UPEC - HYBRIGENICS SERVICES SAS

Maintenance évolutive d'un environnement Java

Rapport de stage

Stagiaire : Alice Pottier (IUT de Sénart/Fontainebleau)
Tuteur : Denis Monnerat (IUT de Sénart/Fontainebleau)
Maître de Stage : Thomas Moncion (Hybrigenics SAS)

31 Mars 2014 – 06 Juin 2014

Table des matières

1. Introduction

Lors de mes études à l'IUT de Sénart-Fontainebleau, j'ai découvert la programmation objet par l'intermédiaire du langage Java et l'une de mes résolutions était d'aller plus loin dans la pratique de ce langage. Cet apprentissage ne s'effectue pas obligatoirement par le développement de nouveaux projets. Il est également possible d'approfondir la connaissance d'un langage par la réalisation de travaux de maintenance. Ces travaux permettent de mieux cerner, dans un sens, les bonnes pratiques concernant le développement de programmes et la mise en place de ceux-ci dans un système en production.

Hybrigenics est une société de biotechnologie dont les plates-formes bio-informatiques reposent en grande partie sur l'utilisation de la technologie Java. La société a été fondée en 1999 c'est-à-dire seulement quatre années après la sortie de la première version de Java. Les versions de Java ont depuis été régulièrement mises à jour. Pour autant, le code développé à Hybrigenics n'a pas suivi les évolutions de ce langage.

A la fois pour consolider les plates-formes actuellement utilisées en production mais également pour permettre le développement de nouveaux projets, la société Hybrigenics procède à des travaux de maintenance de ses plates-formes.

C'est dans ce cadre que j'ai choisi le stage de 2ème année d'IUT au sein de la société Hybrigenics. L'objectif personnel du stage était de découvrir l'univers de l'informatique au sein d'une entreprise tout en améliorant mes connaissances de la technologie Java par l'intermédiaire de ces travaux de maintenance.

2. Présentation de l'entreprise

2.1. La société Hybrigenics

Hybrigenics est une société biotechnologique fondée en 1999 avec pour objectif de fournir des services liés aux techniques de double hybride (recherche des interactions protéine-protéine). Elle est actuellement composée de deux filiales distinctes :

1. **Hybrigenics SA** qui travaille sur l'étude et le développement de l'inecalcitol[1] dans la recherche contre le cancer de la prostate
2. **Hybrigenics SAS** qui fournit les services principalement liés aux techniques du double-hybride

Les deux filiales sont actuellement localisées dans le 14ème arrondissement de Paris. La clientèle de la filiale Hybrigenics SAS se situe, pour les deux tiers, en France et aux Etats-unis et concerne à la fois les secteurs académiques et privés. Le chiffre d'affaire de la société se situe à hauteur de 5 millions d'euros pour l'exercice de l'année 2013.

La filiale Hybrigenics SAS emploie actuellement 39 personnes réparties principalement en quatre grandes composantes :

1. Le service commercial
2. Le conseil scientifique
3. Les opérations industrielles
4. Le service bio-informatique/IT

[1] Dérivé de la vitamine D

Ces quatre composantes forment une chaîne permettant l'accompagnement du client depuis les premiers contacts jusqu'à la livraison des résultats (voir Figure 1). Une fois un projet bien défini par le client avec le service commercial, ce dernier est alors pris en charge par un membre du conseil scientifique qui va alors cerner les différentes étapes du projet. Les opérations industrielles vont alors procéder à la mise en œuvre des protocoles de recherche des interactions protéine-protéine. Toutes les étapes, depuis la gestion des clients jusqu'à la livraison des résultats, sont gérées par l'intermédiaire des plates-formes informatiques suivantes :

- **CRM (Customer Relationship Management)** : plate-forme permettant d'optimiser la gestion des clients et fournissant également des outils de marketing
- **PIMBuilder** : LIMS (Laboratory Information Management System) permettant la gestion de l'ensemble des opérations de criblage[2] ainsi que la traçabilité de ses résultats.
- **BSOL** : plate-forme contenant un ensemble d'outils bio-informatiques nécessaires à la conduite des projets
- **PIMRider** : application web permettant la visualisation des interactions protéine-protéine fournies par les cribles double-hybride
- **Discover Hit** : LIMS permettant la gestion de composés chimiques

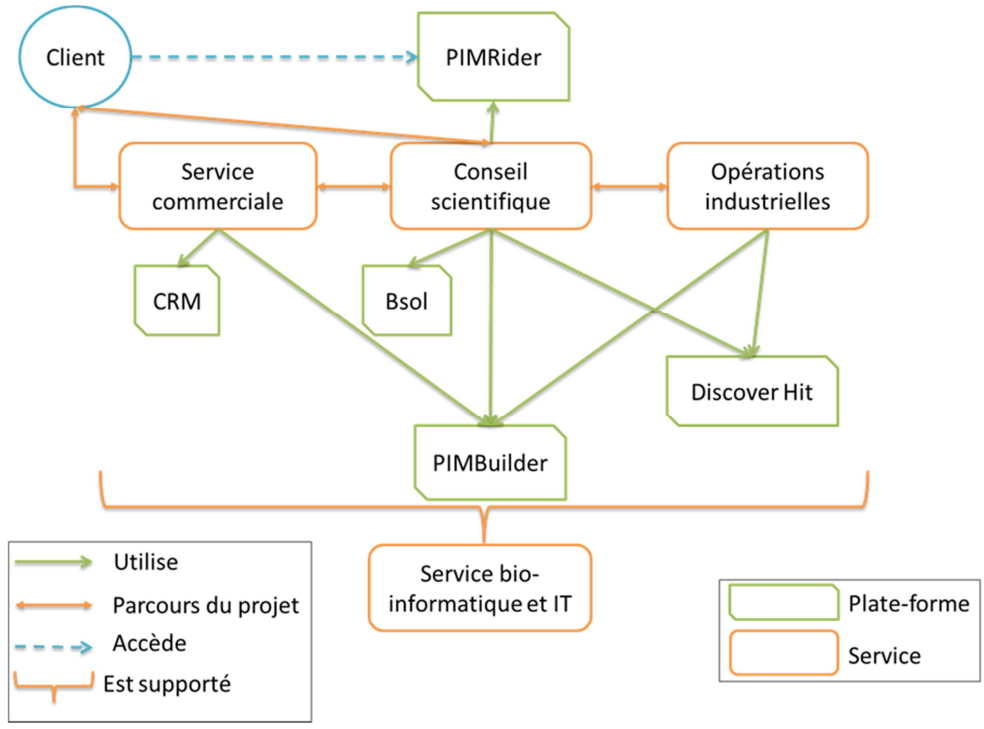


Figure 1 : Utilisation des plates-formes par les différents services de l'entreprise

[2] Procédures permettant de réaliser la recherche des interactions protéine-protéine

2.2. L'environnement de travail

Le stage s'est effectué au sein du service bio-informatique/IT de la filiale Hybrigenics SAS. Ce service est dirigé par Mr Collura (bio-informaticien) et est composé des membres suivants :

- deux bio-informaticiens (Mr El Yousfi et Mr Moncion)
- un administrateur systèmes (Mr Venne)
- un administrateur de base de données (Mme Sahli)

Au cours de ce stage était également présents un apprenti administrateur systèmes (Mr Quillerou) et une stagiaire de Master pour la qualité (Mme Wlaalj).

L'ensemble des plates-formes ainsi que les données des utilisateurs reposent sur une infrastructure virtualisée composée de serveurs UNIX (Distribution Debian). Le personnel, hormis les informaticiens, utilise des postes clients sous Windows. Les membres de l'équipe bio-informatique/IT utilisent des distributions Debian (Linux) possédant des machines virtuelles sous Windows. Le stage s'est donc effectué sous une distribution Debian.

3. Sujet

3.1. Contexte Général

La société Hybrigenics procède actuellement à des opérations de maintenance aussi bien sur les serveurs informatiques que sur les plates-formes bio-informatiques. Ces dernières sont des applications web basées sur la technologie Java EE et reposant sur une architecture trois-tiers (voir Figure 2).

