Analyse Dynamique de l'architecture de Hibernate en lien avec les stratégies de mapping.

Travail de diplôme réalisé en vue de l'obtention du diplôme HES

par :

Dmitry GOMAN

Conseiller au travail de diplôme :

(Philippe DUGERDIL, professeur HES)

Genève, le 9 novembre 2012

Haute École de Gestion de Genève (HEG-GE)

Filière Informatique de gestion

Déclaration

Ce travail de diplôme est réalisé dans le cadre de l'examen final de la Haute école de gestion de Genève, en vue de l'obtention du titre Bachelor en Informatique de gestion. L'étudiant accepte, le cas échéant, la clause de confidentialité. L'utilisation des conclusions et recommandations formulées dans le travail de diplôme, sans préjuger de leur valeur, n'engage ni la responsabilité de l'auteur, ni celle du conseiller au travail de diplôme, du juré et de la HEG.

« J'atteste avoir réalisé seul le présent travail, sans avoir utilisé des sources autres que celles citées dans la bibliographie. »

Fait à Genève, le 9 novembre 2012

Dmitry GOMAN

Remerciements

Je tiens à remercier tout mon entourage qui m'a beaucoup soutenu tout au long de ce travail de Bachelor, votre compréhension et bonne humeur m'ont été très précieuses. Je tiens particulièrement à remercier ma grand-mère qui ne peut malheureusement pas participer à ce moment qu'elle attendait depuis si longtemps. Elle a tout donné dans sa vie pour que je réussisse et je m'en souviendrai tout au long de la mienne.

J'aimerais beaucoup remercier M. Philippe DUGERDIL qui a su me diriger et me donner des conseils précieux. J'aimerais aussi remercier M. David SENNHAUSER et M. Mihnea NICULESCU qui m'ont beaucoup aidé également.

Sommaire/Résumé

Ce travail de Bachelor représente une étude des différentes stratégies de mapping objet/relationnel et le fonctionnement de l'architecture interne du framework Hibernate. Mon choix s'est orienté dans cette direction suite à une envie d'apprendre d'avantage sur cet outil largement utilisé dans le monde professionnel. L'idée est venue de l'intérêt que je porte pour la couche de persistance de données.

La structure de ce travail est composée de quatre éléments essentiels :

- Introduction aux mondes objet et relationnel
- Stratégies de mapping, mises en situation
- Architecture interne Hibernate
- Analyse et conclusion

L'objectif principal de ce travail de Bachelor est d'étudier l'architecture interne Hibernate et son fonctionnement en fonction du choix de la stratégie de mapping objet/relationnel.

Nous allons tenter de déterminer les classes et les packages les plus importants dans l'implémentation de mapping Hibernate, puis étudier les relations entre ces derniers et leurs comportements durant l'exécution de nos use cases.

Une manière de déterminer ces éléments est le traçage d'exécution. Pour ce faire, nous allons d'abord instrumenter le code source d'Hibernate, puis l'inclure dans un projet Java et exécuter le tout en implémentant à chaque fois une stratégie de mapping différente. Les traces générées nous donnerons des informations nécessaires à l'étude de l'architecture interne Hibernate.

Table des matières

Liste des Tableaux

Liste des Figures

Introduction

Le problème de persistance de données est un sujet bien connu par la communauté des développeurs aujourd'hui. De nombreuses applications, si ce n'est pas toutes, utilisent au moins une base de données relationnelle basée sur le langage SQL (MySQL, SQLServer, Oracle, etc.). Et les langages orientés objet ont définitivement prouvé leur utilité dans le développement des applications.

Dans ces premières pages nous allons essayer de comprendre ce que représente exactement cette problématique de persistance de données et pourquoi il y a-t-il une non-correspondance entre le monde objet et le monde relationnel. En répondant à ces questions, nous aurons acquis une bonne base qui nous permettra de mieux comprendre l'utilité du framework Hibernate et surtout d'aborder une de ses fonctionnalités clés : mapping objet/relationnel.

Avant l'arrivée des framework de persistance tels que TopLink, Hibernate et de nombreux autres, la communication entre des applications écrites en Java et des bases de données a été longtemps (et l'est toujours d'ailleurs) gérée à l'aide de l'API JDBC (Java Data Base Connectivity). Voici ci-dessous une définition officielle de JDBC :

> *« The Java Database Connectivity (JDBC) API is the industry standard for database-independent connectivity between the Java program language and a wide range of SQL databases and other tabular data sources... »*[1]

JDBC permet donc sans problèmes de se connecter à une base de données relationnelle et effectuer des opérations rudimentaires CRUD.

Il est important de se rappeler à ce stade que JDBC nécessite de la part du développeur une bonne connaissance non seulement du langage SQL, mais aussi du fonctionnement d'une base de données relationnelle (représentation des graphes d'objets, les différents types, etc.). Il n'y a, bien entendu, pas de problèmes lorsqu'il s'agit d'une petite application à quelques tables, dans ce cas la

---

1 *ORACLE, Java SE technologies – Database [en ligne].* http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html *(consulté le 06.07.2012)*

persistance peut être très bien gérée avec JDBC. Mais à mesure que le nombre de tables et les dépendances entre elles augmentent, l'application risque de devenir très dépendante de la base de données et par conséquent, peu réutilisable. Cette situation a de fortes chances de devenir un véritable calvaire pour le développeur et d'engendrer des coûts supplémentaires, ce qui fait que l'on passe plus de temps à résoudre des problèmes liés à la base de données qu'à développer le cœur de la logique métier.

