fonction accepte un nombre variable d'arguments. Dans la plupart des langages, on symbolise cela par une ellipse « ... » à la fin des arguments.
`exceptions`	**`Set<String>`**	Dans certains langages, en plus du type de retour, on peut déclarer des types d'exceptions qui peuvent être émises par une fonction. Celles-ci ne changent pas toujours l'*API* de manière significative mais peuvent avoir un impact tout de même.

Table 9: Attributs de la classe `Function`

Méthode	*Type de retour*	*Description*
`signature()`	**`String`**	Dans certains langages (comme C++ et Java), il peut y avoir plusieurs méthodes d'une classe qui ont le même nom. Ce procédé est appelé « surcharge ». Pour les différencier, on procédera comme la plupart des compilateurs[11] en suffixant les noms de fonction avec les types de leurs arguments.
`ident()`	**`String`**	Cette méthode est redéfinie ici en utilisant la valeur de la méthode `signature()` à la place de l'attribut `name`.

Table 10: Méthodes de la classe `Function`

11 http://en.wikipedia.org/wiki/Name_mangling

Type

Dans la plupart des langages de programmation il est possible d'attribuer arbitrairement un type aux variables pour – entre autres – déterminer la nature des données qu'elle peuvent contenir et la manière dont elle sont enregistrées et traitées par le système. Concrètement le type d'un élément influe sur la taille que le compilateur ou l'interpréteur lui allouera en mémoire.

La classe `Type` hérite de `Symbol` et est également abstraite. Elle n'a pas d'attributs ni méthodes propres.

SimpleType

Un type simple sera en règle générale un type scalaire du langage de programmation (booléen, nombre entier, caractère, nombre à virgule flottante, etc.) ou un alias vers un autre type pré-existant[12].

La classe `SimpleType` hérite de `Type` et n'en est qu'une version concrète. Elle n'a pas d'attributs ni méthodes propres.

ArrayType

Un tableau contient une série d'éléments d'un certain type. Dans la plupart des langages, ce tableau est de taille fixe. Cette taille peut être déclarée en même temps que la variable (allocation statique) ou à l'allocation mémoire, si la variable est un pointeur vers un tableau (allocation dynamique).

La classe `ArrayType` hérite de `Type` et possède un attribut supplémentaire : le type des éléments contenus dans le tableau. Cet attribut est décrit dans la Table 11.

Attribut	*Type*	*Description*
`elementType`	**`String`**	Nom du type des éléments du tableau tel qu'il est défini dans le code source.

Table 11: Attributs de la classe `ArrayType`

12 Dans certains langages on peut définir des alias vers d'autres types pour rendre le programme plus lisible. Par exemple en C, `typedef long ADDRESS;` permet de déclarer des variables de type `ADDRESS` qui seront interprétées comme `long` par le compilateur. Voir http://en.wikipedia.org/wiki/Typedef

ComplexType

Les types de données complexes sont des types composés de plusieurs types plus élémentaires et qui possèdent une architecture spécifique autorisant des traitements dédiés à leur type.

On utilisera des objets de type `ComplexType` pour représenter des structures de données ou des classes (pour les langages orientés objet). Un `ComplexType` peut contenir un ensemble de symboles : des variables, des fonctions ou même d'autres types internes.

`ComplexType` hérite de la classe `Type` et possède deux attributs supplémentaires décrits dans la Table 12.

Attribut	*Type*	*Description*
`symbols`	**`Set<Symbol>`**	Ensemble de symboles définis à l'intérieur du type complexe.
`superTypes`	**`Set<String>`**	Ensemble de noms de types dont le type complexe hérite.

Table 12 : Attributs de la classe `ComplexType`

APIScope

En règle générale, pour faciliter la maintenance et l'utilisation d'un composant, on place les éléments d'*API* dans des espaces de noms « nommés ». Ceci permet notamment d'éviter les conflits de noms. En *Java* et *Ada*, on utilise des *packages*, en *C#* et *C++* : des *namespaces*, etc.

La classe `APIScope` reflète ce concept. Elle hérite directement de `APIElement` car ce n'est pas un symbole à proprement parler. En revanche, elle contient des symboles et éventuellement un ensemble d'`APIScope`. Ses attributs et méthodes sont détaillés dans les tables 13 et 14, respectivement.

Attribut	*Type*	*Description*
`dependencies`	**`Set<String>`**	Ensemble de symboles dont l'`APIScope` dépend. Cet attribut représente les instructions du type `import` ou `uses` dans le code source.
`symbols`	**`Set<Symbol>`**	Symboles définis dans cet `APIScope`.
`subScopes`	**`Set<APIScope>`**	`APIScope`s contenus à l'intérieur de celui-ci.

Table 13 : Attributs de la classe `APIScope`

Méthode	*Type de retour*	*Description*
`update(APIScope)`	–	Met à jour l'`APIScope` avec le contenu d'un autre passé en argument.

Table 14: Méthodes de la classe `APIScope`

N'importe quelle *API* sera toujours représentée par un `APIScope` « racine » qui n'a pas de nom. Pour les langages qui ne supportent pas les espaces de noms « nommés » (comme *C*) une *API* sera modélisée uniquement par cet `APIScope` « racine ».

L'Illustration 8 montre une vue d'ensemble des modèles précédents organisés les uns par rapport aux autres.

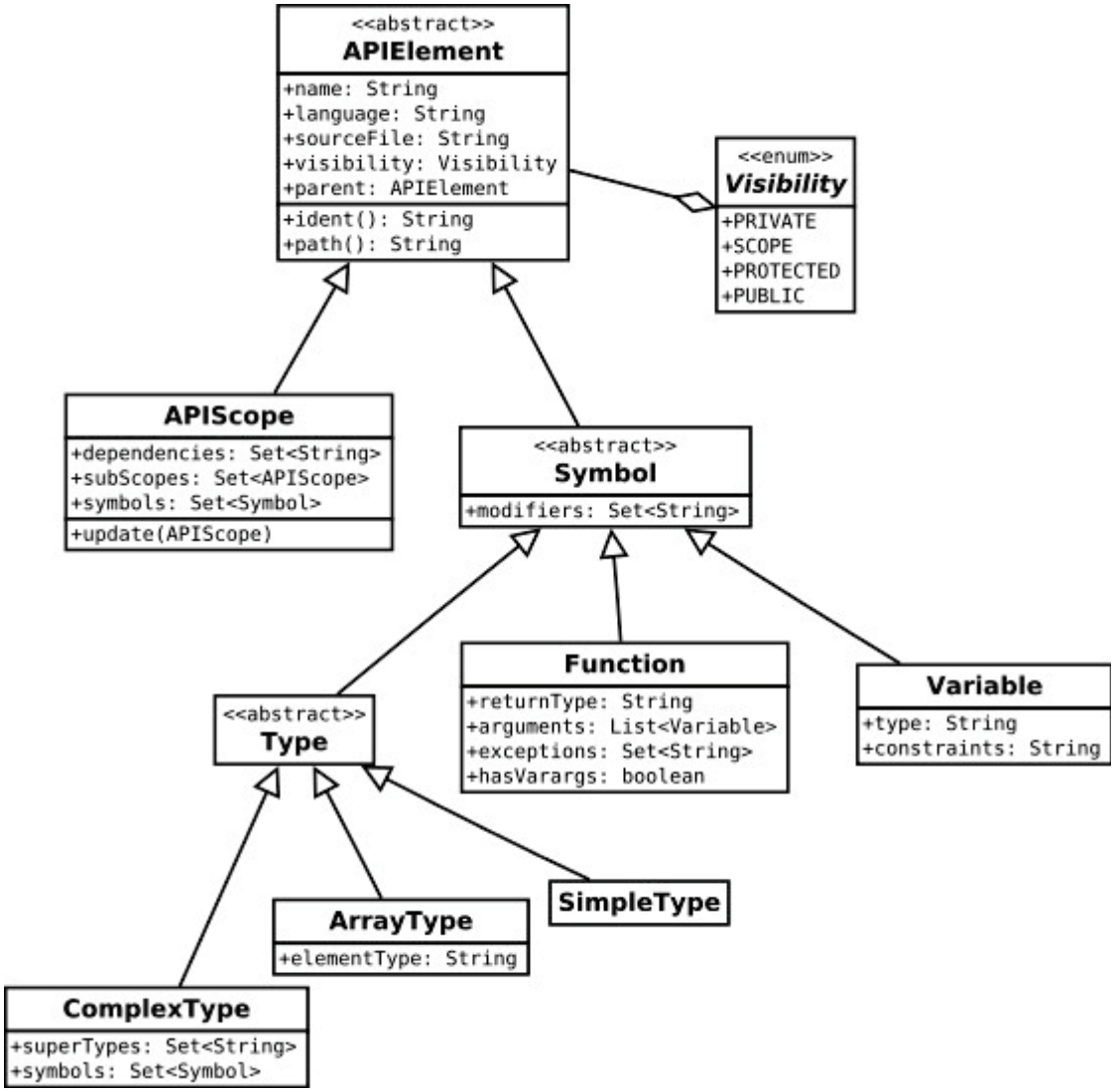


Illustration 8: Diagramme de classes, modélisation d'une API

3.2.2. Modifications d'interface

Maintenant qu'une *API* peut être modélisée, il faut définir comment modéliser des modifications de celle-ci. On partira du principe qu'une modification est détectable entre deux versions (*A* et *B*) des informations d'*API* d'un même composant.

ChangeType

Le type énuméré `ChangeType` sert à qualifier le type d'une modification d'*API* afin de pouvoir associer un niveau de risque à celle-ci. Les différentes valeurs du type énuméré sont détaillées dans la Table 15.

Valeur	***Description***
`REMOVED`	L'élément a été supprimé de l'*API*.
`ADDED`	L'élément a été ajouté à l'*API*.
`CHANGED`	Une des propriétés de l'élément a été modifiée.

Table 15: Types de modification d'API

APIDifference

Les modifications seront considérées de manière unitaire. Si plusieurs changements sont détectés sur un même élément d'*API*, ils donneront lieu à plusieurs « modifications ».

La classe qui représente une modification d'*API* est `APIDifference`. Elle est caractérisée par un type de changement (`ChangeType`), une référence vers le ou les `APIElement`s concernés et dans le cas d'une modification de type `CHANGED`, l'attribut concerné par cette modification. Pour des raisons pratiques et de performance, les valeurs de l'attribut modifié seront stockées dans les objets `APIDifference`.

Attribut	*Type*	*Description*
`changeType`	**`ChangeType`**	Type de modification. Voir paragraphe précédent.
`elementA`	**`APIElement`**	Élément de la version *A* concerné par la modification. Non-défini si il s'agit d'un ajout.
`elementB`	**`APIElement`**	Élément de la version *B* concerné par la modification. Non-défini si il s'agit d'une suppression.
`attribute`	**`String`**	Nom de l'attribut concerné par cette modification. Uniquement défini pour les modifications de type `CHANGED`.
`valueA`	**`Object`**	Valeur de l'attribut pour la version *A*. Uniquement défini pour les modifications de type `CHANGED`.
`valueB`	**`Object`**	Valeur de l'attribut pour la version *B*. Uniquement défini pour les modifications de type `CHANGED`.

Table 16: Attributs de la classe `APIDifference`

Sur l'Illustration 9, on trouvera un diagramme de classes de `ChangeType` et `APIDifference`.

Illustration 9: Diagramme de classes, modélisation d'une modification d'API

3.2.3. Stabilité d'API – Gestion du risque

La notion d'`APIDifference` est trop technique pour pouvoir en tirer des conclusions et évaluer le risque impliqué par celle-ci. Il va falloir trouver une notion dérivée plus exploitable dans la gestion du risque : la « stabilité » d'une *API*. Le principe de base est que « quand une *API* ne change pas, le risque est nul ». Le but va ensuite être d'évaluer le niveau de risque attaché à chaque modification détectée.