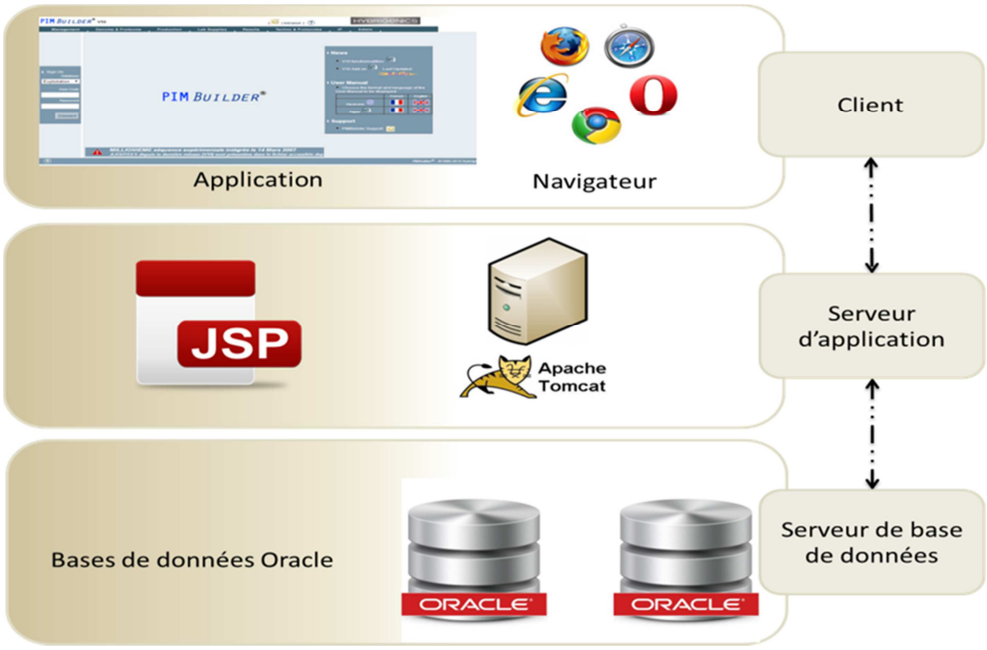


Figure 2: Schéma d'une architecture trois-tiers

En dehors du code développé au sein d'Hybrigenics, ces plates-formes exploitent également plusieurs librairies et/ou frameworks externes (Castor, Jibx, Batik, ...).

Les principaux développements, constituant le cœur des applications, ont été effectués entre 1999 et 2004. Depuis, un grand nombre de composants ont été ajoutés et des améliorations bio-informatiques ont été apportées mais les développements réalisés au début de la société n'ont pas été maintenus (classes Java non modifiées, librairies[3] non mises à jour, …). L'augmentation du nombre de warnings dans le code développé par Hybrigenics au fur et à mesure des versions Java est ainsi assez significative :

SDK Java SE d'Oracle	Nombre de warnings
Version 5 (mise à jour 22)	23678
Version 6 (mise à jour 45)	29700
Version 7 (mise à jour 55)	80745
Version 8 (beta)	80986

La maintenance des plates-formes informatiques est tout aussi fondamentale que le développement de nouveaux projets. Il est important, notamment, d'anticiper au maximum les problèmes pouvant survenir suite à la mise à jour de telle ou telle librairie (la présence de méthodes et/ou de classes de type « deprecated », par exemple, nécessite alors la consolidation des plates-formes en fonction des nouvelles spécifications).

Pour montrer l'importance de cette maintenance, nous pouvons prendre l'exemple de la mise à jour d'un composant chemo-informatique (Chemaxon) au sein de la plate-forme Discover Hit. Cette mise à jour était nécessaire et obligeait également celle de la librairie Xerces (l'ancienne version n'étant pas acceptée par Chemaxon). Cette librairie étant également utilisée sur d'autres plates-formes, une étude des problèmes pouvant survenir était alors nécessaire avant de mettre à jour Chemaxon.

La société envisage également le développement de nouveaux projets basés sur les dernières versions des plates-formes Java SE et Java EE. Avant d'entamer ces nouveaux projets, il est impératif de déterminer si les plates-formes actuellement utilisées en production supportent ces dernières versions.

3.2. Contexte au début du stage

En cours de stage, une mise à jour de la version de la plate-forme Java SE a été entreprise afin de passer de la version 6 du SDK d'Oracle à la version 7 de la plate-forme open-source OpenJDK. Les différentes erreurs de compilation liées à la mise à jour de Java avaient été effectuées par les développeurs et seules les erreurs de type « warning » étaient encore présentes.

La construction des applications web est actuellement réalisée par l'intermédiaire de l'outil Ant et repose sur des fichiers sources présents au sein d'un dépôt CVS. Une migration vers le logiciel de gestion de versions Subversion venait d'être réalisée avant le début du stage. Cette migration a nécessité une refonte de l'architecture des environnements ainsi que des fichiers de construction utilisés par Ant. Cette migration était fonctionnelle mais pas encore basculée en production[4].

Dans ce contexte, le stage s'est effectué en utilisant un dépôt Subversion installé localement sur une machine ainsi que les nouveaux fichiers Ant issus de la refonte de

[3] Ensemble de classes Java qui offre un grand nombre de fonctionnalités (manipulation d'image, envoies d'e-mails,…)

[4] Utilisé par les autres services que la bio-informatique

l'architecture des environnements. De cette manière, il était possible d'effectuer toutes les modifications possibles tout en ayant la possibilité de revenir en arrière en cas de problèmes.

3.3. Objectifs

C'est dans ce contexte qu'un stage permettant de réaliser l'étude de la mise à jour de Java et des librairies a été proposé au sein de la société Hybrigenics. Il était important, pour les développeurs, de déterminer les conséquences de ces mises à jour afin d'anticiper les éventuels travaux de refonte que cela pouvait impliquer.

Un autre volet du stage concernait la recherche de codes inutilisés au sein des plates-formes (code dit « mort »). Les programmes se sont accumulés au cours du temps tout en n'étant jamais supprimés lorsqu'ils devenaient obsolètes.

Les objectifs de ce stage étaient donc multiples :

- Déterminer les conséquences de la mise à jour des plates-formes sous OpenJDK 7 (étude et correction des warnings)
- Effectuer la mise à jour des librairies et en étudier les conséquences au sein du code d'Hybrigenics
- Déterminer les zones de code désormais inutilisées

3.3.1. Etude et correction des warnings

Comme cela a été expliqué précédemment, le nombre de warnings présents au sein du code d'Hybrigenics ne fait que s'accroitre au fur et à mesure du renouvellement des versions de Java. Cet accroissement permanent des warnings incite donc à effectuer une étude de ces derniers afin de déterminer s'ils peuvent poser des problèmes ultérieurement.

Le premier objectif était donc de catégoriser l'ensemble des warnings. Cette catégorisation devait être réalisée à deux niveaux c'est-à-dire par :

1. ***type de warning*** : lorsqu'il existe 80745 warnings, il est important de savoir s'ils sont tous d'un même type ou s'ils sont de types très différents. Dans le premier cas, les corrections peuvent être plus simples. Cela permet également d'ordonner les warnings par importance. En effet, les warnings de type *deprecated*[5] sont plus « dangereux » que des warnings de type *rawtype*.
2. ***package concerné*** : certains packages sont plus importants que d'autres pour le fonctionnement des plates-formes. Il est donc important de déterminer où se situent les warnings au sein des différents packages

Une fois la première étape de catégorisation des warnings effectuée, l'objectif suivant est la correction, dans la mesure du possible, de ces warnings.