Non-correspondance objet/relationnel

La problématique liée à la non-correspondance entre le monde objet et le monde relationnel n'est pas visible tout de suite. Elle engendre selon Christian Bauer et Gavin King (KING Gavin, Bauer Christian. *Hibernate.* Paris: CampusPress, 2005. P. 9), l'inventeur de Hibernate, essentiellement cinq problèmes/différences majeures :

- Granularité
- Sous-types
- Identité
- Associations
- Navigation dans le graphe d'objets

Granularité

Afin de mieux comprendre ce problème de granularité prenons un exemple de modèle objet suivant :

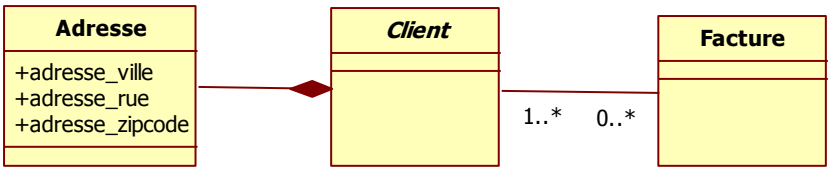


Figure 1. Problème de granularité.

L'Adresse peut devenir une entité à part entière dans un modèle objet, ce qui est une situation courante. Mais un DBA (Data Base Administrator) ne serait pas de même avis que le concepteur de l'application. Le fait d'accepter Adresse comme une entité dans une base de données impliquerait la création d'un champ supplémentaire dans cette table, la clé primaire (respect de la 3^{e} forme normale) afin d'éviter toute redondance de données. Il faudrait également créer un champ de clé étrangère pour l'Adresse du côté Client. Sans ces clés on aurait de la peine à savoir quelle Adresse appartient à quel Client. De plus, il faudrait une requête en plus pour récupérer l'adresse d'un client, alors que si les champs d'Adresse étaient inclus dans Client, une seule requête suffirait. Et qui dit plus de requêtes, plus de problèmes de performance.

Le problème ne se pose même pas du côté objet, la liaison entre les entités se fait naturellement par référence, on peut ainsi avoir une granularité du modèle objet très fine, sans que cela engendre des changements important comme du côté relationnel. Il est évident que dans la situation à deux classes le problème de granularité objet/relationnel ne se pose pas vraiment. On peut donc avoir une granularité fine du côté objet (Adresse et Client sont gérés en tant qu'entités à part entière) et faible du côté relationnel (entité Client qui contient les champs d'Adresse). Mais dans le cas de grosses applications à plusieurs centaines de classes, il faut vraiment trouver un consensus, car dans le cas contraire, on risque de diminuer drastiquement la maintenabilité et la réutilisabilité des composants du côté application et la performance du côté base de données.

Imaginez seulement le cas où l'on devait encore rajouter des informations supplémentaires à l'Adresse : AdresseDomicile, AdresseProfessionnel, AdresseVacances, etc. Le problème se règle facilement du côté objet grâce à l'héritage, mais du côté relationnel ce n'est pas toujours le cas, à moins que votre base de données supporte déjà l'héritage. Devrait-on donc inclure toutes ces adresses dans une seule table Client ou créer une nouvelle table Adresse ? C'est discutable. D'un côté comme de l'autre il y a ses avantages et inconvénients. On peut soit favoriser le modèle objet tout en augmentant la flexibilité de notre application, soit c'est le relationnel qui gagne et on créerait ainsi une non-correspondance entre les deux modèles.

Sous-types

Ce problème est plus connu que le précèdent, il s'agit ici du problème d'héritage. Le concept d'héritage est indéniablement la pierre angulaire de toutes applications codées en langage objet, alors que la technologie SQL ne propose aucun outil pour gérer cet aspect. Il ne nous est donc pas possible de créer une table dans la base de données qui étend une autre.

Prenons l'exemple précédant, la table Client possède deux sous classes Privé et Fournisseur :

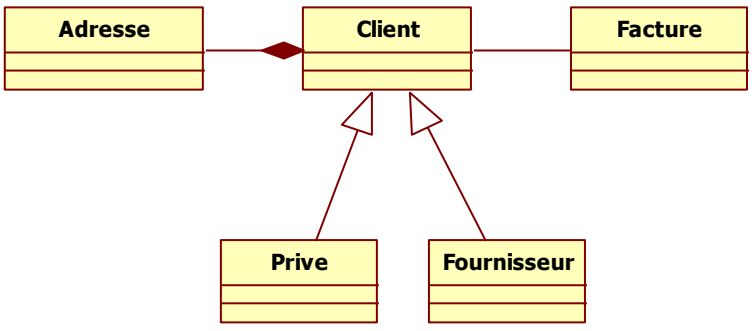


Figure 2. Problème des sous-types.

Dans le modèle objet, l'héritage est parfaitement intégré et nous pouvons par exemple typer les instances de Privé ou de Fournisseur comme étant Client. Ceci n'est malheureusement pas possible avec la technologie SQL. Comment peut-on dire en SQL que la table Privé est en fait une extension de la table Client et qu'elle possède les mêmes attributs ? On peut par exemple créer les mêmes champs dans la table Privé que dans la table Client, mais cela contredit à la 2^{e} forme normal : tout attribut ne composant pas un identifiant dépend d'un identifiant. Alors qu'ici, on aurait donc deux mêmes attributs qui dépendent des identifiant différents. Et tout cela sans mentionner la redondance de l'information que cela engendrerait.

Identité

Une autre incohérence survient lorsque l'on cherche à vérifier l'identité des objets manipulés et prouver qu'ils sont différents. Comparer deux objets en JAVA revient à faire deux choses : comparaison par valeur (redéfinition de la méthode *equals*) et comparaison par identité (emplacement mémoire) alors qu'en SQL en se base uniquement sur les valeurs des clés primaires comme critère de comparaison.