Le problème sera traité de la même manière que les outils de contrôle de qualité de code source[13] tels que *lint*, *CheckStyle*, *PMD*, *Logiscope* ou *Klocwork*. Ces outils proposent un mode d'opération assez similaire : ils définissent un ensemble de règles de codage (configurables ou non) qui sont évaluées sur une représentation du code source analysé. Chaque violation de ces règles est associée à un niveau de risque[14].

Pour clarifier l'intérêt et le fonctionnement de ces outils d'analyse de code, voici un exemple pratique : L'outil *CheckStyle*[15] possède un ensemble de règles de style de code (configurables pour la plupart). L'une d'entre elles vérifie si le développeur n'utilise pas de nombres magiques[16]. Un nombre magique est une constante numérique qui diminue souvent la lisibilité du code.

Si on considère le code *Java* suivant :

```
if (response.getCode() > 400) {
    throw new HttpError(response);
}
```

Ici, le nombre `400` a une signification assez floue. Il serait préférable de le remplacer par une constante nommée. En évaluant ce code, *CheckStyle* va produire une violation de ce type :

```
<violation line="87" column="41" severity="error"
           message="400 should be defined as a constant."
           source="com.checkstyle.checks.coding.MagicNumberCheck" />
```

13 http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis
14 Souvent appelé « severity level » dans la plupart des outils d'analyse statique de code.
15 http://checkstyle.sourceforge.net/
16 http://en.wikipedia.org/wiki/Magic_number_(programming)

Dans *APIWatch*, les classes mises en œuvre pour la gestion de stabilité d'*API* sont les suivantes :

Severity

Le type énuméré `Severity` sert à qualifier le niveau de risque engendré par une modification d'*API*. Les différentes valeurs du type énuméré sont détaillées dans la Table 17.

Valeur	*Description*
`INFO`	Niveau de risque nul. La modification n'entraîne aucune rupture de compatibilité.
`MINOR`	Niveau de risque mineur. La modification porte sur des parties réduites et ne risque pas d'entraîner de rupture de compatibilité. Elle mérite néanmoins d'être contrôlée.
`MAJOR`	Niveau de risque majeur. La modification porte sur des parties sensibles et a des chances non négligeables d'entraîner une rupture de compatibilité. Elle doit impérativement être contrôlée.
`CRITICAL`	Niveau de risque critique. La modification a de fortes chances d'entraîner une rupture de compatibilité ascendante.
`BLOCKER`	Niveau de risque bloquant. La modification entraîne une rupture complète de compatibilité ascendante.

Table 17: Niveaux de risque d'une modification d'API

APIStabilityViolation

Quand une modification est évaluée par une règle de stabilité d'*API* (voir plus loin), une violation peut être émise. Les attributs de la classe `APIStabilityViolation` sont détaillés dans la Table 18.

Attribut	*Type*	*Description*
`difference`	**`APIDifference`**	Modification d'*API* qui a donné lieu à cette violation. Voir 3.2.2.
`rule`	**`APIStabilityRule`**	Règle qui a donné lieu à cette violation. Voir paragraphe suivant : *APIStabilityRule*.
`severity`	**`Severity`**	Niveau de risque de cette violation.
`message`	**`String`**	Message portant plus de détails sur la violation et son contexte.

Table 18: Attributs de la classe `APIStabilityViolation`

Sur l'Illustration 10, on trouvera un diagramme de classes de `Severity` et `APIStabilityViolation`.

Illustration 10: Diagramme de classes, violation de règle de stabilité d'API

APIStabilityRule

Cette classe abstraite (ou interface) sera utilisée pour représenter la notion de règle de stabilité d'*API*. Les attributs de la classe `APIStabilityRule` sont détaillés dans la Table 19.

Méthode	*Type de retour*	*Description*
`id()`	**`String`**	Identifiant unique de la règle.
`name()`	**`String`**	Nom de la règle
`description()`	**`String`**	Description de la règle
`configure(Map<String, String>)`	**`-`**	Configuration du comportement de la règle. Par exemple, quel est le niveau de risque des violations émises dans certains cas. Les détails sont laissés à chaque implémentation.
`isApplicable(APIDifference)`	**`boolean`**	Renvoie `Vrai` si la règle est applicable à une modification d'*API* donnée. Cette méthode permet de décider si il est nécessaire d'évaluer les violations provoquées par une modification d'*API*.
`evaluate(APIDifference)`	**`APIStabilityViolation`**	Évalue une modification d'*API* et renvoie (ou non) une violation de stabilité d'*API*. Le risque associé à cette violation peut être configuré selon les implémentations des règles.

Table 19: Méthodes de l'interface `APIStabilityRule`

On retrouvera un diagramme de la classe de l'interface `APIStabilityRule` sur l'Illustration 11.

Illustration 11: Diagramme de classes, règle de stabilité d'API

Plusieurs implémentations par défaut de l'interface `APIStabilityRule` seront proposées dans *APIWatch* :

Classe	*Description*
`ElementRemoval`	Suppression d'un élément d'*API*.
`ElementAddition`	Nouvel élément d'*API*.
`ReducedVisibility`	Réduction du niveau de visibilité d'un élément d'*API*.
`DependenciesChange`	Modification des dépendances d'un `APIScope`. Uniquement applicable aux éléments de type `APIScope`.
`FunctionTypeChange`	Modification du type de retour d'une fonction. Uniquement applicable aux éléments de type `Function`.
`ModifiersChange`	Changement des « modifers » d'un symbole. Uniquement applicable aux éléments de type `Symbol`.
`SuperTypesChange`	Changement des « super-classes » d'une classe ou interface. Uniquement applicable aux éléments de type `ComplexType`.
`VariableTypeChange`	Modification du type d'une variable. Uniquement applicable aux éléments de type `Variable`.

Table 20: Implémentations par défaut de l'interface `APIStabilityRule`

Toutes ces règles sont configurables par l'utilisateur. Un détail des propriétés ajustables est donné en annexe A.

Il doit être possible de fournir des implémentations supplémentaires de l'interface `APIStabilityRule` (voir paragraphe 3.2.4).

3.2.4. Points d'extension

Afin de rendre *APIWatch* modulaire et facilement extensible, on mettra en place un mécanisme de *plug-ins* pour les rôles suivants :

Analyse de langages spécifiques

Chaque langage de programmation a une syntaxe particulière. Comme il n'est pas possible d'écrire un analyseur « générique », une interface a été créée : `LanguageAnalyser` qui permet de laisser les détails d'implémentation séparés du cœur de l'application.

Les méthodes de cette interface sont détaillées dans la Table 21.

Méthode	*Type de retour*	*Description*
`language()`	**`String`**	Nom du langage supporté
`fileExtensions()`	**`String[]`**	Liste des extensions de fichiers utilisés par le langage. Exemple, pour un analyseur *C++*, cette méthode doit renvoyer `["cpp", "hpp"]`. La valeur retournée sera utilisée pour choisir quel analyseur convient à un fichier donné.
`analyse(File)`	**`APIScope`**	Principale méthode à implémenter. Chaque analyseur a une liberté complète quant à la façon d'analyser l'*API* d'un fichier.

Table 21 : Méthodes de l'interface `LanguageAnalyser`

Pour permettre à *APIWatch* de supporter un nouveau langage, il suffit d'implémenter cette interface et de déclarer cette implémentation par un moyen qui sera décrit dans le paragraphe 3.3.2.

Formats de sérialisation

Les éléments du modèle de données doivent pouvoir être sérialisés dans le but de les afficher à l'utilisateur et/ou les stocker. Afin de ne pas limiter les formats de sérialisation et pour faciliter l'interopérabilité de *APIWatch* avec des systèmes existants, deux interfaces sont disponibles :

APIScopeSerializer

On trouvera dans la Table 22 un détail des méthodes de l'interface.

Méthode	*Type de retour*	*Description*
`format()`	**`String`**	Permet de récupérer l'identifiant – supposé unique – du format de sérialisation.
`dump(APIScope, Writer)`	**–**	*Sérialise* une instance de `APIScope` dans un flux de texte passé en paramètre.
`load(Reader)`	**`APIScope`**	Reconstitue une instance de `APIScope` depuis un flux de texte passé en paramètre.

Table 22: Méthodes de l'interface `APIScopeSerializer`

APIStabilityViolationSerializer

On trouvera dans la Table 23 un détail des méthodes de l'interface.

Méthode	*Type de retour*	*Description*
`format()`	**`String`**	Identifiant du format de *sérialisation*.
`dump( List<APIStability Violation>,Writer)`	**–**	Sérialise une liste de `APIStabilityViolation`s dans un flux de texte passé en paramètre.
`load(Reader)`	**`List<APIStability Violation>`**	Reconstitue une liste de `APIStabilityViolations` depuis un flux de texte passé en paramètre.

Table 23: Méthodes de l'interface `APIStabilityRuleSerializer`

Pour permettre à *APIWatch* de supporter un nouveau format de sérialisation, il suffit d'implémenter ces interfaces et de déclarer les implémentations par un moyen qui sera décrit dans le paragraphe 3.3.2.

Règles de stabilité d'API

L'interface `APIStabilityRule` vue au paragraphe 3.2.3 est également un point d'extension.

3.2.5. Principaux services

Analyse

Le cœur du système *APIWatch* se base sur l'analyse de code source pour en extraire des données d'*API*. Les composants de plus haut niveau (internes ou non à *APIWatch*) feront appel au service `Analyser`. Ce service contient des références vers toutes les implémentations disponibles du service `LanguageAnalyser` indexées par leurs extensions de fichier supportées.

Le service `Analyser` expose une unique fonction `analyse()` qui prend un ensemble de fichiers en paramètre. Pour chaque fichier, il va rechercher un analyseur supportant son extension et lui déléguer l'analyse de celui-ci. Le résultat (`APIScope`) de cette analyse va être stocké temporairement. Enfin, tous ces résultats intermédiaires vont être fusionnés en un seul, représentant l'*API* du code source analysé.

On trouvera sur l'Illustration 12 un diagramme de flux représentant le mécanisme d'analyse utilisé.

Illustration 12 : Diagramme de flux, analyse de code source

Comparaison de données d'API

Pour pouvoir détecter des violations aux règles de stabilité d'*API*, il faut calculer les différences entre deux versions d'un même composant.

Le service `DifferenceCalculator` proposera une fonction `getDiffs()` prenant en paramètre deux objets du type `APIScope` et renvoyant une liste d'objets `APIDifference`. Les différences seront recherchées récursivement dans l'ensemble des objets contenus par les deux `APIScope`s[17].

Un diagramme de flux décrivant le mécanisme de calcul des différences d'*API* est donné sur l'Illustration 13.

Illustration 13: Diagramme de flux, calcul des différences entre deux versions d'API d'un même composant

Détection de violations de stabilité d'API

Une fois les différences entre deux versions d'une même *API* calculées, il faut évaluer ces différences à l'aide de règles de stabilité d'*API*.

Le service `ViolationsCalculator` doit en premier lieu être configuré avec un ensemble de ces règles. Puis, on peut appeler la méthode `getViolations()` avec pour paramètre, les différences calculées. Chaque différence va être évaluée par chaque règle qui va émettre ou non une violation de stabilité d'*API*. Enfin, l'ensemble des violations détectées seront retournées par le service.

17 Voir paragraphe 3.3.4 pour une description détaillée de l'algorithme de comparaison utilisé par *APIWatch*.

L'Illustration 14 montre un diagramme de flux représentant le mécanisme de détection de violations de stabilité d'*API*.