3.3.2. Etude et mise à jour des librairies

L'ajout de librairies externes permet d'utiliser, de manière simple, des solutions déjà existantes. Pour autant, l'incorporation de ces librairies implique certaines conséquences au niveau des plates-formes. Il est en effet important :

- de vérifier les dépendances entre les librairies ; les librairies externes utilisent elles même d'autres librairies pour leur fonctionnement. Il est donc fondamental de déterminer la cohérence de l'ensemble des librairies lorsqu'elles sont ajoutées au sein d'une plate-forme

[5] Méthode qui existe toujours par soucis de compatibilité mais qu'il n'est plus conseillé d'utiliser

- de vérifier si les mises à jour des librairies sont toujours en adéquation avec leur utilisation au sein des plates-formes (la présence de méthodes dépréciées, par exemple, peut alors nécessiter une refonte partielle du code)

Ce deuxième volet des objectifs doit se dérouler en deux phases. La première consiste à déterminer les dépendances entre les librairies ainsi que leur utilisation au sein des plates-formes d'Hybrigenics. Pour cela, l'utilisation d'un outil effectuant ce travail d'analyse, est important. Cette première phase peut également être mise à profit pour déterminer si des librairies présentes au sein des plates-formes sont encore utilisées. La deuxième phase consiste à rechercher si les librairies utilisées à Hybrigenics sont maintenues ou non. Dans le cas où elles le sont, il est alors nécessaire de déterminer les conséquences de la mise en place des nouvelles versions au sein du code d'Hybrigenics.

3.3.3. Recherche et suppression du code non utilisé

De nombreux programmes développés à Hybrigenics sont désormais obsolètes. Une volonté d'Hybrigenics est de se recentrer sur le code fonctionnel et donc de supprimer les programmes inutiles. Ces derniers utilisent parfois des librairies et présentent également un certain nombre de warnings. Il est donc plus intéressant de simplement supprimer ces programmes plutôt que de corriger les warnings ou de mettre à jour les librairies.

Dans ce contexte, un autre volet du stage, plus ou moins indépendant de ceux présentés précédemment, était donc de rechercher les codes dits « morts » au sein de l'ensemble des classes Java.

Cette recherche du code mort ne peut s'effectuer que par l'intermédiaire d'un outil réalisant une analyse automatique du code. Une première phase consiste donc à comparer les différents outils existants, la deuxième étant alors d'utiliser l'outil sélectionné pour rechercher d'éventuels codes morts.

4. Déroulement du stage

4.1. Familiarisation avec les outils et les plates-formes

Une première période d'apprentissage des outils Subversion (logiciel de gestion de versions) et Ant (logiciel d'automatisation des tâches) ainsi que des environnements développés par Hybrigenics fut nécessaire.

Les fichiers sources des différents environnements étaient présents au sein d'un dépôt Subversion local. Ce dernier contenait deux types de projets (voir Figure 3) :

1. **le projet javaland** qui contient l'ensemble des classes Java et des librairies utilisées par Hybrigenics
2. **les projets pimbuilder, bsol, hitdb et pimrider** qui contiennent les fichiers sources des applications web. Les classes Java et les librairies nécessaires au fonctionnement de ces applications sont récupérées à partir de la javaland par l'intermédiaire des références externes permises par Subversion.

La centralisation des classes et des librairies au sein d'un unique projet « *javaland* » permet d'éviter la duplication de ces dernières au sein des différents environnements web. Chaque projet possède des fichiers de construction, utilisés par Ant, réalisant l'installation de l'application.

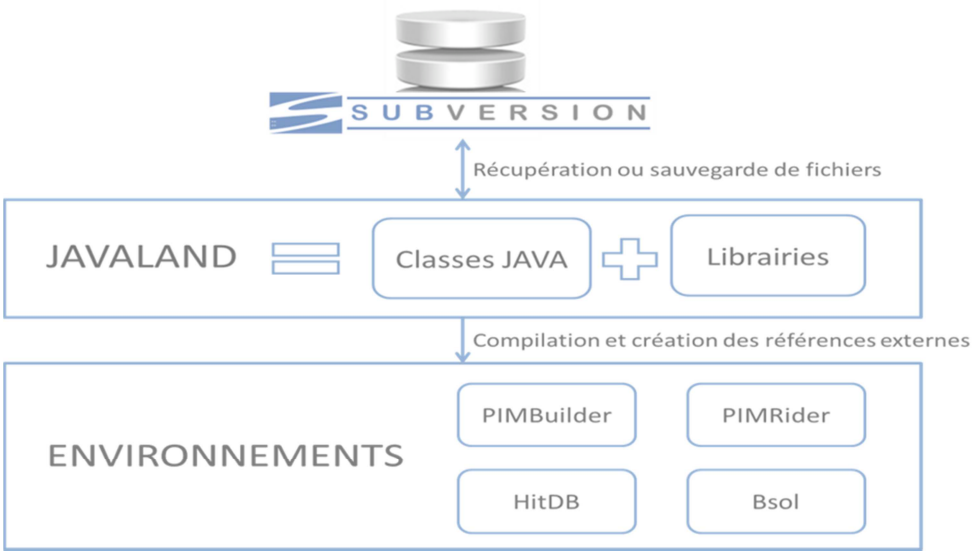


Figure 3 : Organisation des environnements d'Hybrigenics

L'apprentissage de l'outil Ant était donc nécessaire. C'est en effet par cet outil que l'étude des warnings fut possible. Les connaissances liées au fonctionnement d'un dépôt Subversion ainsi que les commandes permettant de basculer les modifications étaient également nécessaires.

4.2. Etude des warnings

Les premiers travaux effectués au cours du stage furent donc l'étude et la catégorisation des warnings affichés lors de la compilation des classes Java. Cette étude est nécessaire pour déterminer l'état des classes Java et les possibles refontes que cela peut impliquer. C'est également un préambule nécessaire à l'étude de la mise à jour des librairies.

4.2.1. Méthodologie

Lors de la construction des environnements par Ant, les warnings sont caractérisés lors du lancement de la cible « compile » qui effectue la compilation des classes Java. L'affichage des warnings dépend alors de la valeur de l'option « -Xlint » ainsi de la valeur maximale du nombre de warnings à afficher « -Xmaxwarns ».

Le premier encadré (voir Figure 4) correspond au fichier build.xml qui contient la déclaration de toutes les tâches (ou cible) que Ant doit effectuer et le second correspond au fichier build.properties qui contient les propriétés utiles à chacune d'entre elle.

```
<!--=================================================================-->
<!-- Compilation of java classes                                     -->
<!--=================================================================-->
<target name="compile" description="Compile java classes">
    <echo>Compiling java classes</echo>

    <!-- Compilation of classes in src directory -->
    <javac srcdir="${dir.src}"
           destdir="${dir.compile}"
           source="${javac.source}"
           target="${javac.target}"
           debug="${javac.debug}"
           verbose="${javac.verbose}"
           includeAntRuntime="${javac.includeAntRuntime}">
        <classpath><path refid="classpath.path"/></classpath>
        <compilerarg value="${javac.compilerarg.xlint}"/>
        <compilerarg line="${javac.compilearg.xmaxerrs}"/>
        <compilerarg line="${javac.compilearg.xmaxwarns}"/>
    </javac>
</target>
```

```
#================================================
# Java compilation options
#================================================
# Source version (specifications of source code version)
javac.source=1.7
# Target version (specifications of generated bytecode)
javac.target=1.7
# Debug mode
javac.debug=off
# Verbose output
javac.verbose=false
# Use or not environment
javac.includeAntRuntime=false
# Xlint option for compiler arguments
javac.compilerarg.xlint=-Xlint:all
# Xmaxerrs option for compiler arguments
javac.compilearg.xmaxerrs=-Xmaxerrs 99999
# Xmaxwarns option for compiler arguments
javac.compilearg.xmaxwarns=-Xmaxwarns 99999
```