Associations

Il existe en fait une différence très subtile entre les manières de représenter les associations entre entités dans les deux mondes. En JAVA par exemple, on représente des associations à l'aide de références d'objets et des collections de références d'objets. Alors que dans une base de données relationnelle ce sont des clés étrangères qui le font.

La non-correspondance devient tout de suite visible lorsqu'il s'agit des associations de types « plusieurs-à-plusieurs ». Il est possible en JAVA d'avoir de telles associations (un objet A possède une collection d'objets B; cet objet B possède lui-même une collection d'objets A). Nous pouvons donc facilement, depuis l'objet A, récupérer (naviguer vers) n'importe quelle instance de B pour autant qu'elle se trouve dans la collection de A et vice versa. Cette navigation ne se fait pas de la même manière du côté relationnel, elle est en fait complétement dépourvue de sens, « car vous pouvez créer des associations de données libres avec les jointures de tables et projection. » (BAUER, KING, 2005, p. 14). Seule chose que l'on peut faire dans le cas d'une association « plusieurs-à-plusieurs » est de créer une table de liaison, qui contiendra un tableau de correspondance des clés primaires des deux tables en question, ce qui est, à présent, dépourvue de sens du côté objet, car il n'y a pas besoin de créer une nouvelle entité qui nous indiquerait qui est liée avec qui.

Navigation dans un graphe d'objets

La navigation dans un graphe d'objets dans le monde objet se fait très naturellement grâce aux références. Cela ne pose aucun problème tant que l'on y reste, le problème survient lorsque l'on passe la frontière pour aller chercher des informations dans le monde relationnel. Si la profondeur du graphe est importante (un objet père, qui contient la référence sur un fils, qui est lui-même père d'un autre fils, etc.), la performance de l'application risque d'être drastiquement diminuée.

Dans un modèle relationnel, si la profondeur d'un graphe d'objets est importante, il serait très coûteux en terme de performance de le charger au complet afin de consulter seulement un élément bien précis, car le coût d'appel d'un tel élément est proportionnel à sa profondeur dans un graphe. Il s'agit là d'un problème célèbre de « n+1 sélections ».

Comme nous venons de voir, la liste des problèmes est plutôt considérable et leurs conséquences peuvent s'avérer critiques pour une application. Selon BAUER et KING (2005, p.16): « 30% du code des applications Java est consacré à gérer la mise en rapport SQL/JDBC et la non-correspondance des modèles objet/relationnel. ».

L'objectif de ce travail de Bachelor

Le framework Hibernate a éveillé ma curiosité essentiellement par sa popularité dans le monde professionnel. Je me suis donc intéressé de plus près à cet outil et plus précisément à une de ses fonctionnalités clé, qui est le mapping objet/relationnel. Finalement, j'ai décidé de faire mon travail de Bachelor au tour de ce sujet. Après une discussion avec M. Philippe Dugerdil, mon professeur de cours Génie logiciel, il m'a proposé un sujet, dont l'objectif serait de découvrir l'architecture fonctionnel du framework Hibernate dans le cadre d'un mapping objet/relationnel.

De nos jours, Hibernate est, certes, très populaire, mais il est également une boite noire pour de nombreux développeurs en ce qui concerne son architecture interne. Elle n'est abordée que partiellement dans la documentation officielle du framework. Le livre écrit par Gavin King, son inventeur, reste une source très fiable malgré le fait qu'il a été publié en 2004. Mais de nombreuses versions ont vu le jour depuis et l'architecture du framework a évolué.

L'objectif donc que nous nous sommes fixé est de comprendre comment le framework fonctionne de l'intérieur et de déterminer si ses composants sont faiblement couplés et cohésifs. Nous allons nous intéresser plus particulièrement au fonctionnement lors d'une implémentation des stratégies de mapping objet/relationnel, d'une part parce que ce sujet me tient à cœur et d'autre parce que l'architecture Hibernate est tout simplement immense, rien que la partie *core* regroupe en elle-même plus de deux mille classes. L'étude de cette envergure prendrait un certain temps, voir un temps incertain.

Pour déterminer si les composants Hibernate sont faiblement couplés et cohésifs, nous devons pouvoir mesurer ces métriques. Une des manières pour y arriver est de générer une trace d'exécution. « Il s'agit donc en premier lieu de définir quels sont les éléments de code à « tracer ». Dans le cadre de Java, il s'agit essentiellement de connaitre les méthodes exécutées et les classes auxquelles elles appartiennent, de manière univoque. Il faut donc identifier également le package auquel ces classes appartiennent. De plus, la relation hiérarchique d'appel doit être conservée : il s'agit de savoir quelle méthode appelle quelle autre. Finalement, il est nécessaire de connaitre les paramètres passés lors des appels de méthodes car ces derniers sont analysés lors de l'exploitation de la trace. En conséquence, plutôt que d'instrumenter les appels de méthodes, il faut instrumenter les méthodes elles-mêmes en indiquant dans la trace le début et la fin de la méthode. Il sera ainsi possible de déterminer que tous les éléments tracés entre ce début et cette fin sont des éléments appelés par cette méthode et ainsi de suite. » (DUGERDIL, « Source code instrumentor-JAVA », p.3).

L'outil de génération de traces que nous allons utiliser dans le cadre de ce travail a été développé à la HEG dans l'équipe du Professeur Philippe Dugerdil dont le nom est « Instrumentor ».