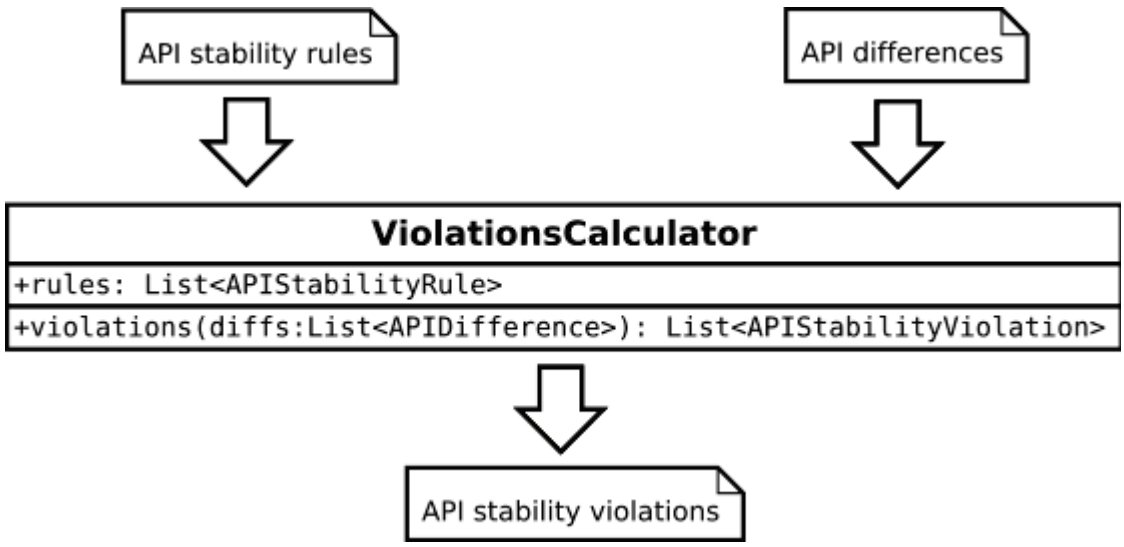


Illustration 14: Diagramme de flux, détection des violations de stabilité d'API

3.2.6. Interface avec l'utilisateur

APIWatch se présentera sous la forme d'un outil en ligne de commande et d'une interface *web*.

Ligne de commande

L'application *APIWatch* est destinée en premier lieu à être utilisée dans le cadre d'une gestion de projet logiciel en intégration continue[18]. Pour rendre *APIWatch* facilement automatisable, le plus simple est de fournir une interface en ligne de commande.

L'interface en ligne de commande doit permettre les opérations suivantes :

- Extraction des données d'*API* d'un ensemble de fichiers et/ou répertoires (de manière récursive). *Sérialisation* de ces données dans un format choisi par l'utilisateur. Envoi des données dans un fichier ou dans l'interface *web* de *APIWatch* en renseignant un nom et une version de composant.

18 L'intégration continue est une discipline du génie logiciel consistant à automatiser le processus d'intégration (compilation, assemblage, analyse de code et tests) d'une application. Ceci afin de permettre aux équipes de développement d'intégrer leur travail le plus fréquemment possible (parfois plusieurs fois par jour) et par conséquent de détecter les problèmes le plus tôt possible. Martin Fowler explique la discipline d'intégration continue en détail dans un article *Continuous Integration* [3].

- Calcul des violations de stabilité d'*API* depuis deux versions analysées auparavant. Les données utilisées en entrée doivent pouvoir provenir de fichiers et/ou de l'interface *web* de *APIWatch*. Les règles de stabilité utilisées pour cette opération doivent être configurables par l'utilisateur. Les violations peuvent ensuite être exportées dans un format choisi par l'utilisateur (texte brut, *XML*, *JSON*, etc.).

Application « web »

Afin de pouvoir garder une trace des données d'*API* des différentes versions des composants logiciels, il faut un support de stockage. Le moyen le plus adapté est d'utiliser une base de données relationnelle (un système de fichiers pourrait convenir mais serait plus difficile à mettre en œuvre). Les moyens d'accès à une base de données sont multiples et manquent de standardisation. De plus, pour l'utilisateur final, l'accès direct à une base de données n'est pas convivial. La technologie *HTTP* a été choisie pour masquer cette complexité et réaliser une interface utilisateur simple et compatible avec tous les navigateurs *web*.

L'interface *web* de *APIWatch* sera donc couplée à une base de données. Elle doit fournir les fonctionnalités suivantes :

- Stockage des données d'*API* dans la base de données. Les données doivent être organisées par composant et version de composant. Le résultat d'analyse produit depuis l'interface en ligne de commande doit pouvoir être « poussé » dans la base de données via des requêtes *HTTP*.
- Accès aux données d'*API* stockées dans la base de données. Dans une page au format *HTML* pour consultation dans un navigateur web et également au format brut (i.e. *XML*, *JSON*, etc.) pour utilisation depuis l'interface en ligne de commande.

3.2.7. Découpage logique

Pour des raisons de modularité et pour faciliter la réutilisation, le découpage en composants suivant a été choisi :

Core

Comme son nom l'indique, ce composant est le « cœur » de *APIWatch*. Il contient tout le modèle de données décrit aux paragraphes 3.2.1, 3.2.2 et 3.2.3. Il expose les points d'extension définis au paragraphe 3.2.4. Enfin, il contient toute la logique nécessaire aux services décrits au paragraphe 3.2.5.

XXX-Analyser

Ce composant est la première brique modulaire de l'application. Il dépend de ***Core***. Son rôle est d'apporter le support d'un nouveau langage de programmation à *APIWatch* en implémentant le point d'extension `LanguageAnalyser`. Plusieurs composants de ce « type » (un par langage) pourront donc exister simultanément dans l'application.

XXX-Serializer

Ce composant est lui aussi un module interchangeable. Il dépend de ***Core***. Son rôle est d'apporter le support d'un nouveau format de sérialisation à *APIWatch* en implémentant les points d'extension `APIScopeSerializer` et `APIStabilityRuleSerializer`. Plusieurs composants de ce « type » (un par format de sérialisation) pourront donc exister simultanément dans l'application.

API Stability Rules

Ce composant est lui aussi un module interchangeable. Il dépend de ***Core***. Son rôle est d'apporter le support de nouvelles règles de stabilité d'*API* à *APIWatch* en implémentant le point d'extension `APIStabilityRule`. Plusieurs composants de ce « type » pourront donc exister simultanément dans l'application. Un composant implémentant les règles de base de stabilité d'*API* décrites au paragraphe 3.2.3 sera embarqué par défaut dans *APIWatch*.

Command-Line Interface

Ce composant permet l'invocation des services de *APIWatch* via une interface en ligne de commande comme décrit en 3.2.6. Il a une dépendance forte vers ***Core***, et un couplage lâche vers les composants ***XXX-Analyser***, ***XXX-Serializer*** et ***API Stability Rules*** à travers le mécanisme de points d'extension.

Web Interface

Ce composant permet la persistance et l'accès aux données analysées par *APIWatch* par le biais d'une interface web comme décrit en 3.2.6. Il a une dépendance forte vers ***Core***, et un couplage lâche vers les composants ***XXX-Serializer*** à travers le mécanisme de points d'extension.

§

On trouvera un schéma représentant tous ces composants et leurs interdépendances dans l'Illustration 15.

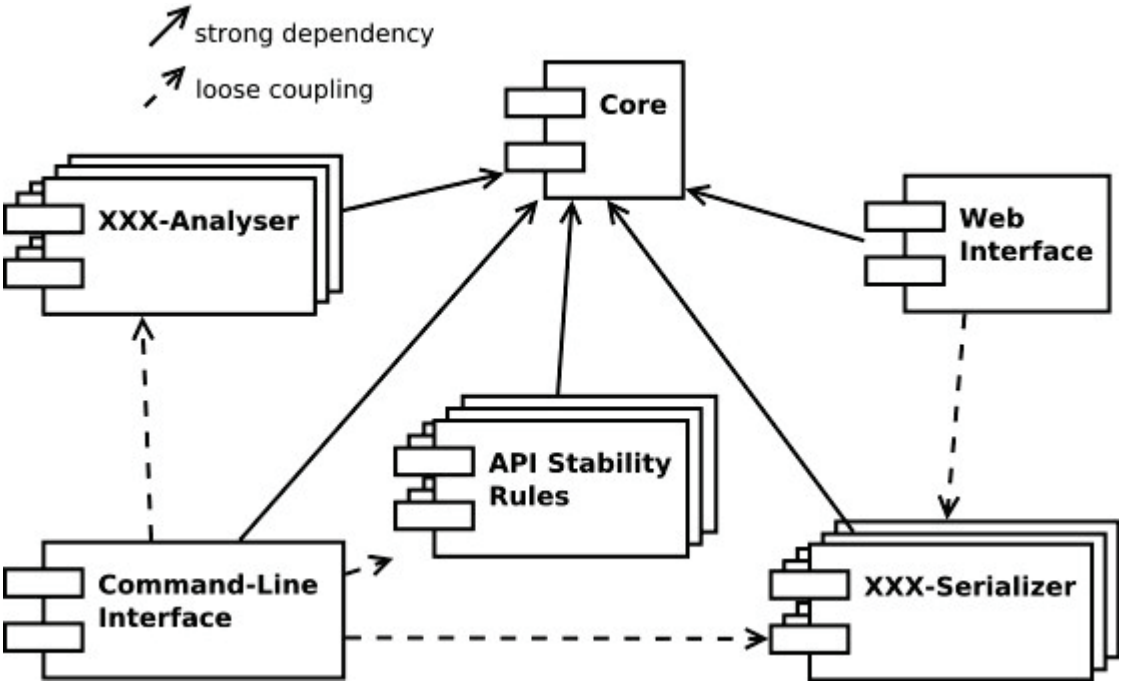


Illustration 15: Structure et dépendances des composants de l'application APIWatch

3.3. Mise en œuvre

D'une manière générale, *APIWatch* a été conçu et développé en prenant en compte les considérations suivantes (par ordre de priorité) :

1. ***Modularité*** : Le choix d'une technologie ne doit jamais être irréversible. Il faut éviter au maximum de faire reposer l'application sur une bibliothèque logicielle précise et aménager la possibilité de remplacer une dépendance par une autre en cas de besoin.
2. ***Simplicité*** : Il est primordial de ne considérer que les solutions les plus simples et triviales. Steve Oualline souligne ce point dans son livre *Practical C Programming* [4] : *« You should take every opportunity to make sure that your program is clear and easy to understand. Do not be clever. Clever kills. Clever makes for unreadable and unmaintainable programs. »* .
3. ***Faible empreinte*** : On voit de plus en plus d'applications développées sur la base de *frameworks*[19]. Cette pratique a de nombreux avantages (gain de temps, standardisation, réutilisation, etc.). Néanmoins, elle conduit souvent les développeurs à négliger l'optimisation de leur code. Le *framework* utilisé répond généralement au besoin mais propose également d'autres services parfois superflus. Cela implique une empreinte de fonctionnement inutilement grande (espace disque et mémoire nécessaires). Dans le contexte, d'un outil spécialisé comme celui-ci, où l'essentiel de la complexité réside dans les algorithmes plutôt que dans l'environnement, le gain de temps apporté par l'utilisation d'un *framework* complexe serait minime.

3.3.1. Choix techniques

Langage

APIWatch sera développé en langage *Java* pour bénéficier des avantages suivants :

- Langage mature créé en 1996[20] par *Sun Microsystems*.
- Contrôle fort du typage à la compilation (prévention d'erreurs).
- Langage orienté-objet, adapté pour implémenter le modèle de données décrit au paragraphe 3.2.
- Exécution dans une machine virtuelle (sécurité de fonctionnement).

19 Un *framework* est un kit de composants structurels, qui sert à créer les fondations ainsi que les grandes lignes de tout ou d'une partie d'un logiciel.