Affichage des warnings : all/none

Nombre d'erreurs à afficher

Nombre de warnings à afficher

Figure 4 : Rapport entre la cible et les options de Ant

La sortie des résultats est donc une longue liste de warnings au fur et à mesure des classes compilées par Ant (voir Figure 5). Cette sortie ne fournit pas une bonne visibilité sur les différentes catégories de warnings rencontrées.

```
[javac]   missing type arguments for generic class Hashtable<K,V>
    [javac]   where K,V are type-variables:
    [javac]     K extends Object declared in class Hashtable
    [javac]     V extends Object declared in class Hashtable
    [javac]
/home/stagiaire/home_apottier/Environnements/javaland/src/hgx/annotator/annotatorTools.java:19
4: warning: [rawtypes] found raw type: Hashtable
    [javac]                   Hashtable hnameseqL = new Hashtable();
    [javac]                   ^
    [javac]   missing type arguments for generic class Hashtable<K,V>
    [javac]   where K,V are type-variables:
    [javac]     K extends Object declared in class Hashtable
    [javac]     V extends Object declared in class Hashtable
    [javac]
/home/stagiaire/home_apottier/Environnements/javaland/src/hgx/annotator/annotatorTools.java:19
4: warning: [rawtypes] found raw type: Hashtable
    [javac]                   Hashtable hnameseqL = new Hashtable();
    [javac]                                         ^
    [javac]   missing type arguments for generic class Hashtable<K,V>
    [javac]   where K,V are type-variables:
    [javac]     K extends Object declared in class Hashtable
    [javac]     V extends Object declared in class Hashtable
```

```
    [javac]
/home/stagiaire/home_apottier/Environnements/javaland/src/hgx/annotator/annotatorTools.java:19
5: warning: [rawtypes] found raw type: Hashtable
    [javac]                   Hashtable hnameseql = new Hashtable();
    [javac]                   ^
    [javac]   missing type arguments for generic class Hashtable<K,V>
    [javac]   where K,V are type-variables:
    [javac]     K extends Object declared in class Hashtable
    [javac]     V extends Object declared in class Hashtable
    [javac]
/home/stagiaire/home_apottier/Environnements/javaland/src/hgx/annotator/annotatorTools.java:19
5: warning: [rawtypes] found raw type: Hashtable
    [javac]                   Hashtable hnameseql = new Hashtable();
```

Figure 5 : Extrait d'une sortie de Ant

Pour avoir une meilleure visibilité sur les warnings, il était important de catégoriser ces derniers en fonction de leur type mais également en fonction des packages concernés. Un travail de ce type est réalisable par l'intermédiaire d'un script écrit en Bash. Il est ainsi possible d'utiliser les différents outils fournis par les systèmes UNIX comme *sed*, *grep* ou encore *cut*. Ce script permet donc de prendre en paramètre les sorties fournies par Ant et de fournir en sortie les fichiers permettant une visualisation des warnings par type (voir Figure 6) et permet également de montrer les différents types de warnings au sein de chaque package (voir Figure 7).

```
deprecation:21
      1  buildSub_BPsHierarchy(Vector,Vector,boolean) in BP has been deprecated
     20  getCharacterOutputStream() in CLOB has been deprecated

rawtypes:30835
    496  found raw type: Enumeration
   5128  found raw type: Hashtable
  25211  found raw type: Vector

serial:304
      1  serializable class ACT_ACTROLE_RO has no definition of serialVersionUID
      1  serializable class ACT_G has no definition of serialVersionUID
      1  serializable class ACT_LINKTO has no definition of serialVersionUID
      1  serializable class ACTROLE_G has no definition of serialVersionUID
      1  serializable class BAITCLNG_G has no definition of serialVersionUID
      1  serializable class BAITCLONEPICK_G has no definition of serialVersionUID
      1  serializable class BAITTOXBA_G has no definition of serialVersionUID
      1  serializable class BAITTOXDP_G has no definition of serialVersionUID
      1  serializable class BAITTOX_G has no definition of serialVersionUID
      1  serializable class BAITVALIDDP_G has no definition of serialVersionUID
      1  serializable class BAITVALID_G has no definition of serialVersionUID
      1  serializable class BCLUSTER_BO has no definition of serialVersionUID
      1  serializable class BCLUSTER_G has no definition of serialVersionUID
      1  serializable class BCLUSTERNAT_G has no definition of serialVersionUID
      1  serializable class BDOMAIN_G has no definition of serialVersionUID
      1  serializable class BIBLIO_BOANNOT has no definition of serialVersionUID
      1  serializable class BIBLIO_G has no definition of serialVersionUID
      1  serializable class BOANNOT_G has no definition of serialVersionUID
      1  serializable class BO_COMP_LOTBO has no definition of serialVersionUID
      1  serializable class BOCONSOTRACK_G has no definition of serialVersionUID
      1  serializable class BODISTRIB_G has no definition of serialVersionUID
      1  serializable class BO_DISTRIB has no definition of serialVersionUID
      1  serializable class BO_G has no definition of serialVersionUID
      1  serializable class BOPARAM_G has no definition of serialVersionUID
```

Figure 6 : Extrait des warnings triés par catégorie

```
annotator:18
     16 rawtypes
      2 unchecked
biobank:41
      1 cast
      5 rawtypes
      1 static
     34 unchecked
bioseqanalysis:332
     10 dep-ann
     28 deprecation
      1 fallthrough
    223 rawtypes
```

```
      5 static
     92 unchecked
blastident:27
      4 deprecation
     15 rawtypes
      8 unchecked
bsol:15
      1 cast
      8 deprecation
      3 rawtypes
      3 static
clustering:1431
     40 cast
      2 dep-ann
      4 deprecation
    954 rawtypes
      4 serial
      2 static
    472 unchecked
```

Figure 7 : Extrait des warnings triés par catégorie pour chaque package

4.2.2. Résultats

La catégorisation des 80745 warnings engendrés par la compilation a montré certaines informations intéressantes. La première information concerne la catégorisation par type de warning. Elle montre qu'une grande majorité des warnings est liée au fait que les classes n'utilisent pas les génériques (rawtypes) de Java. Cela s'explique en grande partie par le fait que cette spécification n'existait pas lors des premiers développements des plates-formes.

L'étude des warnings les plus importants à savoir ceux de type « deprecated » a montré qu'ils concernaient principalement des classes construites il y a environ une dizaine d'années par l'intermédiaire du framework Castor (framework permettant d'effectuer le mapping xml/objet). Les nouvelles versions de Java considèrent désormais les interfaces utilisées à l'époque (*org.xml.sax.DocumentHandler* par exemple) comme étant obsolètes. Une première conclusion de cette étude fut donc la nécessité dans un avenir plus ou moins proche de reconstruire ces classes avec la dernière version de Castor.

La catégorisation en fonction des packages concernés a également apportée des informations intéressantes. Cette dernière a montré que 71% des warnings concernaient le package *hgx.pimbuilder* (package central de l'application PIMBuilder). Nous atteignons même une valeur de 88% en considérant les packages *hgx.pimbuilder*, *hgx.pimrider* et *hgx.hitdb*. Ces packages ont une particularité à savoir qu'ils contiennent les classes permettant d'effectuer le mapping entre les objets et les bases de données relationnelles. Ces classes ont un rôle capital dans le fonctionnement de l'architecture trois-tiers utilisée par les plates-formes d'Hybrigenics. En regardant un peu plus en détail où sont situés les warnings au sein de ces packages, il apparait que 76% d'entre eux sont contenus dans ces classes (voir Figure 7). L'analyse de ces warnings a montré qu'une grande majorité était liée au fait que les génériques n'étaient pas utilisés au sein de ces classes.

Les classes effectuant le mapping entre les objets et les bases de données relationnelles ne sont pas construites manuellement mais sont produites de manière automatique par un programme développé en interne (classes présentes dans le package *hgx.generator*). Cette constatation a donc orienté la suite du stage. Il était en effet intéressant de déterminer s'il était possible de supprimer une grande partie des warnings en modifiant le générateur de code.

4.3. Modification du générateur de code

Actuellement, plusieurs frameworks proposent des outils permettant la construction automatique de classes effectuant le mapping objet/relationnel (Hibernate est un des plus

utilisés). A l'époque de la mise en place des plates-formes d'Hybrigenics, ces frameworks étaient en cours de construction. Un générateur interne à Hybrigenics a donc été développé pour construire de manière automatique les classes permettant le mapping objet/relationnel.

Cet outil permet d'adapter facilement le code aux modifications effectuées au sein des bases de données. Dès lors qu'un changement est réalisé au niveau d'une base de données, les classes sont construites de nouveau et remplacent les anciennes.