Il existe deux version d'Instrumentor, la version standalone et la version plugin Eclipse. Nous utiliserons la version Eclipse. Cette version est composée de deux archives jar :

- ***ch.hesge.csim2.instrumentor.runtime_1.0.0.jar*** *(à mettre dans le BuildPath du projet à instrumenter)*
- ***ch.hesge.csim2.instrumentor_1.0.0.jar*** *(à mettre dans le répertoire /plugins d'Eclipse)*

Une fois les projets instrumentés, compilés et que les traces sont générées, nous utiliserons un autre outil développé également dans l'équipe de M. Dugerdil dont le nom est « DDRA ». Cette application nous permettra de faire l'analyse nécessaire de la trace pour déterminer le degré de couplage et de cohésion.

1. Mapping objet/relationnel Hibernate

Dans cette partie nous allons étudier la fonctionnalité mapping objet/relationnel proposée par Hibernate. Nous connaissons désormais les difficultés que peut poser la non-correspondance entre les deux mondes dans une application JAVA. Etudions à présent plus en détails cette fonctionnalité.

Dans un premier temps nous allons étudier le concept de mapping et de solution ORM (Hibernate en est une). Dans un deuxième temps nous passerons à la pratique et nous apprendrons à mapper des objets, nous verrons qu'il existe plusieurs manières pour le faire, ainsi que des contraintes d'utilisation. Ensuite, nous verrons les avantages et les inconvénients du mapping objet/relationnel Hibernate. Et pour finir nous étudierons les différentes stratégies de mapping avec des exemples de code.

1.1 Mapping objet/relationnel

Le terme « mapping » provient de l'anglais et signifie « map-making », ce qui se traduit en français par « le fait de créer des cartes ». En programmation, on utilise ce terme pour décrire des liens entre des objets du domaine et des tables du modèle de données. L'objectif poursuivi par le mapping objet/relationnel est de cartographier l'application, afin de savoir quelle classe est associée à quelle table de la base de données, même chose pour les attributs et les champs. Nous utiliserons beaucoup par la suite le verbe « mapper », donc si on dit qu'une classe A est mappée dans une table B, cela veut dire que les informations de cette classe, ses attributs donc, sont sauvegardés des champs de la table B.

Cette cartographie se fait de manière inconsciente dans des petites applications, il s'agit en fait de faire la chose suivante, prenons l'exemple d'une classe Voiture :

Classe *Voiture* est associée avec la table *VOITURE*.

Attribut *Voiture.marque* est associé avec le champ *VOITURE_MARQUE*.

Attribut *Voiture.nbSieges* est associé avec le champ *VOITURE_NB_SIEGES*.

Et ainsi de suite.

Mais dans des applications avec un modèle d'objets complexe (multiples héritages, niveaux d'héritage) et une granularité différente par rapport à celle du modèle relationnel (cf. « Introduction. Problème de granularité »), la situation peut être différente. Il se peut, en effet, qu'il y ait un mélange de différents types de mapping. C'est-à-dire qu'il puisse y avoir, par exemple, des classes qui sont mappées dans une seule et unique table et d'autres qui sont toutes mappées dans

une seule table commune (le cas d'un héritage). Ce mélange n'est pas un hasard ou une erreur d'un développeur amateur. Il a été aperçu en fait que les classes peuvent être mappées selon trois stratégies majeures.

Une question très naturelle survient alors : « Pourquoi s'embêter avec ces stratégies, alors que l'on peut très bien se contenter d'une table pour chaque classe ? ». En effet, ceci est une des trois stratégies de mapping que nous allons étudier, mais appliquer une et une seule stratégie à l'ensemble des classes d'une application, surtout dans des cas complexes, n'est pas toujours la meilleure des solutions (cf. « Introduction. Problème de granularité et des sous types »). Dans la pratique, chaque classe est mappée selon une des trois stratégies.

L'objectif que nous nous fixons, dans les pages qui suivent, est de découvrir ces trois stratégies de mapping, ainsi que d'apprendre à les mettre en œuvre avec le framework Hibernate.

1.2 Solution ORM

Le concept de mapping objet/relationnel n'est pas nouveau, il est apparu vers la fin des années 1980[2] et il a fallu une vingtaine d'années pour que les solutions ORM (*Object/Relational Mapping*, l'acronyme anglais de mapping objet/relationnel) telle qu'Hibernate prennent de l'ampleur et s'affirment auprès des développeurs.

Une solution ORM est une API qui facilite la gestion de base de données dans une application Java. En d'autres termes, elle « libère le développeur de 95% du travail lié à la persistance des objets, comme l'écriture d'instructions SQL complexes avec de nombreuses jointures de tables… » (BAUER, KING, 2005, p. 32).

Avec un tel outil, le développeur peut s'affranchir complétement du code SQL et se concentrer sur le corps du développement métier. Cela ne veut pas pour autant dire que la connaissance de la technologie SQL n'est pas requise si on utilise une solution ORM. Il est important que le développeur connaisse un minimum, bien évidemment plus la complexité du modèle objet est importante (multiples héritages, associations, interfaces, etc.) plus le degré de connaissance doit l'être aussi.

[2] KING Gavin, Bauer Christian. *Hibernate.* Paris: CampusPress, 2005. P. 25

1.3 Annotations et fichiers de mapping XML

Il est possible de mapper des objets de deux manières différentes : annotations JPA (disponibles à partir de Java 5.0) et fichiers XML. Nous travaillerons plutôt avec des annotations que des fichiers XML. C'est un choix personnel, qui n'a pas d'influence sur le travail. Voici ci-dessous deux exemples d'une classe mappée avec des annotations (représentées avec des « *@Xxx* ») et la même classe mappée à l'aide d'un fichier XML.

```java
package org.java.hibernate.dom;

import javax.persistence.Entity;

@Entity
public class User {

    @Id
    private int userId;
    public int getUserId() {
        return userId;
    }
    public void setUserId(int userId) {
        this.userId = userId;
    }
    public String getUserName() {
        return userName;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
    private String userName;
}
```

Figure 3. Version annotée de la classe User.

```xml
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd" >

<hibernate-mapping package="org.java.hibernate.xml">
    <class name="User" table="user">
        <id
            column="USER_ID"
            name="Id"
            type="integer"
        >
            <generator class="vm" />
        </id>
        <property
            column="USER_NAME"
            length="50"
            name="UserName"
            not-null="true"
            type="string"
         />
    </class>
</hibernate-mapping>

```

Figure 4. Version fichier XML de la classe User.