20 http://en.wikipedia.org/wiki/Java_version_history

- La machine virtuelle *Java* présente des performances acceptables[21] pour la plupart des usages relatifs à *APIWatch* (lecture de fichiers, manipulation de chaînes de caractères, recherche dans des tables de hachage, etc.).
- Support de multiples plate-formes sans compilation spécifique (*Linux* et autres **NIX*, *Windows*, etc.).
- Large panel de bibliothèques de fonctions réutilisables.
- Outillage étendu (environnements de développement, outils de compilation et assemblage, automatisation de la gestion des dépendances externes et des tests unitaires, etc.).
- Communauté *open-source* active.

Intégration

Maven[22] a été retenu pour gérer l'intégration de *APIWatch*. *Maven* est un outil de production de projets logiciels *Java* comparable au système *Make* sous *UNIX*. *Maven* utilise un paradigme connu sous le nom de *Project Object Model* (*POM*) afin de décrire un projet logiciel, ses dépendances avec des modules externes et l'ordre à suivre pour sa production. Il est livré avec un grand nombre de tâches pré-définies, comme la compilation de code *Java* ou encore sa modularisation.

Analyse de code source

Pour l'analyse de code source, *APIWatch* s'appuiera sur *ANTLR*[23]. *ANTLR* est un successeur des outils *Lex*[24] et *YACC*[25] écrit en *Java*. Il est distribué sous la licence BSD. Il se base sur l'écriture de grammaires décrivant un langage de programmation. Il utilise ensuite ces grammaires pour générer le code source nécessaire à l'analyse de ce langage. Plus de détails seront donnés sur le

21 Il est extrêmement délicat de parler de « performance » d'un programme informatique. Le terme est éminemment subjectif et imprécis. Il dépend du contexte d'utilisation et des contraintes imposées. Par exemple, un traducteur Français-Anglais qui donne un résultat en 5 secondes pour un texte de 100 mots a une performance tout à fait acceptable. En revanche, un calculateur de trajectoire de fusée qui donne un résultat dans le même temps est totalement inutilisable.

22 http://maven.apache.org/

23 ***AN**other **T**ool for **L**anguage **R**ecognition* est un outil créé et développé par *Terrence Parr*. http://www.antlr.org/

24 *Lex* est un générateur d'analyseurs lexicaux écrit par *Mike Lesk* et *Eric Schmidt* et utilisé sur les plate-formes *UNIX* depuis 1975. Son fonctionnement est décrit dans l'ouvrage *UNIX TIME-SHARING SYSTEM: UNIX PROGRAMMER'S MANUAL* http://cm.bell-labs.com/7thEdMan/v7vol2b.pdf

25 ***Y**et **A**nother **C**ompiler **C**ompiler* est un outil fonctionnant de pair avec *Lex*. Il permet de générer des compilateurs depuis une définition grammaticale d'un langage de programmation.

mécanisme d'analyse de code source – avec ou sans *ANTLR* – dans le paragraphe 3.3.3.

Web

L'interface *web* de *APIWatch* se basera sur le standard `javax.servlet`[26]. Ceci permettra un déploiement aisé dans tous les environnements qui supportent ce standard (*Apache Tomcat*[27], *Jetty Server*[28], *JBoss*[29], *IBM Websphere*[30], etc.).

Afin de respecter une architecture du type « Modèle-Vue-Contrôleur[31] », la génération des pages *HTML* se fera grâce au moteur de *templates*[32] *Apache Velocity*[33].

Interface avec les bases de données relationnelles

Il a été jugé préférable de découpler au maximum la communication entre *APIWatch* et le système de gestion de bases de données pour permettre notamment de changer de mécanisme de stockage si nécessaire. Aussi, cela nous affranchit d'écrire les requêtes dans le langage propriétaire[34] dudit système. On a donc choisi d'utiliser un *ORM*[35] qui permet d'avoir une couche d'abstraction entre le langage de programmation et la base de données sous-jacente. Pour rester cohérent avec les considérations mentionnées au début du chapitre 3.3, un *ORM* le plus simple et léger possible a été choisi : *ORMLite*[36]. Il est écrit en *Java* et supporte le standard *JDBC*[37] ce qui le rend compatible avec la plupart des moteurs de base de données.

---

26 http://docs.oracle.com/javaee/6/api/javax/servlet/package-summary.html
27 http://tomcat.apache.org/
28 http://jetty.codehaus.org/jetty/
29 http://www.jboss.org/
30 http://www-01.ibm.com/software/websphere/
31 Le modèle de conception Modèle-Vue-Contrôleur (ou MVC) est une pratique qui vise à séparer les rôles dans une application interactive (potentiellement graphique). Le modèle se charge de gérer les données de l'application. La vue est responsable de la présentation de celles-ci à l'utilisateur. Le contrôleur lui se charge de la médiation entre le modèle et la vue. http://en.wikipedia.org/wiki/Model-view-controller
32 Un moteur de *templates* utilise un mécanisme de « texte à trous » pour générer des documents. http://en.wikipedia.org/wiki/Template_engine_(web)
33 http://velocity.apache.org/
34 La plupart des systèmes de gestion de bases de données relationnelles utilisent un langage dérivé du standard *SQL* (*Structured Query Language*). Cependant, chaque système possède ses particularités. Notamment pour ce qui est du typage des données.
35 Un mapping objet-relationnel (en anglais *object-relational mapping* ou *ORM*) est une technique de programmation informatique qui crée l'illusion d'une base de données orientée objet à partir d'une base de données relationnelle en définissant des correspondances entre cette base de données et les objets du langage utilisé.
36 http://ormlite.com/
37 ***J**ava **D**ata**B**ase **C**onnectivity* http://en.wikipedia.org/wiki/Java_Database_Connectivity

3.3.2. Modularité et extensibilité

Pour mettre en place le système de *plug-ins* mentionné dans le chapitre 3.2.4, on utilisera un mécanisme introduit dans la version 6 de *Java* : `JavaServiceLoader`[38]. Il permet de rechercher les implémentations d'une interface ou classe abstraite visibles depuis un `ClassLoader`[39].

Dans le composant ***Core*** de *APIWatch*, le service `Analyser` va découvrir les implémentations de l'interface `LanguageAnalyser` grâce au code *Java* suivant :

```
ServiceLoader<LanguageAnalyser> loader =
                            ServiceLoader.load(LanguageAnalyser.class);
Map<String, LanguageAnalyser> analysers =
                            new HashMap<String, LanguageAnalyser>();

for (LanguageAnalyser impl : loader) {
    for (String fileExt : impl.fileExtensions()) {
        analysers.put(fileExt, impl);
    }
}
```

L'application *APIWatch* est composée de plusieurs modules au format *JAR*[40]. Pour contribuer à un ou plusieurs des points d'extension définis au paragraphe 3.2.4, il suffit de fournir un fichier `.jar` contenant :

- Les classes implémentant les interfaces concernées par le ou les points d'extension.
- Un ou plusieurs fichiers qui portent le nom qualifié des interfaces concernées, placés dans le dossier `META-INF/services`. Ces fichiers doivent contenir le nom qualifié des classes implémentant les interfaces concernées.

Afin de clarifier les choses, voici un exemple pratique :

On souhaite rajouter le support du langage C à *APIWatch*.

- On écrit donc une classe `CAnalyser` qui implémente l'interface `LanguageAnalyser`.
- On crée également un fichier `org.apiwatch.analyser.LanguageAnalyser`. Dans ce fichier, on renseigne le nom qualifié de l'implémentation `org.apiwatch.analyser.CAnalyser`.

38 http://docs.oracle.com/javase/6/docs/api/java/util/ServiceLoader.html
39 En *Java*, un `ClassLoader` est un objet responsable de charger des classes en mémoire. http://docs.oracle.com/javase/6/docs/api/java/lang/ClassLoader.html
40 Spécification du format *JAR* : http://docs.oracle.com/javase/1.4.2/docs/guide/jar/jar.html

Il suffit ensuite de générer un fichier `.jar` contenant ces fichiers :

```
org/apiwatch/analyser/CAnalyser.class
META-INF/services/org.apiwatch.analyser.LanguageAnalyser
```

Dans la distribution de *APIWatch* il y aura un dossier `lib` contenant tous les fichiers `.jar`. Au démarrage des outils en ligne de commande, tous ces fichiers seront ajoutés au *class path*[41] de l'application. Il suffit donc de placer notre fichier `.jar` créé dans ce dossier `lib`.

3.3.3. Analyse formelle de code source

Il a été privilégié de laisser une grande liberté quant aux détails d'implémentation des analyseurs spécialisés pour chaque langage de programmation. Dans le paragraphe suivant, on donnera cependant un aperçu d'une manière courante[42] d'analyser du code source et d'en extraire les données d'*API*.

Pour analyser un langage formel, on procède en général en plusieurs étapes qui transforment une donnée en une représentation intermédiaire de celle-ci (en anglais, *Intermediate Representation* ou *IR*). On considérera que le code source est représenté par un flux de caractères (la plupart du temps provenant de la lecture du contenu d'un fichier).

Analyse lexicale

Cette étape consiste à découper le flux de caractères en unités lexicales ou lexèmes[43] (en anglais, *tokens*) sans se préoccuper de leur signification. En anglais un analyseur lexical est appelé un *lexer*.

Certains *tokens* sont constants, comme les opérateurs ou les mots-clés du langage, d'autres sont variables, comme les nombres ou les chaînes de caractères. On utilise souvent des expressions régulières[44] pour décrire les *tokens* valides d'un langage.

---

41 Le *class path* d'une application *Java* est une liste de chemins dans lesquels le `ClassLoader` par défaut va rechercher les classes à charger. Ces chemins peuvent pointer vers des fichiers `.jar` ou des dossiers contenant des fichiers `.class`

42 La discipline d'analyse de langages est bien trop complexe pour être couverte entièrement dans ce mémoire. Pour les besoins de ce mémoire, je me suis largement inspiré du livre écrit par Terrence Parr [2].

43 Pour un langage de programmation un token est un « mot » valide du langage. Chaque langage possède un vocabulaire qui lui est propre. Par exemple, en *C*, les mots **`return`**, **`while`** ou encore `->` sont des mots réservés qui ont une signification bien précise, en revanche `printf` ou `malloc` sont des identifiants définis par l'utilisateur.

44 Une expression régulière décrit un ensemble de chaînes de caractères possibles selon une syntaxe précise. http://en.wikipedia.org/wiki/Regular_expression

Prenons comme exemple un langage mathématique simplifié dans lequel on peut écrire des multiplications de nombres entiers. Voici les définitions de *tokens* nécessaires à l'analyse lexicale d'un tel langage[45] :

```
/**
 * Opérateur de multiplication.
 *
 * Ce token est constant.
 */
STAR : '*'  ;

/**
 * Nombre entier.
 *
 * Ce token est variable, il peut être composé de un ou
 * plusieurs chiffres
 */
NUMBER : ('0'..'9')+ ;

/**
 * Espaces, tabulations et retours à la ligne.
 *
 * Ces tokens sont ignorés par le lexer car ils n'ont pas
 * de signification particulière.
 */
WS : (' '|'\t'|'\r'|'\n')+ {$channel = HIDDEN;}  ;
```

Dans l'Illustration 16 on peut voir le procédé de découpage en *tokens* d'une expression multiplicative simple. Les *tokens* de type `WS` sont placés dans un flux séparé afin qu'ils soient ignorés par l'analyseur syntaxique.