Comme l'a montré l'étude des warnings, les classes produites par le générateur contiennent un grand nombre de warnings principalement liés aux génériques de Java. En dehors de quelques ajustements, le cœur du générateur date du début des années 2000. A cette époque, les génériques n'étaient pas encore inclues dans les spécifications de la plate-forme Java. Il était alors intéressant de déterminer si le code du générateur pouvait être modifié afin de rajouter dans les classes produites les nouvelles spécifications liées à l'utilisation des génériques.

4.3.1. Méthodologie

La procédure de construction et de remplacement des classes effectuant le mapping objet/relationnel a été reprise non pas pour adapter le code aux modifications d'une base de données mais pour tenter de corriger les warnings présents dans ces classes.

La méthodologie à suivre est donc de regarder les différents warnings présents dans les classes produites et de déterminer si nous pouvons modifier le générateur afin de corriger ces warnings. Une fois les classes de nouveau construites, ces dernières remplacent celles présentes dans l'environnement *javaland*. Une nouvelle compilation de l'ensemble des classes est alors effectuée pour déterminer si les modifications réalisées ne posent pas de problèmes à la compilation et réduisent bien le nombre de warnings (voir Figure 8).

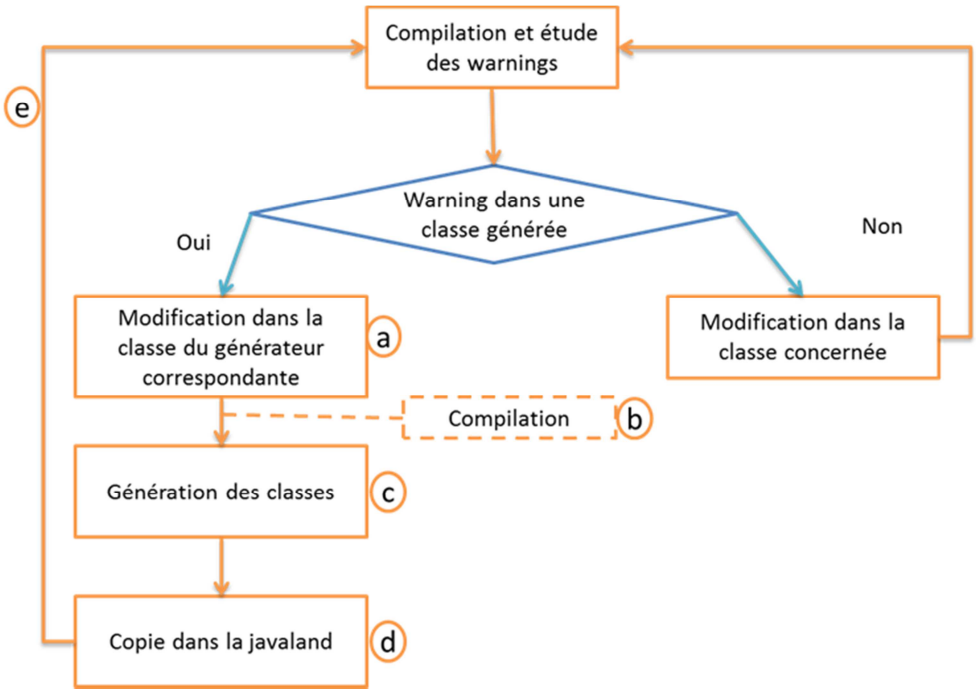


Figure 8 : Procédure de corrections dans le générateur

Les warnings liés aux génériques étant les plus nombreux, les premières corrections effectuées ont concerné ce type de warning.

4.3.2. Résultats

Les premières modifications ont permis de supprimer un nombre assez conséquent de warnings. 10490 warnings ont ainsi été éliminés en effectuant quelques changements au niveau du générateur de code.

Malheureusement, les modifications du générateur n'ont pas pu aller au-delà. Certains ajouts de génériques au niveau des classes générées ont montrés que ces dernières n'étaient alors plus viables. En effet, ces ajouts au niveau de certains attributs de type Vector ont montré que plusieurs catégories de classes, pourtant très différentes, étaient utilisés au sein de ces attributs.

Les warnings présents dans les classes générées ne peuvent donc être corrigées à moins d'effectuer une profonde refonte du générateur de code. Lors de l'étude des warnings, nous avons constaté que 67 % des warnings (54608 sur les 80745 au total) étaient liés aux classes produites par le générateur. Une solution envisagée par les développeurs dans le futur est alors de remplacer le générateur par une autre solution comme Hibernate par exemple.

4.4. Etude et mise à jour des librairies

Le projet *javaland* contient plus d'une centaine de librairies utilisées ou non par les différentes applications web. Ces librairies peuvent également avoir des fonctions en dehors des applications web (exécution de programmes sur des serveurs de calcul notamment). Les librairies peuvent être divisées en deux catégories, c'est-à-dire :

1. celles nécessaires à la compilation des classes
2. celles nécessaires à l'exécution des classes (la non présence de ces librairies n'empêche pas la compilation des classes mais peut provoquer des erreurs lors de leur exécution)

La plupart des librairies utilisées à Hybrigenics ont été ajoutées lors de la mise en place des plates-formes mais n'ont jamais été mises à jour depuis. Il est donc important de déterminer si ces librairies sont toujours maintenues et si les dernières versions sont toujours en adéquation avec leur utilisation au sein des plates-formes d'Hybrigenics.

L'étude des warnings avait pour objectif de déterminer les possibles refontes à réaliser au niveau du code d'Hybrigenics. Ce travail était également un préambule à l'étude de la mise à jour des librairies. Il est en effet important de caractériser les différences existantes au niveau de la compilation par la mise à jour de telle ou telle librairie.

4.4.1. Méthodologie

Avant d'entamer la campagne de mise à jour des librairies, il était important, dans un premier temps, de s'approprier l'utilité de chaque librairie au sein des plates-formes d'Hybrigenics. Ce travail doit permettre de déterminer, entre autres :

- les classes d'Hybrigenics impactées par telle ou telle mise à jour d'une librairie
- les dépendances entre les librairies
- les librairies encore utilisées

Ce travail est fortement facilité par l'utilisation d'un outil tel que Tattletale. Ce dernier, conçu par JBoss, fournit une vue complète des informations concernant les librairies d'un projet comme :

- les dépendances entre les classes

- les dépendances entre les librairies
- la présence de classes en doublon
- les librairies inutilisées

Tattletale utilise en entrée une liste de librairies et produit en sortie un rapport au format HTML (voir Figure 9).