Pour ceux qui préfèrent la version XML du mapping, sachez qu'il faut bien faire attention à l'emplacement du fichier de mapping d'une classe, ce chemin doit être précisé dans le fichier « *hibernate.cfg.xml* » (nous étudierons ce fichier en détails plus tard), car le cas échéant, Hibernate ne trouvera pas ce fichier et par conséquent, ne pourra pas mapper la classe.

Voici ci-dessous un exemple de la classe *org.java.hibernate.dom.User* et de son fichier de mapping version XML, dont le chemin est précisé dans le fichier de configuration de Hibernate à l'aide de la balise <mapping> :

```xml
        <mapping resource="org/java/hibernate/xml/User.hbm" />
    </session-factory>
</hibernate-configuration>
```

Figure 5. Balise <mapping>.

1.3.1 Avantages et inconvénients

Il existe néanmoins des avantages et des inconvénients de ces deux manières de mapper des classes Java. L'annotation permet de mapper « en live » lorsqu'on développe une application. Cette approche est conseillée lorsqu'on ne possède pas encore la base de données, car les annotations permettent de créer des tables et des champs sans que le développeur le fasse manuellement. Alors que le mapping avec des fichiers XML se fait à partir d'une base de données préalablement

existante, on ne renseigne que les tables à mapper et cela nous crée des classes dans notre application.

En tant qu'un jeune développeur Java et un novice avec Hibernate, je conseille plutôt utiliser des annotations, elles sont plus simples à comprendre. Il y a une très bonne suite de tutoriels faits par Koushks sur Youtube, c'est un très bon moyen de débuter avec Hibernate.

Un autre avantage des annotations est la taille du projet. Les projets annotés contiennent moins de fichiers, car il n'y a plus de fichiers XML. La maintenance est plus facile, car toute l'information se trouve dans la classe Java et selon ce que j'ai pu observer sur les forums, les annotations sont moins sujettes aux erreurs.

1.3.2 Annotations JPA et Hibernate

Les annotations que nous allons utiliser dans ce travail proviennent à la fois de Java Persistence API et de Hibernate (JPA est d'ailleurs inclus dans les fichiers d'Hibernate, donc il n'y a pas besoin de le télécharger en plus). Les annotations proposées par ces deux API sont complémentaires et fonctionnent très bien ensemble. Les annotations de base proviennent de JPA alors que les annotations plus sophistiquées, telles que les générateurs de clés primaires, sont plutôt proposées par Hibernate.

Voici ci-dessous un exemple d'annotations des deux API :

```
@ElementCollection
@JoinTable(name="UTILISATEUR_ADRESSE",
        joinColumns=@JoinColumn(name="USER_ID")
)
@GenericGenerator(name="hilogen", strategy="hilo")
@CollectionId(columns = { @Column(name="ADRESSE_ID") }, generator = "hilogen", type = @Type(type="long"))
private Collection<Adresse> listeAdresses = new ArrayList<Adresse>();
```

Figure 6. Annotations JPA et Hibernate.

Les deux premières annotations @ElementCollection et @JoinTable proviennent de JPA et les deux dernières @GenericGeneartor et @CollectionId proviennent de Hibernate.

1.4 Stratégies de mapping

Une des choses les plus intéressantes du langage orienté objet est sans aucun doute l'héritage et le polymorphisme. C'est aussi un challenge, pour toute solution ORM digne de ce nom, d'implanter un mécanisme qui permettrait la mise en œuvre de ce concept. Dans les paragraphes qui suivent, nous allons apprendre à mapper des classes JAVA de telle sorte pour que l'on puisse utiliser l'héritage. Afin d'illustrer nos propos, nous allons nous servir de l'exemple suivant :

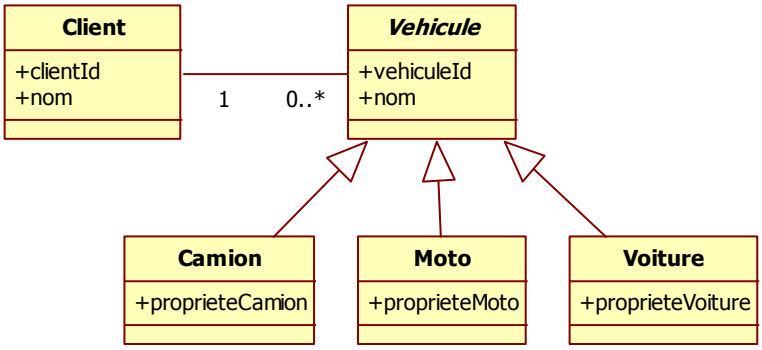


Figure 7. Exemple de diagramme de classes.

1.4.1 Une table par classe (TABLE PER CLASS)

La manière la plus naturelle de mapper des classes dans une base de données serait de créer une table pour chaque classe concrète d'un modèle de domaine. Ainsi toutes les propriétés, héritées ou non, de la classe seront mappées dans une seule et unique table. Cette solution convient parfaitement pour des applications qui répondent aux critères suivants :

- Absence d'associations polymorphiques ou de requêtes polymorphiques
- Les classes en bas de l'arbre d'héritage possèdent peu d'information, d'attributs

Tout d'abord soyons claires sur ce que c'est exactement une association polymorphique, selon BAUER et KING (2005, p. 111) « une association polymorphique est une association à une superclasse et donc à toutes les classes dans la hiérarchie ». Dans le modèle présentée au de but de ce chapitre, la classe *Client* a une association polymorphique avec la superclasse classe *Vehicule.*

On parle souvent aussi des requêtes polymorphiques, dans notre cas, une requête polymorphique est celle qui nous retourne des instances des sous-classes de la superclasse *Vehicule*, lorsqu'on se réfère à cette dernière.