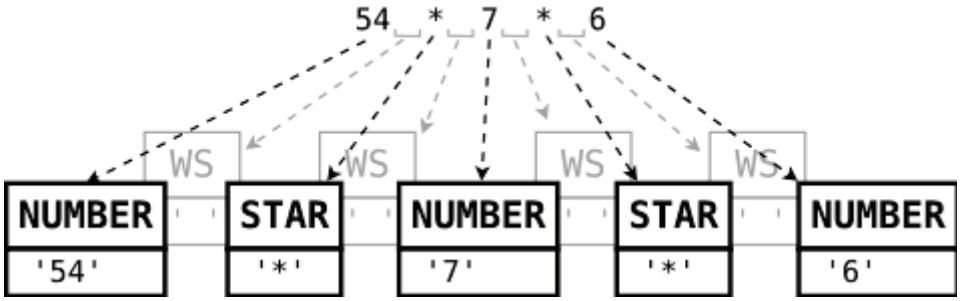


Illustration 16: Découpage d'un flux de caractère en unités lexicales

45 La syntaxe utilisée ici est celle de l'outil *ANTLR*.

Analyse syntaxique

En anglais un analyseur syntaxique est appelé un *parser*. L'étape de *parsing* consiste à reconnaître des « phrases » valides dans le flux de *tokens* produit par le *lexer*. Pour cela, on se base généralement sur une grammaire non-contextuelle[46]. Celle-ci définit récursivement l'ensemble des *tokens* valides du langage et l'ordre dans lequel ils doivent apparaître pour former des phrases correctes. Au fur et à mesure de la reconnaissance de structures valides du langage, le *parser* produit une représentation arborescente du code source analysé.

Reprenons le langage mathématique simplifié abordé au paragraphe précédent. Dans celui-ci, la règle de grammaire décrivant une expression multiplicative s'écrit de la manière suivante[47] :

```
/**
 * Définition récursive d'une expression multiplicative.
 */
multiplicative_expression
  : NUMBER (STAR multiplicative_expression)*
  ;
```

Si on évalue le résultat produit par l'exemple précédent en organisant les *tokens* de manière arborescente, voici un résultat possible :

Illustration 17: Résultat du parsing d'un flux de tokens

46 Appelée également grammaire algébrique, elle permet de décrire de manière formelle la syntaxe d'un langage.

47 Les identifiants en majuscules sont des références aux *tokens* produits par le *lexer*.

Dans l'Illustration 17, on constate que les *tokens* de type `STAR` ont été placés comme nœuds de l'arbre et ceux de type `NUMBER` comme feuilles. Cela permet de représenter l'expression sous une forme voisine de la notation polonaise[48].

Le résultat d'une analyse syntaxique produit toujours un « arbre de syntaxe abstraite » (appelé généralement par son nom anglais : *Abstract Syntax Tree* ou *AST*). Cet arbre porte la représentation en deux dimensions du code analysé (là où les flux de caractères et de *tokens* sont des représentations à une seule dimension).

Analyse sémantique

Une fois l'*AST* construit par le parser, il faut maintenant lui donner du « sens ». C'est l'étape la plus délicate d'un processus d'analyse de langages. Ici on ne produit plus de représentation intermédiaire mais bien une conversion finale vers le format désiré. Ce format désiré peut avoir plusieurs formes :

- Un résultat dans le cas de l'évaluation d'une expression mathématique.
- Une traduction vers un autre langage (par exemple, *C* vers *Java*).
- Des instructions en code machine dans le cas d'un compilateur.

Ces formes sont rarement obtenues directement depuis l'*AST* tel que produit par le *parser*. Il est alors nécessaire de réécrire des parties de l'*AST*[49] afin d'arriver à une forme assez triviale qui pourra être directement traduite dans le format désiré. Parfois, certaines parties du code analysé seront volontairement exclues, puisque non exploitées par les analyseurs sémantiques pour des raisons de performance. Par exemple, dans les analyseurs de *APIWatch*, on aura tendance à exclure les sous-arbres des corps de fonctions car ils ne portent que rarement – voire jamais – des informations d'*API*.

Comme l'*AST* est une structure arborescente, on utilisera généralement un algorithme du type « visiteur » pour le parcourir. Cette méthode est appelée *tree walking* en anglais. Elle consiste à analyser chaque nœud de l'arbre récursivement. Il y a deux façons de « visiter » un arbre : avec un parcours en profondeur[50] ou un parcours en largeur[51]. Selon les besoins, l'une ou l'autre de ces méthodes sera utilisée.

48 La notation polonaise représente les formules algébriques sans utiliser de parenthèses en préfixant ou suffixant les opérandes par l'opérateur.

49 Un compilateur devra souvent optimiser les instructions entrées par le programmeur avant de pouvoir générer du code machine performant.

50 Le parcours en profondeur consiste à explorer toute la sous-arborescence d'un sommet avant de visiter ses voisins.

51 Le parcours en largeur consiste à explorer chaque voisin d'un sommet avant d'en examiner la sous-arborescence.

Pour en finir avec l'exemple précédent, partons du principe que le *parser* produit un *AST* utilisant des objets de ce type (code d'une classe *Java*) :

```
class MultiExprAST {
    public String token;
    public MultiExprAST left;
    public MultiExprAST right;
/* classe volontairement incomplète pour des raisons de lisibilité */
}
```

Voici le code nécessaire pour évaluer le résultat de l'expression mathématique analysée (également en *Java*) :

```
int evaluate(MultiExprAST ast) {
    if (ast.token.equals("*"))
        return evaluate(ast.left) * evaluate(ast.right);
    else
        return Integer.valueOf(ast.token);
}
```

On constate que la fonction `evaluate` est récursive et qu'elle utilise un parcours en profondeur pour évaluer le résultat de l'expression multiplicative portée par l'*AST*.

Analyse de langages avec *ANTLR*

La programmation d'analyseurs est une tâche extrêmement fastidieuse, répétitive et sujette aux erreurs. *ANTLR* est un outil qui permet de définir un langage formel par une grammaire écrite dans une syntaxe dérivée de la *Backus-Naur Form*[52]. Depuis cette grammaire, *ANTLR* est capable de générer des analyseurs lexicaux, syntaxiques et même sémantiques, et ce dans de nombreux langages de programmation : *Java*, *C*, *C++*, *Python*, etc.[53] Dans son livre *The Definitive ANTLR Reference* [5], Terrence Parr explique comment écrire et exploiter les grammaires.

La communauté *ANTLR* met à disposition des grammaires pour de nombreux langages de programmation. Le support des langages *Java* et *C* a été implémenté dans la distribution standard de *APIWatch* à l'aide de ces grammaires.

52 La forme de Backus-Naur (souvent abrégée en BNF, de l'anglais *Backus-Naur Form*) est une notation permettant de décrire les règles syntaxiques des langages de programmation.

53 La liste complète des langages cibles supportés par *ANTLR* est disponible sur leur site web http://www.antlr.org/wiki/display/ANTLR3/Code+Generation+Targets

Exemple 1 : Java

La grammaire disponible sur le site officiel *ANTLR* était déjà très complète. Elle permettait de construire des *AST* depuis n'importe quel code *Java* (édition 5 ou 6). Il a uniquement été nécessaire d'écrire une fonction récursive `walk` qui, au fur et à mesure du parcours de l'*AST*, construit une structure de données utilisant le modèle d'interface logicielle décrit au paragraphe 3.2.1.

Exemple 2 : C

L'unique grammaire *C* disponible a été écrite par l'auteur de *ANTLR* lui-même : Terrence Parr. Malheureusement, elle prenait en charge uniquement le *C ANSI 89*[54] et elle n'était pas en mesure de produire des *AST*. Celle-ci a dû être modifiée pour qu'elle supporte les nouveaux éléments introduits dans les standards *C ANSI* de 1999 et 2011. On a également ajouté le support des *built-ins*[55] et de l'opérateur `__attribute__` du compilateur *GCC*[56].

Les programmes écrits en langage *C* utilisent généralement des instructions destinées à un préprocesseur[57] de texte. Ces instructions permettent l'inclusion de segments de code contenu dans d'autres fichiers (avec l'instruction `#include`), la substitution de chaînes de caractères (définies avec `#define`) ou encore de la compilation conditionnelle (avec `#ifdef` et `#ifndef`). Or, la syntaxe de ces instructions n'a aucun rapport avec celle du langage *C*. Il est donc obligatoire que chaque fichier que l'on souhaite analyser avec *APIWatch* soit traité auparavant par le préprocesseur.

Cela entraîne un problème majeur : après le passage du préprocesseur, le code écrit par le développeur va être noyé au milieu de toutes les inclusions dont il a eu besoin. Une majorité de ces inclusions provenant des bibliothèques du système, elles apportent des définitions qui n'ont rien à voir avec le code original. Si on les analyse, elles vont produire du « bruit » et l'*API* donnée en résultat sera fausse. Pour clarifier les choses, voici un exemple. Prenons le code *C* suivant contenu dans un fichier `hello.c` :

```
#include <stdio.h>
int say_hello(char *name, FILE *stream) {
     return fprintf(file, "Hello %s!\n", name);
}
```

54 L'organisme ANSI (*American National Standards Institute*) a produit plusieurs spécifications standard du langage *C*. Il y a eu trois révisions de ce standard en 1989, 1999 et 2011.

55 Les *built-ins* sont des fonctions élémentaires prises en charge par certains compilateurs. Ces fonctions font généralement appel à des instructions matérielles complexes telles que l'arithmétique à virgule flottante ou les opérations de synchronisation entre processeurs.

56 *GCC* est un compilateur *open-source* largement utilisé du logiciel libre. http://gcc.gnu.org/

57 *GCC* propose également un préprocesseur. http://gcc.gnu.org/onlinedocs/cpp/

On constate que le code définit une unique fonction `say_hello` et tient dans 4 lignes. Si on fait passer ce code par le préprocesseur de *GCC* avec la commande suivante :

```
gcc -E hello.c > hello.i
```

On obtient un fichier `hello.i` de 851 lignes qui ressemble à ceci (la quasi-totalité du fichier a été masquée pour des raisons de lisibilité) :

```
[...]
# 45 "/usr/include/stdio.h" 3 4
struct _IO_FILE;
[...]
# 65 "/usr/include/stdio.h" 3 4
typedef struct _IO_FILE FILE;
[...]
# 322 "/usr/include/stdio.h" 3 4
extern int fprintf (FILE *__restrict __stream,
                    __const char *__restrict __format, ...);
[...]
# 2 "hello.c" 2
int say_hello(char *name, FILE *stream) {
     return fprintf(file, "Hello %s!\n", name);
}
```

Seules les 3 dernières lignes définissent l'*API* du code de l'utilisateur. Tout le reste fait partie du « bruit » évoqué plus haut. Néanmoins, afin que l'analyseur reconnaisse la fonction `fprintf` et le type `FILE` utilisés dans la fonction `say_hello`, il doit avoir rencontré leur définition avant – ceux-ci ne faisant pas partie des mots-clés du langage *C*. Tout ce « bruit » est donc nécessaire mais il faut trouver un moyen de l'exclure des données d'*API*.

Il se trouve que le préprocesseur insère des directives spécifiques qui permettent au compilateur de savoir d'où provient chaque ligne de code (nom de fichier et numéro de ligne)[58]. Notamment pour lui permettre de localiser précisément les erreurs de syntaxe s'il en trouve. Ces directives respectent une syntaxe bien définie :

```
# <n° de ligne> "chemin du fichier source" <indicateurs divers>
```

Il est important de noter que l'ensemble des fichiers du système et/ou des bibliothèques susceptibles d'être inclus par le code de l'utilisateur sont localisés à des endroits précis et généralement connus.