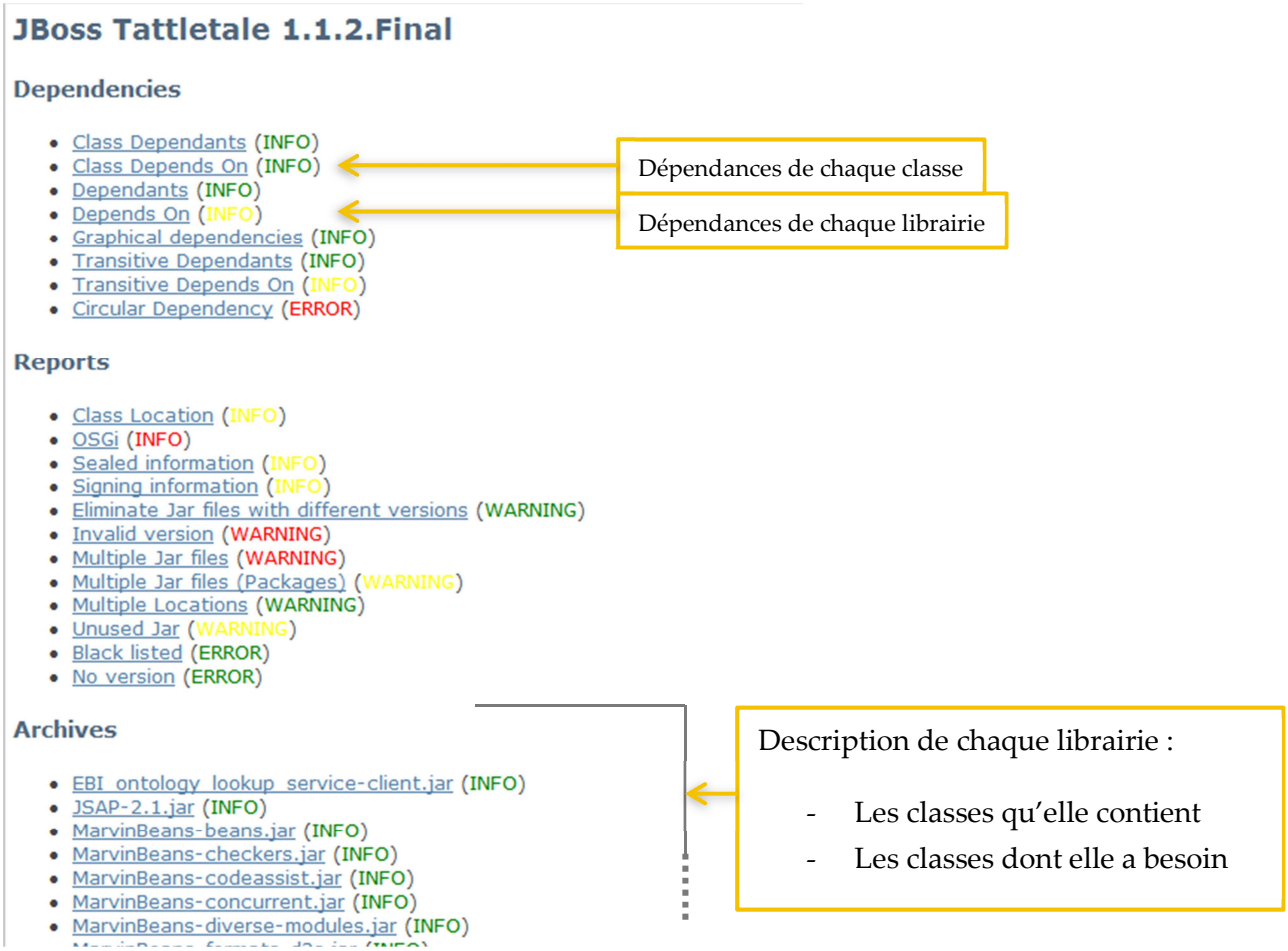


Figure 9 : Page d'accueil de Tattletale

L'outil Tattletale ne fonctionnant uniquement que par l'intermédiaire de librairies, il n'est pas possible d'utiliser directement les classes d'Hybrigenics. Il est donc nécessaire de construire une librairie *hgx.jar* contenant l'ensemble des classes d'Hybrigenics. Cette librairie est alors ajoutée aux autres librairies du projet javaland (voir Figure 10).

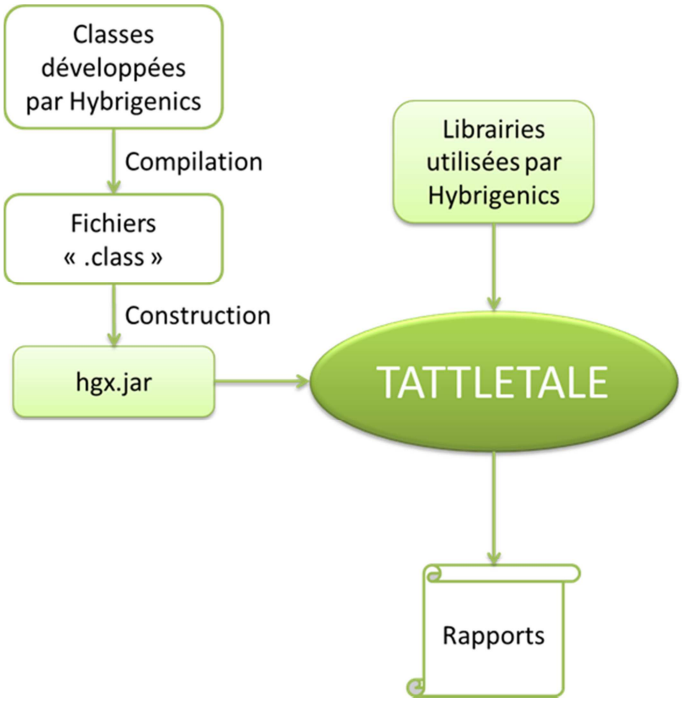


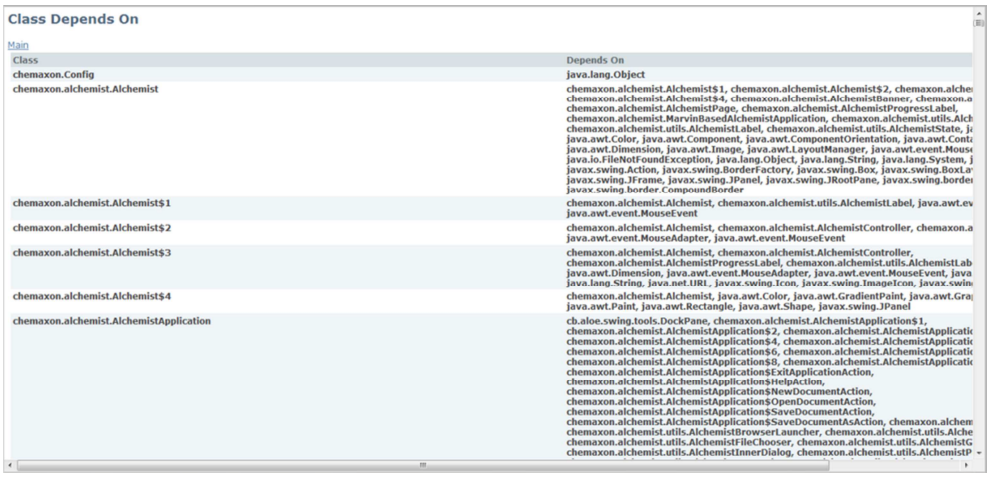
Figure 10 : Fonctionnement de Tattletale

La librairie *hgx.jar* peut être produite de manière automatique par Ant. Une cible Ant a donc été ajoutée dans le fichier build.xml afin de réaliser cette construction :

```
<target name="makeJar" description="Build hgx.jar">
    <echo>Building hgx.jar</echo>
    <delete dir="landJar"/>
    <mkdir dir="landJar"/>
    <jar destfile="landJar/hgx.jar" basedir="compile" includes="hgx/**"/>
</target>
```

Figure 11 : Cible permettant la construction du fichier hgx.jar

Un des points intéressants de Tattletale est le rapport concernant les dépendances des classes (voir Figure 9). Pour chacune des classes de l'ensemble des librairies, les dépendances à d'autres classes sont listées (voir Figure 12).