Le problème avec cette stratégie survient lorsqu'on désire, par exemple, de retrouver toutes les véhicules d'un client. Du côté JAVA il n'y a rien de plus simple, car *Client* peut posséder une liste de *Vehicule* et on retrouve les instances des classes voulues grâce aux références. Avec la stratégie une table par classe concrète, la classe abstraite *Vehicule* n'existe pas du côté relationnel. Du coup, il ne nous est pas possible de représenter une association polymorphique à la superclasse pour des classes *Camion, Moto* et *Voiture*. Ce qui nous pose un problème, car la classe *Vehicule* est associée à *Client*, les tables « filles » auraient donc besoin d'une référence de clé étrangère à la table *Client*.

Dans le cas d'une association « One To Many » de *Client* avec *Vehicule*, Hibernate créera une table d'associations *client_vehicule* et utilisera une jointure pour trouver les véhicules d'un client. À

première vue le problème semble être réglé, mais en réalité cela n'est pas vraiment le cas, car les « contraintes de clés étrangères standard se réfèrent à une table exactement. Il n'est pas simple de définir une clé étrangère qui se réfère à plusieurs tables. On peut expliquer cela en rappelant que le Java <…> est moins strictement typé que le SQL. » (BAUER, KING, 2005, p.11). Vous trouverez ci-dessous le modèle conceptuel de données de notre exemple mappé selon la stratégie TABLE PER CLASS.

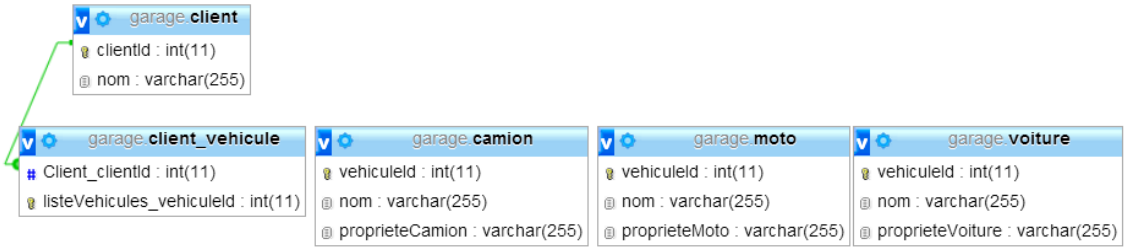


Figure 8. Stratégie de mapping : TABLE PER CLASS.

1.4.2 Une table par hiérarchie de classes (SINGLE TABLE)

Une autre manière de mapper des objets serait de créer une seule table par hiérarchie de classes. Cela veut dire que dans notre cas *Camion*, *Moto* et *Voiture* seront mappées dans une seule et unique table *Vehicule*. C'est la stratégie par défaut utilisée par Hibernate.

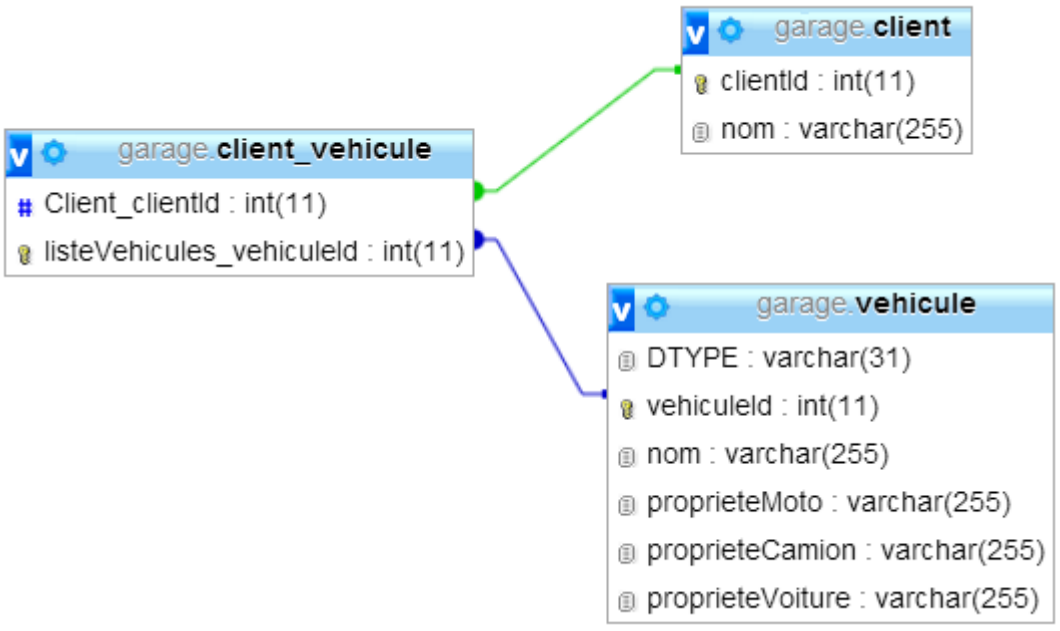


Figure 9. Stratégie de mapping : SINGLE TABLE.

Pour distinguer les sous-classes on utilise la valeur de la colonne discriminateur (DTYPE), qui est créée par défaut par Hibernate. Sa valeur par défaut est le nom de la classe de l'enregistrement, son but est tout simplement de faire la distinction entre les différents sous-types enregistrés dans une table.