58 Cette fonctionnalité est décrite en détail sur le site du compilateur *GCC* http://gcc.gnu.org/onlinedocs/cpp/Preprocessor-Output.html

On procédera donc de la manière suivante :

- On traite le fichier source original avec le préprocesseur et on stocke le résultat dans un fichier temporaire qui sera analysé.
- On répertorie l'ensemble des chemins (dossiers) qui contiennent des entêtes du système et/ou provenant de bibliothèque externes.

  ```
  /usr/include
  ```

  ```
  /usr/lib64/gcc/x86_64-pc-linux-gnu/4.5.3/include
  ```

- Durant l'analyse lexicale, le *lexer* va maintenir une table contenant les informations de provenance des inclusions. Il va mémoriser la ligne à laquelle l'inclusion a été rencontrée, le fichier et la ligne de ce fichier depuis lequel elle a été faite. Comme il connaît l'ensemble des chemins qui ne sont pas en rapport avec le code utilisateur, le *lexer* peut identifier si ce qui suit l'inclusion fait partie ou non d'un fichier « système ».

Ligne dans le fichier analysé	***Chemin du fichier source***	***Ligne dans le fichier source***	***Fichier « système »***
	...		
133	/usr/include/stdio.h	45	true
	...		
484	/usr/include/stdio.h	322	true
	...		
847	hello.c	2	false

Table 24: Exemple de table de provenance des inclusions produite par le lexer.

- Il est important de noter que chaque *token* produit par le *lexer* contient la ligne à laquelle il a été détecté. Durant l'analyse syntaxique, avant d'insérer un *token* dans l'*AST*, le *parser* va inspecter la table produite par le *lexer* (en plus du flux de *tokens*) et va donc pouvoir décider si le *token* fait partie du code écrit par l'utilisateur en comparant les numéros de ligne.

Grâce à cette méthode, l'*AST* produit par le *parser* ne contiendra que les symboles définis par l'utilisateur. On procédera alors de la même façon que pour le langage *Java* pour générer une structure de données à base des modèles définis au paragraphe 3.2.1.

3.3.4. Algorithme de comparaison

Pour détecter des modifications entre deux versions d'une même *API*, on doit comparer deux structures de données construites à base des modèles définis au paragraphe 3.2.1. Rappelons que les données d'*API* produites par *APIWatch* sont en structures arborescentes.

Il existe de nombreux algorithmes de comparaison d'arbres. Kuo-Chung Tai en propose un dans son article *The Tree-To-Tree Correction Problem* [6] qui permet de calculer la « distance » entre deux arbres A et A', c'est-à-dire le nombre d'opérations élémentaires[59] nécessaires pour transformer A en A' ou *vice-versa*. L'algorithme présenté résout ce problème avec une complexité de $O(n * n' * p^2 * p'^2)$, où n et n' sont les nombres de nœuds dans A et A', respectivement, et p et p' sont les niveaux de profondeur de A et A', respectivement. Ce niveau de complexité est bien trop élevé pour être viable dans un logiciel comme *APIWatch*. En effet, les données d'*API* manipulées pourront atteindre des tailles considérables[60] et il est impératif que leur comparaison reste une opération rapide – sinon triviale.

On proposera ici un algorithme plus simple qui part du principe essentiel que chaque élément de l'arbre est identifiable de manière unique. Cet algorithme se base sur une propriété des tables de hachage[61] : la recherche d'un élément par sa clé de hachage se fait en $O(1)$ quel que soit le nombre d'éléments dans la table si chaque clé de hachage est unique.

La première étape consiste à « aplatir » les deux arbres. On va parcourir la structure arborescente et insérer chaque `APIElement` rencontré dans une table de hachage. La clé utilisée sera le résultat de la méthode `path()` de la classe `APIElement`. Comme la structure est un arbre, il ne peut pas y avoir deux éléments qui ont la même valeur de `path()`. Prenons l'exemple d'un arbre d'*API* simple comme décrit sur l'Illustration 18 :

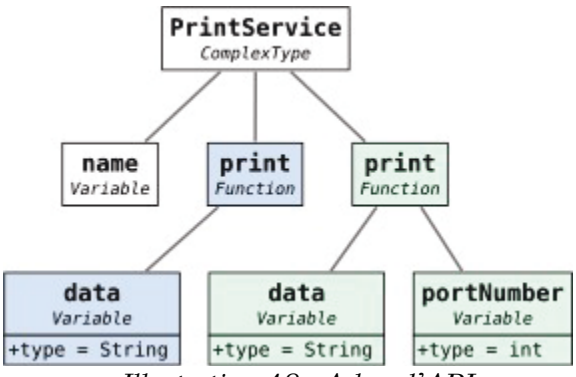


Illustration 18: Arbre d'API

59 Une opération élémentaire consiste à changer, supprimer ou ajouter un nœud d'un arbre.

60 Il est impossible – et inutile – de donner des estimations de taille ici. Un algorithme qui donne une solution en $O(n * n' * p^2 * p'^2)$ n'est pas exploitable dans notre cas.

61 C'est une structure de données qui permet une association clé-élément.

Si on calcule les valeurs `path()` de chaque élément et qu'on les insère dans une table de hachage, on obtient le résultat montré dans l'Illustration 19 :

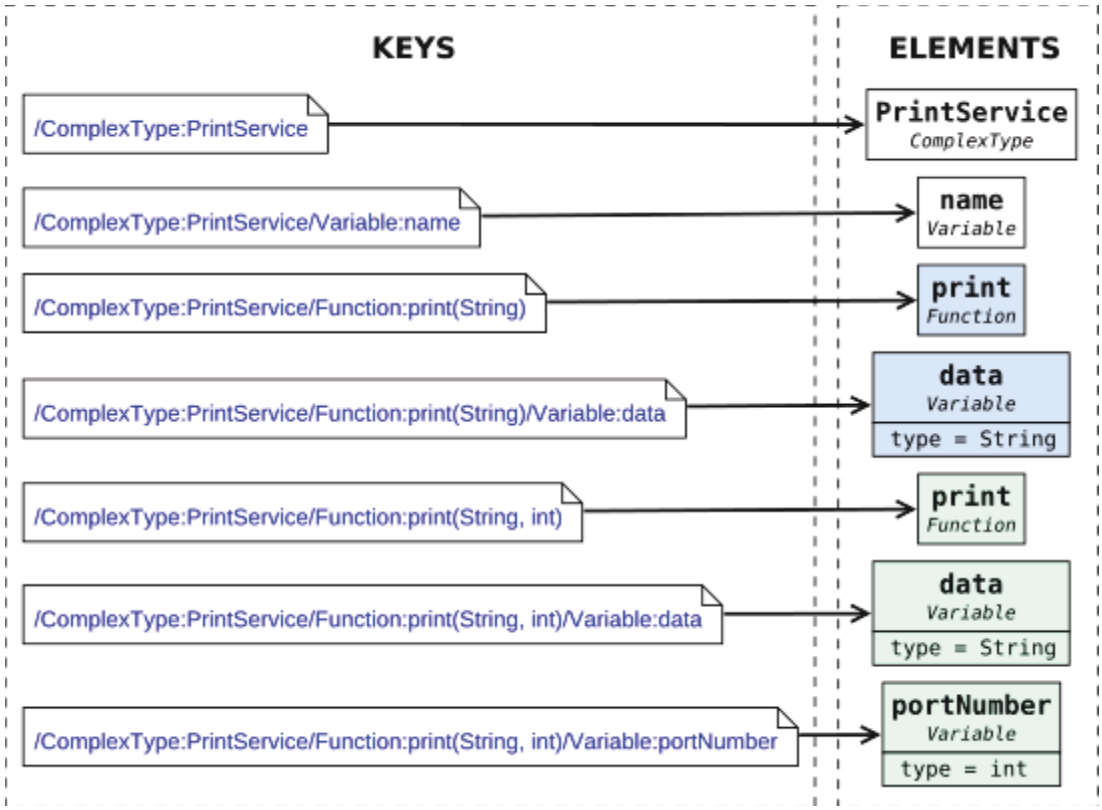


Illustration 19: Arbre d'API « aplati » dans une table de hachage

Voici la fonction *Java* qui permet « d'aplatir » un `APIElement` et toute sa sous-arborescence dans une table de hachage :

```
Map<String, APIElement> flatten(APIElement e, Map<String, APIElement> map) {
      map.put(e.path(), e);
      if (e instanceof APIScope)
            for (APIScope scope : ((APIScope) e).subScopes)
                  flatten(scope, map);

      if (e instanceof SymbolContainer)
            for (Symbol symbol : ((SymbolContainer) e).symbols())
                  flatten(symbol, map);

      if (e instanceof Function)
            for (Variable arg : ((Function) e).arguments)
                  map.put(arg.path(), arg);

      return map;
}
```

La fonction `flatten` sera appelée sur les éléments racines des deux versions *A* et *B* du composant logiciel concerné.

```
Map<String, APIElement> elementsA =
                flatten(versionA, new HashMap<String, APIElement>());
Map<String, APIElement> elementsB =
                flatten(versionB, new HashMap<String, APIElement>());
```

Une fois les arbres « aplatis » dans deux tables de hachage, il suffit de parcourir celles-ci pour détecter les ajouts et suppressions d'éléments, ainsi que les éléments communs. Voici le code *Java* nécessaire pour effectuer ces comparaisons :

```
/* Elements de la version A également présents dans la version B */
Map<String, APIElement> commonA = new HashMap<String, APIElement>();
/* Elements de la version B également présents dans la version A */
Map<String, APIElement> commonB = new HashMap<String, APIElement>();
List<APIElement> added = new ArrayList<APIElement>();
List<APIElement> removed = new ArrayList<APIElement>();
for (Map.Entry<String, APIElement> e : elementsA.entrySet()) {
     if (elementsB.containsKey(e.getKey())) {
          commonA.put(e.getKey(), e.getValue());
     } else {
          removed.add(e.getValue());
     }
}
for (Map.Entry<String, APIElement> e : elementsB.entrySet()) {
     if (elementsA.containsKey(e.getKey())) {
          commonB.put(e.getKey(), e.getValue());
     } else {
          added.add(e.getValue());
     }
}
```

On remarquera que les ensembles d'éléments communs aux deux versions sont également placés dans des tables de hachage. Pour calculer les réelles différences entre les éléments communs, il va falloir comparer en détail chaque élément de `commonA` avec son homologue dans `commonB`. L'utilisation de tables de hachage permet d'accéder à l'élément correspondant dans l'ensemble `commonB` avec une complexité de *O(1)*. On utilisera ensuite une méthode `getDiffs` de la classe `APIElement` qui renvoie une liste de `APIDifference`, en comparant en détail chaque attribut des deux objets. Les éléments qui ont été ajoutés ou supprimés, seront utilisés pour créer des `APIDifference` de type `ADDED` et `REMOVED`.

Voici le code *Java* permettant de créer la liste des différences :

```
List<APIDifference> diffs = new ArrayList<APIDifference>();
for (Map.Entry<String, APIElement> e : commonA.entrySet()) {
     APIElement eltA = e.getValue();
     /* L'accès à eltB se fait en O(1) */
     APIElement eltB = commonB.get(e.getKey());
     diffs.addAll(eltA.getDiffs(eltB));
}
for (APIElement e : removed) {
     diffs.add(new APIDifference(ChangeType.REMOVED, e, null));
}
for (APIElement e : added) {
     diffs.add(new APIDifference(ChangeType.ADDED, null, e));
}
```

Déterminons à présent la complexité de l'algorithme proposé, appliqué sur deux structures de données d'*API* contenant respectivement n et n' éléments et de profondeur p et p', ces structures ayant C éléments en commun. Pour la suite, on posera les équivalences suivantes :

$$N = max(n,\ n')$$
$$P = max(p,\ p')$$
$$C \leq N$$

La Table 25 montre que l'algorithme proposé effectue la comparaison entre deux arbres en $O(N * P)$. Ceci le rend exploitable même sur des jeux de données de taille considérable.