Class Depends On

Main

Class	Depends On
chemaxon.Config	java.lang.Object
chemaxon.alchemist.Alchemist	chemaxon.alchemist.Alchemist$1, chemaxon.alchemist.Alchemist$2, chemaxon.alchemist.Alchemist$4, chemaxon.alchemist.AlchemistBanner, chemaxon.alchemist.AlchemistPage, chemaxon.alchemist.AlchemistProgressLabel, chemaxon.alchemist.MarvinBasedAlchemistApplication, chemaxon.alchemist.utils.AlchemistLabel, chemaxon.alchemist.utils.AlchemistState, java.awt.Color, java.awt.Component, java.awt.ComponentOrientation, java.awt.Dimension, java.awt.Image, java.awt.LayoutManager, java.awt.event.MouseEvent, java.io.FileNotFoundException, java.lang.Object, java.lang.String, java.lang.System, javax.swing.Action, javax.swing.BorderFactory, javax.swing.Box, javax.swing.BoxLayout, javax.swing.JFrame, javax.swing.JPanel, javax.swing.JRootPane, javax.swing.border.CompoundBorder
chemaxon.alchemist.Alchemist$1	chemaxon.alchemist.Alchemist, chemaxon.alchemist.utils.AlchemistLabel, java.awt.event.MouseEvent
chemaxon.alchemist.Alchemist$2	chemaxon.alchemist.Alchemist, chemaxon.alchemist.AlchemistController, java.awt.event.MouseAdapter, java.awt.event.MouseEvent
chemaxon.alchemist.Alchemist$3	chemaxon.alchemist.Alchemist, chemaxon.alchemist.AlchemistController, chemaxon.alchemist.AlchemistProgressLabel, java.awt.Dimension, java.awt.event.MouseAdapter, java.awt.event.MouseEvent, java.lang.String, java.net.URL, javax.swing.Icon, javax.swing.ImageIcon,
chemaxon.alchemist.Alchemist$4	chemaxon.alchemist.Alchemist, java.awt.Color, java.awt.GradientPaint, java.awt.Paint, java.awt.Rectangle, java.awt.Shape, javax.swing.JPanel
chemaxon.alchemist.AlchemistApplication	cb.aloe.swing.tools.DockPane, chemaxon.alchemist.AlchemistApplication$1, chemaxon.alchemist.AlchemistApplication$2, chemaxon.alchemist.AlchemistApplication$4, chemaxon.alchemist.AlchemistApplication$6, chemaxon.alchemist.AlchemistApplication$8, chemaxon.alchemist.AlchemistApplication$ExitApplicationAction, chemaxon.alchemist.AlchemistApplication$HelpAction, chemaxon.alchemist.AlchemistApplication$NewDocumentAction, chemaxon.alchemist.AlchemistApplication$OpenDocumentAction, chemaxon.alchemist.AlchemistApplication$SaveDocumentAction, chemaxon.alchemist.AlchemistApplication$SaveDocumentAsAction, chemaxon.alchemist.utils.AlchemistBrowserLauncher, chemaxon.alchemist.utils.AlchemistFileChooser, chemaxon.alchemist.utils.AlchemistInnerDialog, chemaxon.alchemist.utils.AlchemistP

Figure 12 : Début du fichier contenant les dépendances de chaque classe

Etant donné le grand nombre de classes présentes au sein de toutes les librairies, cette page HTML est longue à charger. De plus, il est intéressant pour les développeurs de

connaître, au moins dans un premier temps, seulement les dépendances des classes développées à Hybrigenics.

Un premier travail a donc consisté à intégrer dans le rapport de Tattletale une nouvelle page centrée uniquement sur les dépendances des classes développées par Hybrigenics. Une autre amélioration a également consisté à fournir des liens vers les dépendances concernées (voir Figure 14). Il existe alors deux cas possibles :

1. La dépendance est une autre classe d'Hybrigenics. Dans ce cas, le lien pointe vers la classe concernée
2. La dépendance est une classe présente dans une librairie externe. Dans ce cas, le lien pointe vers les informations concernant la librairie

Pour réaliser cela, un script écrit en Bash a été utilisé afin de parser les différentes pages HTML fournies par Tattletale et ainsi construire une nouvelle page HTML (voir Figure 14). Cette dernière est alors intégrée au rapport général fourni par Tattletale (voir Figure 13).

Figure 13 : Nouvelle page d'accueil de Tattletale contenant un lien vers les dépendances spécifiques des classes d'Hybrigenics

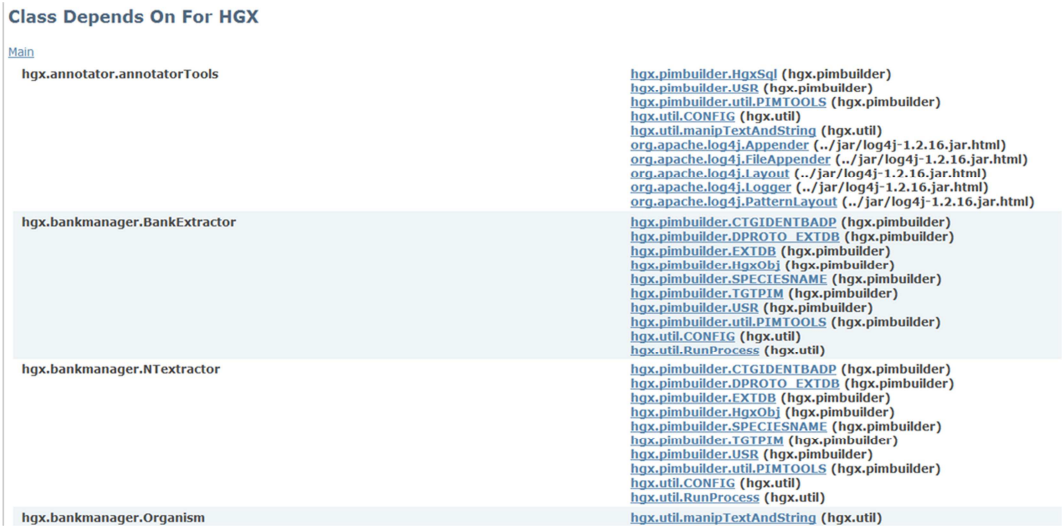


Figure 14 : Page HTML contenant les dépendances pour chaque classe d'Hybrigenics

Une fois ce travail de modification du rapport HTML de Tattletale effectué, la campagne de mise à jour des différentes librairies est alors possible. Cette campagne consiste, pour chaque librairie à se poser les questions suivantes :

1. La librairie est-elle encore utilisée au sein des plates-formes ?
2. La librairie est-elle encore maintenue ?
3. La mise à jour de la librairie engendre-t-elle des problèmes au niveau de la compilation ou des warnings ?

Le cheminement de ces questions peut être représenté par un diagramme de décision (voir Figure 15). Si une nouvelle version d'une librairie existe, celle-ci va remplacer l'ancienne version au sein de l'environnement *javaland*. Si la phase de compilation ne pose pas de problèmes particuliers, la phase d'étude des warnings est alors nécessaire. Il est en effet important de déterminer, par exemple, si des méthodes ou des classes utilisées ne sont pas désormais marquées comme étant dépréciées.

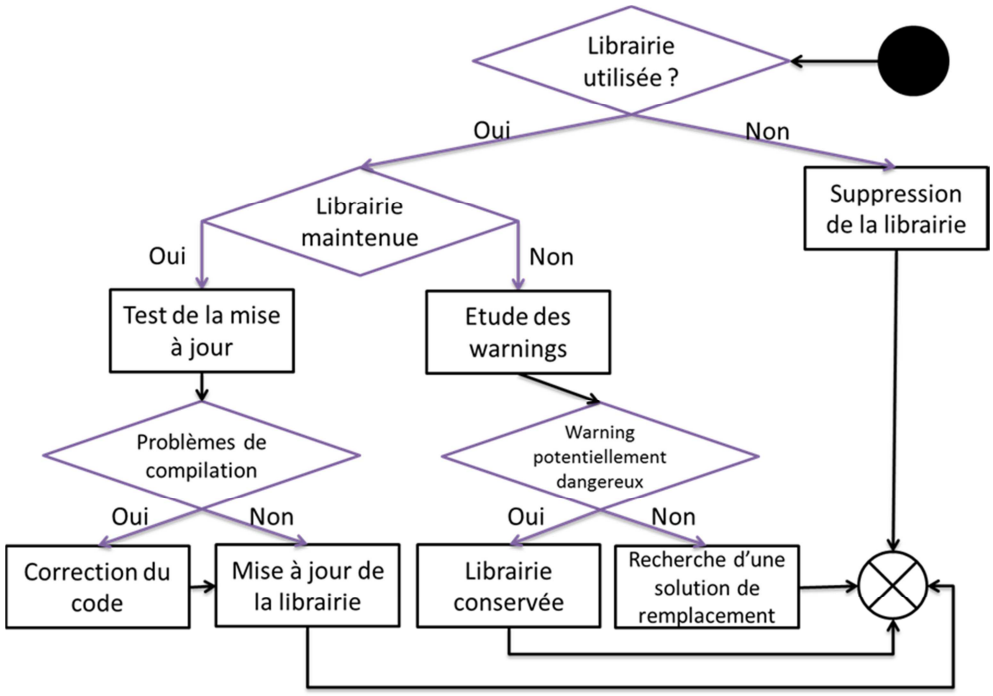


Figure 15 : Arbre de décision concernant la mise à jour des librairies

Cette procédure concerne l'étude des librairies au niveau des classes Java de l'environnement *javaland*. Certaines librairies peuvent également être utilisées par les applications web au niveau des fichiers JSP. Ces fichiers sont une abstraction « haut niveau » des servlets (celles-ci étant des classes Java). Une phase de pré-compilation des JSP est donc possible pour déterminer la cohérence de ces fichiers. Cette étape n'a pas été mise en place au sein des plates-formes d'Hybrigenics et il n'était donc pas possible de réaliser la même procédure que pour les classes Java (les fichiers JSP génèrent actuellement des erreurs de compilation).