Cette stratégie convient lorsqu'on a un diagramme de classes où les requêtes polymorphiques deviennent une nécessité et que les sous-types possèdent peu de propriétés (faible impact sur la performance de la base de données au chargement des sous-types dans la mémoire). C'est-à-dire que les sous-types comportent majoritairement du comportement diffèrent (des méthodes propre à eux) par rapport à la superclasse.

C'est d'ailleurs le mapping par défaut d'une hiérarchie de classes avec Hibernate. Pour ce faire il suffit d'annoter les classes de notre exemple comme *@Entity* et Hibernate va mapper ces classes dans une seule et unique table Vehicule.

1.4.3 Une table par sous-classe (JOINED)

Cette stratégie est plutôt utilisée dans des cas complexes avec des multiples niveaux d'héritage où les sous-classes possèdent un nombre important de propriétés. L'héritage est représenté grâce aux clés étrangères et *« chaque sous-classe qui déclare des propriétés persistantes (dont les classes abstraites ou même les interfaces) possède sa propre table ».* (BAUER, KING, 2005, p.107). Si on reprend notre exemple, on va avoir une table pour toutes les classes et les sous-classes de *Vehicule* :

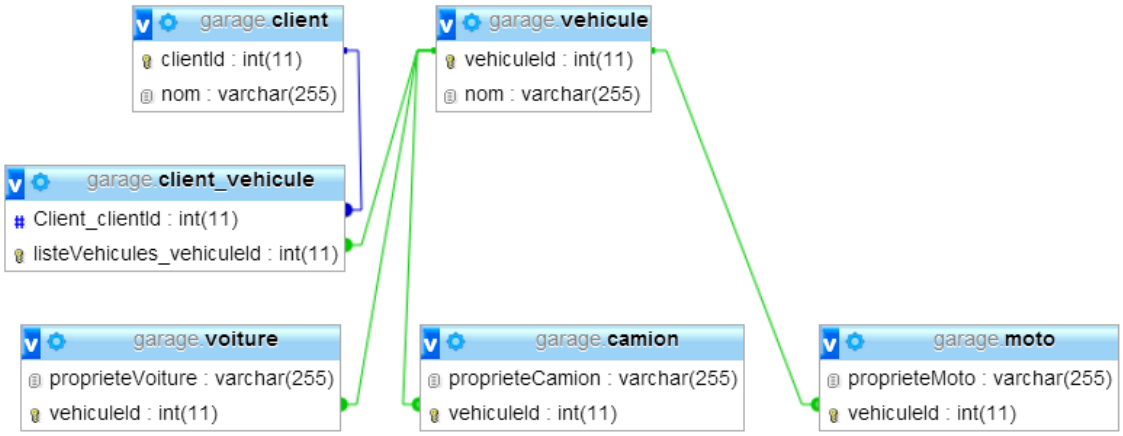


Figure 10. Stratégie de mapping : JOINED.

Comme vous avez pu le remarquer, la seule différence du côté relationnel entre la stratégie SINGLE TABLE et JOINED réside dans le fait que chaque table possède que ses propres attributs non hérités.

1.4.4 Recommandations techniques

Il n'y a rien de plus simple pour définir une stratégie de mapping avec Hibernate. Pour cela nous avons besoin essentiellement de quatre annotations de l'API JAVA Persistence, qui est fournie avec le framework Hibernate.

@Entity

@Id

@GeneratedValue

@Inheritance

Les annotations ***@Entity*** et ***@Inheritance*** doivent être placées avant la déclaration de la classe :

```
 7 import javax.persistence.Inheritance;
 8 import javax.persistence.InheritanceType;
 9
10 @Entity @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
11 public abstract class Vehicule {
```

Figure 11. Annotations *@Entity* et *@Inheritance.*

@Entity permet à Hibernate d'indiquer que la classe suivante doit être sauvegardée dans la base de données.

@Inheritance(strategy=InheritanceType.XXX) permet de définir une stratégie de mapping. On met le nom de la stratégie à la place des « ***XXX*** », voici ci-dessous des exemples de déclarations de chaque stratégie de mapping.

@Inheritance(strategy=InheritanceType.SINGLE_TABLE)

@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)

@Inheritance(strategy=InheritanceType.JOINED)

Le fonctionnement de cette dernière annotation mérite d'être étudié un peu plus en détails. Elle a tout son sens uniquement quand la classe annotée possède des classes qui en héritent. Dans notre cas, nous avons annoté la classe *Vehicule,* qui est abstraite et qui possède trois classes filles. À chaque fois que Hibernate traitera des classes qui héritent de cette dernière, il appliquera la stratégie renseignée.

Si on reprend l'exemple de *Véhicule* et de la classe *Moto* qui en hérite, à chaque fois que l'on va demander à Hibernate de sauvegarder une instance de *Moto*, il va détecter la stratégie de la classe mère et l'appliquer pour des classes fille.