Opération	***Complexité réelle***	***Complexité équivalente***
Aplatissement des données d'*API*. On considérera que le calcul de la clé de hachage d'un élément – avec la méthode `path` – se fait en $O(d)$ au maximum pour un arbre de profondeur d.	$O(n * p + n' * p')$	$O(N * P)$
Détection des éléments ajoutés et supprimés.	$O(n + n')$	$O(N)$
Calcul des différences sur les éléments communs (C est borné par N).	$O(C)$	$O(N)$
Total		***$O(N * P)$***

Table 25: Complexité des étapes de l'algorithme de comparaison proposé

3.4. Fonctionnement et utilisation

Dans ce chapitre, on proposera un cas d'utilisation de *APIWatch*. Il sera supposé que la machine virtuelle *Java* est déjà installée sur la machine cible et que l'exécutable `java` est accessible depuis le `PATH` système. On considérera également qu'un serveur d'applications *Java* (*Apache Tomcat*) est installé et accessible à l'adresse http://localhost:8080/.

3.4.1. Installation

La distribution de l'application se présente sous la forme d'une archive au format `.tar.gz`. Une fois l'archive copiée sur la machine cible, il faut la décompresser – de préférence dans un répertoire protégé du système afin d'éviter qu'il soit modifiable par les utilisateurs :

```
root:~ # cd /usr/lib
root:lib # tar zxvf /tmp/apiwatch-1.0.tar.gz
apiwatch-1.0/lib/apiwatch-core-1.0.jar
apiwatch-1.0/lib/antlr-runtime-3.3.jar
...
apiwatch-1.0/lib/apiwatch-rules-base-1.0.jar
apiwatch-1.0/war/apiwatch-war-1.0.war
apiwatch-1.0/conf/rules-config.ini
...
apiwatch-1.0/bin/apiwatch
apiwatch-1.0/bin/apidiff
apiwatch-1.0/bin/apiscan
```

Il est recommandé d'ajouter le chemin `/usr/lib/apiwatch-1.0/bin` au `PATH` système pour que les exécutables `apiwatch`, `apidiff` et `apiscan` soient accessibles facilement. La commande suivante permet de vérifier que les outils en ligne de commande ont été correctement installés :

```
root:~ # apiwatch --help
usage: apiwatch [-h] [-e ENCODING] [-x PATTERN [PATTERN ...]]
                [-i PATTERN [PATTERN ...]] [-u USERNAME] [-p PASSWORD]
                [-v {TRACE,DEBUG,INFO,WARN,ERROR}] [-f {text,json}]
                [-r RULES_CONFIG] [-s {INFO,MINOR,MAJOR,CRITICAL,BLOCKER}]
                REFERENCE FILE_OR_DIR [FILE_OR_DIR ...]

APIWATCH version 1.0

...
```

Tous les outils en ligne de commande de *APIWatch* ont une option `-h` ou `--help` qui permet d'afficher la manière de les invoquer ainsi que leurs différentes options.

Pour installer l'application *web* dans le serveur d'applications, il faut décompresser le fichier `/usr/lib/apiwatch-1.0/war/apiwatch-war-1.0.war` dans un dossier `webapps/apiwatch/` du serveur *Tomcat*. Puis relancer ce serveur pour qu'il prenne en compte la présence d'une nouvelle application *web*.

```
root:~ # cd /var/lib/tomcat-6/webapps
root:webapps # unzip /usr/lib/apiwatch-1.0/war/*.war -d apiwatch
Archive:  /usr/lib/apiwatch-1.0/war/apiwatch-war-1.0.war
 creating: apiwatch/META-INF/
 inflating: apiwatch/META-INF/MANIFEST.MF
...
root:webapps # service tomcat-6 restart
 * Stopping tomcat-6 ...    [ ok ]
 * Starting tomcat-6 ...    [ ok ]
```

L'application *web* *APIWatch* devrait être accessible à l'adresse http://localhost:8080/apiwatch/. Sur l'Illustration 20 on peut voir une capture d'écran de la page d'accueil (aucun composant logiciel n'a encore été analysé).

Illustration 20: Page d'accueil de l'application web après le premier démarrage

3.4.2. Analyse de code source

L'analyse simple de code source se fait avec l'outil `apiscan`. Celui-ci permet de stocker les résultats d'une analyse dans un fichier ou dans un serveur *APIWatch*.

Ici, on souhaite analyser l'*API* exposée par le code source d'un composant logiciel appelé `jenkins-core` pour deux versions de celui-ci : `1.447` et `1.466`. On considérera que l'appel de l'outil `apiscan` se fait depuis un dossier contenant tous les fichiers de code source.

Voici la commande nécessaire pour analyser récursivement le contenu du répertoire courant. L'option `-o/--output` permet de spécifier l'emplacement où l'on souhaite stocker le résultat de l'analyse. Ici on a fourni l'*URL* du serveur *APIWatch* mais on peut également donner un chemin vers un répertoire et le résultat sera stocké dans un fichier `<répertoire>/api.json`. Si l'option `-o/--output` est omise, le résultat est envoyé vers la sortie standard :

```
user:src $ apiscan * -o http://localhost:8080/apiwatch/jenkins-core/1.447/
[INFO] Sent results to URL: http://localhost:8080/apiwatch/jenkins-core/1.447/
```

Quand on choisit d'envoyer le résultat vers le serveur *APIWatch*, il faut spécifier une *URL* respectant la forme suivante[62] :

```
<URL de base de l'application><nom du composant>/<version>/
```

Sur l'Illustration 21 on peut voir la page d'accueil après avoir analysé deux versions du composant `jenkins-core`.

Illustration 21: Page d'accueil après analyse de deux versions du composant `jenkins-core`

62 Si une version a déjà été stockée, il ne sera pas possible de l'écraser.

En cliquant sur le nom du composant, on accède au détail des versions de celui-ci :

Illustration 22: Versions d'un composant logiciel

En cliquant sur le nom d'une des versions on peut visualiser les données d'*API* analysées :

Illustration 23: Données d'API pour une version

Les icônes utilisées sur la page représentée dans l'Illustration 23 correspondent aux objets du modèle d'*API* décrit au chapitre 3.2.1. Certaines icônes sont décorées en fonction de la visibilité de l'élément qu'elles représentent. La couleur de l'icône ou de la décoration donne la visibilité (rouge pour `PRIVATE`, jaune pour `PROTECTED` et bleu pour `SCOPE`). Si l'icône est verte ou n'a pas de décoration, la visibilité de l'élément est `PUBLIC` :

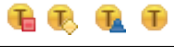
Icône(s)	*Objet*
	`APIScope`
	`Variable`
	`Function`
	`ComplexType`
	`SimpleType` et `ArrayType`
	Dépendance

Table 26: Icônes utilisées pour représenter le modèle d'API

Sur cette page, on peut filtrer les éléments que l'on souhaite afficher par plusieurs critères :

- ***Leur nom*** grâce au champ de recherche.
- ***Leur visibilité*** grâce à la liste déroulante rouge en haut à droite.

3.4.3. Comparaison entre deux versions

Une fois les deux versions analysées et stockées dans la base de données, on peut évaluer les différences d'*API* et les violations qu'elles entraînent. Ceci peut être fait de deux façons différentes :

Via l'interface web

Sur la page d'un composant, il est possible de sélectionner deux versions que l'on souhaite comparer (voir Illustration 24). En cliquant sur le bouton *View differences between selected A and B versions*, le serveur *APIWatch* va effectuer une comparaison de l'*API* des deux versions sélectionnées. Il va ensuite évaluer toutes les différences trouvées à l'aide de règles de stabilité d'*API*, chaque différence pouvant donc donner lieu à une ou plusieurs violations. L'Illustration 25 montre le résultat de la comparaison entre les versions `1.447` et `1.466` du composant `jenkins-core`. Pour certaines violations, il est possible d'avoir plus de détails en cliquant sur le bouton avec un symbole de loupe. L'Illustration 26 montre la boîte de dialogue affichée dans ce cas.

Illustration 24: Sélection des versions à comparer

Illustration 25: Violations de stabilité d'API entre deux versions d'un même composant

Illustration 26: Détail d'une violation de stabilité d'API

Par ligne de commande

Grâce à l'outil `apidiff`, on peut évaluer les violations à la stabilité d'une *API* entre deux versions d'un même composant logiciel.

La syntaxe de base de la commande est : `apidiff <version A> <version B>` Les versions peuvent être un chemin de fichier ou une URL du serveur. Voici un exemple de résultat obtenu si on compare les deux versions enregistrées précédemment :

```
user:~ $ apidiff http://localhost:8080/apiwatch/jenkins-core/1.447/ \
                     http://localhost:8080/apiwatch/jenkins-core/1.466/
[CRITICAL] <REM001> Removed PROTECTED Function:configure() @
'hudson\ExtensionFinder.java:416'
[BLOCKER] <REM001> Removed PUBLIC Variable:LOG_STARTUP_PERFORMANCE @
'jenkins\model\Jenkins.java:3610'
...
[BLOCKER] <TYP001> Changed type of PUBLIC variable 'CONFIG_DELEGATE_TO'
(Class -> Class<Plugin>) @
'hudson\os\windows\ManagedWindowsServiceConnector.java:42'
```

On peut modifier le format de sortie des violations de stabilité d'*API* détectées avec l'option `-f/--format`.

Ici, *APIWatch* a évalué les violations à l'aide de la configuration par défaut des règles de stabilité d'*API* (cf. Annexe A). On peut changer cette configuration pour l'adapter à un projet en utilisant l'option `-r/--rules-config` avec un fichier personnalisé. La version par défaut d'un tel fichier se trouve dans le répertoire `conf` de l'installation d'*APIWatch*.

3.4.4. Comparaison avec une version de référence

L'outil `apiwatch` permet de combiner les fonctionnalités de `apiscan` et `apidiff`. Il va analyser un ensemble de fichiers et dossiers, puis en comparer l'*API* avec celle d'une version de référence passée en paramètre. `apiscan` sera le plus souvent utilisé en intégration continue pour contrôler la stabilité des *API* au fur et à mesure du développement.

La syntaxe de la commande est la suivante : `apiwatch <reference> <fichiers à analyser>`. Comme pour `apidiff`, la référence peut être un fichier ou une URL du serveur.

Voici un exemple :

```
user:src $ apiwatch http://localhost:8080/apiwatch/jenkins-core/1.466/ *
[INFO] <REM001> Removed PRIVATE Function:jnlpConnect(SlaveComputer) @
'hudson\TcpSlaveAgentListener.java:314'
[INFO] <REM001> Removed PRIVATE
Function:runJnlpConnect(DataInputStream, PrintWriter) @
'hudson\TcpSlaveAgentListener.java:227'
[INFO] <REM001> Removed PRIVATE Function:getSecretKey() @
'hudson\TcpSlaveAgentListener.java:118'
...
[INFO] 20 violations.
```

La plupart des options de `apiscan` et `apidiff` s'applique également à `apiwatch`.

Conclusion

Dans le cadre de ce travail, je me suis efforcé de donner une réponse au problème soulevé dans le chapitre *Contexte et enjeux* : la maîtrise de la stabilité des *API* et de la compatibilité ascendante. Le projet *APIWatch* a été mené en suivant toutes les étapes d'un développement logiciel :

- Expression de besoin
- Spécifications formelles
- Conception et architecture
- Développement
- Intégration, tests et validation
- Écriture d'une documentation technique et utilisateur

Cet outil est modulaire, facilement évolutif et fonctionne sur la plupart des environnements. Il est aujourd'hui utilisé de manière opérationnelle par les équipes de développement des sociétés du groupe *Thales*.