4.4.2. Résultats

Le rapport rendu par Tattletale a permis de cibler certaines librairies candidates à la suppression. Elles sont inutilisées dans le code et aucune autre librairie n'en dépend. Ces librairies ne sont pas non plus utilisées par les applications web.

Lors du stage, les développeurs ont supprimés 20 packages Java désormais inutiles. Ces packages ont été supprimés par la connaissance du code qu'avait les développeurs et non par l'analyse du code mort. Une nouvelle analyse des rapports de Tattletale a alors montré que

certaines librairies, uniquement utilisées par les packages supprimés, étaient alors également candidates à la suppression.

Concernant les librairies utilisées par les plates-formes, les recherches ont montré qu'elles sont globalement maintenues mais ne sont pas à jour dans l'environnement *javaland*. Les tests de mise à jour de ces librairies se sont révélés concluants. Les compilations des classes n'ont pas produites d'erreurs et aucun nouveau warning n'est apparu.

Cette étude a également montré que les librairies utilisées par les frameworks Castor et JiBX ne pouvaient pas être mises à jour. Pour fonctionner, ces frameworks utilisent des classes produites de manière automatique. Les classes produites par les anciens frameworks de Castor et JiBX ne sont plus fonctionnelles avec les nouvelles librairies. Pour réaliser la mise à jour de ces frameworks, il est donc nécessaire de reconstruire les classes avec les nouvelles librairies.

4.5. Recherche du code mort

De nombreux programmes ont été développés à Hybrigenics. Certains de ces programmes ne sont désormais plus utilisés par les plates-formes mais sont toujours présents au sein de l'environnement *javaland*. Une volonté d'Hybrigenics est de supprimer l'ensemble des packages et/ou classes Java qui ne sont plus utilisés. Pour ce faire, il était alors intéressant de regarder ce que donnerait l'utilisation d'un programme d'analyse du code mort sur l'ensemble des classes Java de l'environnement *javaland*.

4.5.1. Méthodologie

Plusieurs outils effectuant la recherche de codes inutilisés existent. La première étape du travail fut donc de lister les points forts et les points faibles de ces différents outils (voir Figure 16) afin de rechercher celui qui correspond le mieux aux besoins du projet.

	Point Positif	Point Négatif
PMD	- Plug in Eclipse - Trouve le code non utilisé - Vérifie les bonnes pratiques de codage - Trouve le code dupliqué	- Moins flexible - Méthodes et Paramètres jamais utilisés - Se concentre sur les bonnes pratiques
Checkstyle	- Plug in Eclipse - Très configurable - Vérifie les normes de codages - Vérifie les mauvaises pratiques - Trouve le code dupliqué	
FindBugs	- Vérifie les mauvaises pratiques de code - Peut être utiliser comme une tâche Ant	- Moins configurable - Trouve moins de problèmes que les autres - Orienté recherche de bug
UCDetector	- Détecte le code mort	

Figure 16 : Points forts et points faibles des différents outils de recherche du code mort

Le choix s'est finalement porté sur le plugin d'Eclipse UCDetector. Une première tentative fut donc d'utiliser cet outil sur l'ensemble de la *javaland* par l'intermédiaire d'Eclipse. Le nombre de classes développées par Hybrigenics étant trop important, l'utilisation d'UCDetector via Eclipse est impossible.

UCDetector fournit la possibilité de l'utiliser en mode standalone sans passer par l'interface graphique d'Eclipse. Pour faire fonctionner UCDetector par ce type de mode, les sources du projet doivent être téléchargées. Ces dernières permettent de réaliser un jeu de construction contenant les librairies d'Eclipse ainsi que le projet contenant la *javaland*. La détection du code mort est alors réalisée simplement par le lancement d'un script Bash.

UCDetector fournit son rapport sous une forme HTML mais également sous une forme texte. Les informations sont présentées par classe compilée et non par type de code mort. De la même manière que pour l'étude des warnings, l'écriture d'un script Bash a permis de catégoriser ces différents types (classes inutilisées, méthodes inutilisées, …).

4.5.2. Résultats

UCDetector fournit des résultats intéressants concernant les méthodes et/ou variables inutilisées. Les résultats sont en revanche plus difficilement exploitables en ce qui concerne les classes. Dans le cas du code développé à Hybrigenics, UCDetector ne peut être utilisé complètement comme un outil d'aide à la décision. L'expérience des développeurs est donc nécessaire pour déterminer quels sont les programmes inutiles au sein de l'ensemble des classes Java.

Au cours du stage, 20 packages avait été déterminés par les développeurs comme candidats à la suppression. Avant de réaliser cette suppression, les développeurs souhaitaient connaitre les résultats fournis par un outil comme UCDetector. Les résultats ont permis de confirmer la non utilisation de certaines classes mais, d'un point de vue théorique, presque toutes les classes pouvaient être utilisées.

5. Bilan

L'intérêt du stage était de déterminer si les mises à jour de Java et des librairies utilisées par les plates-formes d'Hybrigenics pouvaient poser des problèmes nécessitant par la suite des refontes du code.

L'étude des warnings a montré qu'une grande partie d'entre eux était liée à la non utilisation des génériques. Ces warnings sont principalement localisées dans les classes permettant la réalisation du mapping objet/relationnel. Ces classes étant produites par un générateur développé à Hybrigenics, une tentative de modification de ce dernier a alors été entreprise pour tenter de corriger les warnings. Ce travail a montré que de telles modifications n'étaient pas possibles sans réaliser une refonte plus profonde du générateur.

L'étude des librairies a montrée plusieurs points intéressants. Premièrement, certaines librairies ont pu être supprimées car elles n'étaient plus exploitées. Deuxièmement, pour la très grande majorité des librairies, les opérations de mises à jour n'ont pas montré de problèmes majeurs nécessitant une refonte du code.

L'étude du code mort a permis de montrer que l'expérience des développeurs est nécessaire pour déterminer qu'elles sont les packages et/ou classes encore utilisées par les plates-formes. Suite à ce travail, 20 packages ont été supprimés. La suppression de ces packages a permis également de supprimer d'autres librairies liées uniquement à ces packages.

6. Conclusion

Cette première expérience dans un service informatique a été très intéressante et enrichissante. Ce stage était utile aux développeurs afin de déterminer les conséquences des différentes mises à jour de Java et des librairies. Ce stage a également été conçu pour que je puisse améliorer mes connaissances de la technologie Java.

J'ai ainsi pu découvrir les principes de la construction de plates-formes Java par l'intermédiaire des outils Ant et Subversion. Les travaux de maintenance m'ont également montré l'importance d'avoir des bonnes pratiques que ce soit dans la manière de concevoir des programmes mais également dans la mise en place de ces derniers dans un système en production.

Dans un domaine plus pratique, ce stage m'a également permis de découvrir l'utilité des scripts Bash. Ces scripts permettent d'automatiser certaines procédures mais sont également très utiles pour effectuer le parsing de fichiers grâce à l'utilisation des outils du système comme sed ou grep.

Les travaux effectués au cours de ce stage m'ont également montré l'importance de la maintenance du code. Ce travail est fondamental et inévitable pour qu'une plate-forme puisse continuer de fonctionner.

Ce stage m'a permis d'observer la vie d'un informaticien au sein d'une entreprise. Il m'a également montré les nouvelles compétences que je devais encore acquérir. Cela m'a donc motivé et déterminé à continuer mes études dans le domaine de l'informatique.