Cependant, la maîtrise de la stabilité des *API* et de la compatibilité ascendante est une discipline complexe. La solution proposée dans ce mémoire permet d'automatiser une partie du travail mais elle est limitée à plusieurs niveaux :

- *APIWatch* détecte des changements dans le code source en considérant les données d'interface programmatiques (signatures de fonctions, typage, etc.). Si un composant logiciel a des interfaces passant par d'autres voies – lecture/écriture de fichiers par exemple, il ne pourra pas (dans l'état actuel des choses) en détecter les modifications.
- L'autre limitation est un corollaire de la première. *APIWatch* n'a pas été conçu pour analyser la sémantique des *API*. Il ne pourra pas détecter de changement comportementaux d'une fonction ou d'un objet.

Afin d'étendre la fonctionnalité de *APIWatch* pour prendre en compte la notion de sémantique dans les programmes analysés, une évolution que j'ai envisagée est le couplage avec des langages de spécification formelle tels que

JML[63], *ACSL*[64], ou *SPARK*[65]. Cette évolution demanderait cependant un changement du modèle décrit au chapitre 3.2.1.

Néanmoins, l'intervention humaine sera toujours nécessaire si l'on souhaite obtenir un certain niveau de confiance pour un logiciel livré.

À ce jour, *APIWatch* se présente comme une application en ligne de commande pouvant être couplée à un serveur. Une des évolutions possibles est l'intégration de cette solution dans les principaux environnements de développement comme la plate-forme *Eclipse*, pour en faciliter l'utilisation par les équipes de développement.

63 Le *Java Modeling Language* (*JML*) est un langage de spécification pour *Java*, il est basé sur le paradigme de la programmation par contrat. Il utilise la logique de *Hoare*. Les spécifications sont ajoutées dans les commentaires du code en *Java*. Plusieurs outils permettent d'exploiter les spécifications *JML*. Le plus connu est *ESC/Java* (*Extended Static Checker*). http://www.eecs.ucf.edu/~leavens/JML//index.shtml

64 *ANSI/ISO C Specification Language* (*ACSL*) est un langage inspiré de *JML* qui fait partie de la solution d'analyse de code *Frama-C* http://frama-c.com/acsl.html

65 *SPARK* est un sous-ensemble du langage *Ada* développé par *AdaCore*. Il est fourni avec un langage de spécification similaire à *JML*, ainsi qu'un ensemble d'outils de vérification http://www.adacore.com/sparkpro/

Annexes

A. Propriétés configurables des implémentations par défaut de `APIStabilityRule`

La configuration des règles de stabilité d'*API* peut être modifiée grâce à une option `-r/--rules-config` des outils en ligne de commande `apidiff` et `apiscan` (voir paragraphe 3.4.3). Une version par défaut du fichier qui doit être passé en paramètre à cette option se trouve dans le répertoire `conf` de l'installation de *APIWatch*. Ce fichier est au format standard *INI*, chaque section porte le code de la règle suivie de l'ensemble des paramètres de celle-ci. En voici un extrait :

```
;=======================================;
; API STABILITY RULES CONFIGURATION FILE ;
;=======================================;

[ADD001]
; Element Addition
enabled=true
privateSeverity=INFO
protectedSeverity=INFO
scopeSeverity=INFO
publicSeverity=INFO

[REM001]
; Element Removal
enabled=true
privateSeverity=INFO
protectedSeverity=CRITICAL
scopeSeverity=CRITICAL
publicSeverity=BLOCKER

[...]
```

Voici à présent l'ensemble des paramètres par défaut des règles embarquées dans *APIWatch*. Toutes les règles ont un paramètre en commun : `enabled`. Si ce paramètre est égal à `false`, la règle sera ignorée pour le calcul des violations de stabilité d'*API*.

ElementRemoval `[REM001]`

Suppression d'un élément d'*API*.

Paramètre	*Type*	*Valeur par défaut*	*Description*
`privateSeverity`	**`Severity`**	`INFO`	Niveau de risque des violations émises lors d'une suppression d'un élément de visibilité `PRIVATE`.
`scopeSeverity`	**`Severity`**	`CRITICAL`	Niveau de risque des violations émises lors d'une suppression d'un élément de visibilité `SCOPE`.
`protectedSeverity`	**`Severity`**	`CRITICAL`	Niveau de risque des violations émises lors d'une suppression d'un élément de visibilité `PROTECTED`.
`publicSeverity`	**`Severity`**	`BLOCKER`	Niveau de risque des violations émises lors d'une suppression d'un élément de visibilité `PUBLIC`.

Table 27: Propriétés configurables de la règle de stabilité d'API : `ElementRemoval`

ElementAddition `[ADD001]`

Ajout d'un élément d'*API*.

Paramètre	*Type*	*Valeur par défaut*	*Description*
`privateSeverity`	**`Severity`**	`INFO`	Niveau de risque des violations émises lors d'un ajout d'un élément de visibilité `PRIVATE`.
`scopeSeverity`	**`Severity`**	`INFO`	Niveau de risque des violations émises lors d'un ajout d'un élément de visibilité `SCOPE`.
`protectedSeverity`	**`Severity`**	`INFO`	Niveau de risque des violations émises lors d'un ajout d'un élément de visibilité `PROTECTED`.
`publicSeverity`	**`Severity`**	`INFO`	Niveau de risque des violations émises lors d'un ajout d'un élément de visibilité `PUBLIC`.

Table 28: Propriétés configurables de la règle de stabilité d'API : `ElementAddition`

ReducedVisibility `[VIS001]`

Réduction de la visibilité d'un élément d'*API*.

Paramètre	*Type*	*Valeur par défaut*	*Description*
`severity`	**`Severity`**	`CRITICAL`	Niveau de risque des violations émises lors d'une réduction de niveau de visibilité.

Table 29: Propriétés configurables de la règle de stabilité d'API : `ReducedVisibility`

DependenciesChange `[DEP001]`

Paramètre	*Type*	*Valeur par défaut*	*Description*
`addPrivateSeverity`	**`Severity`**	`INFO`	Niveau de risque des violations émises lors d'un ajout de nouvelles dépendances à un `APIScope` de visibilité `PRIVATE`.
`addScopeSeverity`	**`Severity`**	`MAJOR`	Niveau de risque des violations émises lors d'un ajout de nouvelles dépendances à un `APIScope` de visibilité `SCOPE`.
`addProtectedSeverity`	**`Severity`**	`MAJOR`	Niveau de risque des violations émises lors d'un ajout de nouvelles dépendances à un `APIScope` de visibilité `PROTECTED`.
`addPublicSeverity`	**`Severity`**	`MAJOR`	Niveau de risque des violations émises lors d'un ajout de nouvelles dépendances à un `APIScope` de visibilité `PUBLIC`.
`remPrivateSeverity`	**`Severity`**	`INFO`	Niveau de risque des violations émises lors d'une suppression de dépendances à un `APIScope` de visibilité `PRIVATE`.
`remScopeSeverity`	**`Severity`**	`INFO`	Niveau de risque des violations émises lors d'une suppression de dépendances à un `APIScope` de visibilité `SCOPE`.
`remProtectedSeverity`	**`Severity`**	`INFO`	Niveau de risque des violations émises lors d'une suppression de dépendances à un `APIScope` de visibilité `PROTECTED`.
`remPublicSeverity`	**`Severity`**	`INFO`	Niveau de risque des violations émises lors d'une suppression de dépendances à un `APIScope` de visibilité `PUBLIC`.

Table 30: Propriétés configurables de la règle de stabilité d'API : `DependenciesChange`

FunctionTypeChange `[TYP002]`

Changement du type de retour d'une fonction.

Paramètre	*Type*	*Valeur par défaut*	*Description*
`privateSeverity`	**`Severity`**	`INFO`	Niveau de risque des violations émises lors d'un changement de type de retour sur une fonction de visibilité `PRIVATE`.
`scopeSeverity`	**`Severity`**	`CRITICAL`	Niveau de risque des violations émises lors d'un changement de type de retour sur une fonction de visibilité `SCOPE`.
`protectedSeverity`	**`Severity`**	`CRITICAL`	Niveau de risque des violations émises lors d'un changement de type de retour sur une fonction de visibilité `PROTECTED`.
`publicSeverity`	**`Severity`**	`BLOCKER`	Niveau de risque des violations émises lors d'un changement de type de retour sur une fonction de visibilité `PUBLIC`.

Table 31: Propriétés configurables de la règle de stabilité d'API : `FunctionTypeChange`

ModifiersChange `[MOD001]`

Modification de l'attribut `modifiers` d'un symbole.

Paramètre	*Type*	*Valeur par défaut*	*Description*
`privateSeverity`	**`Severity`**	`INFO`	Niveau de risque des violations émises lors d'un changement de « modifiers » sur un symbole de visibilité `PRIVATE`.
`scopeSeverity`	**`Severity`**	`MAJOR`	Niveau de risque des violations émises lors d'un changement de « modifiers » sur un symbole de visibilité `SCOPE`.
`protectedSeverity`	**`Severity`**	`MAJOR`	Niveau de risque des violations émises lors d'un changement de « modifiers » sur un symbole de visibilité `PROTECTED`.
`publicSeverity`	**`Severity`**	`MAJOR`	Niveau de risque des violations émises lors d'un changement de « modifiers » sur un symbole de visibilité `PUBLIC`.

Table 32: Propriétés configurables de la règle de stabilité d'API : `ModifiersChange`

SuperTypesChange `[TYP003]`

Modification des `superTypes` d'un type complexe.

Paramètre	*Type*	*Valeur par défaut*	*Description*
`privateSeverity`	**`Severity`**	`INFO`	Niveau de risque des violations émises lors d'un changement de `superTypes` sur un `ComplexType` de visibilité `PRIVATE`.
`scopeSeverity`	**`Severity`**	`MAJOR`	Niveau de risque des violations émises lors d'un changement de `superTypes` sur un `ComplexType` de visibilité `SCOPE`.
`protectedSeverity`	**`Severity`**	`MAJOR`	Niveau de risque des violations émises lors d'un changement de `superTypes` sur un `ComplexType` de visibilité `PROTECTED`.
`publicSeverity`	**`Severity`**	`CRITICAL`	Niveau de risque des violations émises lors d'un changement de `superTypes` sur un `ComplexType` de visibilité `PUBLIC`.

Table 33: Propriétés configurables de la règle de stabilité d'API : `SuperTypesChange`

VariableTypeChange `[TYP001]`

Changement du type d'une variable.

Paramètre	*Type*	*Valeur par défaut*	*Description*
`privateSeverity`	**`Severity`**	`INFO`	Niveau de risque des violations émises lors d'un changement de type d'une variable de visibilité `PRIVATE`.
`scopeSeverity`	**`Severity`**	`CRITICAL`	Niveau de risque des violations émises lors d'un changement de type d'une variable de visibilité `SCOPE`.
`protectedSeverity`	**`Severity`**	`CRITICAL`	Niveau de risque des violations émises lors d'un changement de type d'une variable de visibilité `PROTECTED`.
`publicSeverity`	**`Severity`**	`BLOCKER`	Niveau de risque des violations émises lors d'un changement de type d'une variable de visibilité `PUBLIC`.

Table 34: Propriétés configurables de la règle de stabilité d'API : `VariableTypeChange`

Bibliographie et références

[1] Wilhelm Hasselbring. *Component-Based Software Engineering*, 2000.

[2] Terrence Parr. *Language Implementation Patterns*, 2009.

[3] Martin Fowler. *Continuous Integration*, http://martinfowler.com/articles/continuousIntegration.html

[4] Steve Oualline. *Practical C Programming*, 1997.

[5] Terrence Parr. *The Definitive ANTLR Reference*, 2007.

[6] Kuo-Chung Tai. *The Tree-To-Tree correction problem* in Journal of the ACM, Volume 26 Issue 3, july 1979.

[7] Andy Tripp. *Manual Tree Walking Is Better Than Tree Grammars*, 2006. http://www.antlr.org/article/1170602723163/treewalkers.html

[8] Ed Post. *Real Programmers don't use Pascal* http://www.pbm.com/~lindahl/real.programmers.html