Afin de sauvegarder une classe dans une base de données, Hibernate a besoin de connaître son ID, autrement dit, nous devons spécifier à l'aide de l'annotation ***@Id*** quel attribut de la classe servira d'ID. L'annotation *@GeneratedValue* permet à son tour de définir une stratégie de génération de l'ID. La stratégie la plus appréciée par les développeurs est sans doute *AUTO*. En mettant le paramètre *strategy=GenrationType* de cette annotation à *AUTO*, nous indiquons que l'ID de cette entité sera incrémenté automatiquement par Hibernate, à chaque fois que nous voudrons par exemple sauvegarder une nouvelle instance de cette classe dans la base de données. Ainsi on ne se préoccupe plus de consulter la base de données pour trouver le dernier ID enregistré.

```
1  package gomand.dom;
2
3  import javax.persistence.Entity;
4  import javax.persistence.GeneratedValue;
5  import javax.persistence.GenerationType;
6  import javax.persistence.Id;
7
8  @Entity
9  public class Moto {
10
11     @Id @GeneratedValue(strategy=GenerationType.AUTO)
12     private int id;
```

Figure 12. Annotations *@Id* et *@GeneratedValue.*

2. Architecture Hibernate

Hibernate est un framework en couche où les interfaces entre elles sont étroitement liées. Hibernate représente une couche de persistance pour une application qui communique avec la base de données ou son rôle est de gérer les entités que l'on veut persister, le cache et la synchronisation de données client avec la base de données. Hibernate facilite grandement la vie des développeurs grâce à ces interfaces basiques. Il est très facile de démarrer avec Hibernate, pour sauvegarder un objet il suffit de quelques appels de méthodes, à condition que la configuration soit bien faite bien sûr.

Hibernate utilise d'autres API telle que JDBC et JTA (Java Transaction API), cette utilisation est totalement transparente avec Hibernate pour le client.

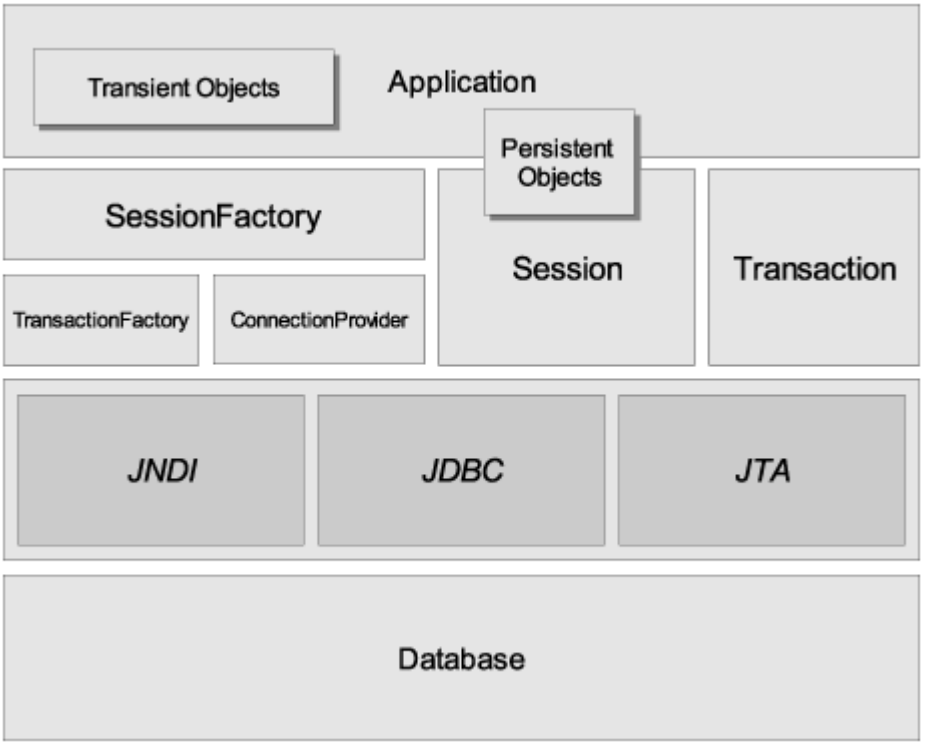


Figure 13. Architecture Hibernate.

2.1 Interfaces basiques

2.1.1 Interface Session et SessionFactory

L'interface *org.hibernate.SessionFactory* est la factory des instances de *org.hibernate.Session*. Une *SessionFactory* est en général unique par application[3] et peut être donc créée par exemple à l'initialisation de cette dernière. Cette interface est loin d'être légère, car elle est non seulement responsable de fournir des *Session*, mais aussi de gérer toutes les métadonnées liées au mapping objet/relationnel, la mise en cache des instructions SQL et contenir le cache des entités utilisées par l'application cliente. Elle est aussi ce que l'on appelle *thread-safe*, cela veut dire qu'elle peut être partagée entre de nombreux threads de l'application.

À son tour, *Session* est un objet beaucoup moins lourd que *SessionFactory* et on peut en avoir plusieurs par application. Cet objet gère à la fois ce qu'on appelle le cache de premier niveau qui est constitué des objets persistés, les *Transaction* et la recherche typiquement des objets par identifiant.

2.1.2 Interface Transaction

L'interface Transaction instanciée par la Session (*Session.beginTransaction()*) et associée à cette dernière. Son rôle est de synchroniser (*registerSynchronization(Synchronization s) :void*) par exemple les informations d'une session avec la base de données.

---

3 Dans le cas où l'application gère plusieurs bases de données, il y aura une *SessionFactory* par base de données.

3. Architecture de l'expérimentation

L'architecture de l'expérimentation est basée sur un use-case qui a pour but de mettre en œuvre le mapping « minimal » de Hibernate. Cela veut dire que nous aurons un modèle d'objets de domaine très simple avec quelques liens d'héritage. Les opérations sur des objets seront très simples, suffisamment simples pour implémenter le CRUD complet. Voici ci-dessous la description de notre use case et son architecture :

« Nous allons créer un base de données qui contiendra des véhicules d'un client. Ces véhicules peuvent être de trois types : camion, moto et voiture. À l'aide du framework Hibernate, nous allons dans un premier temps sauvegarder le client et ses véhicules dans la base de données, puis modifier leurs informations et en supprimer quelques-unes. Finalement, nous afficherons ce qui reste comme véhicules d'un client dans la console. »

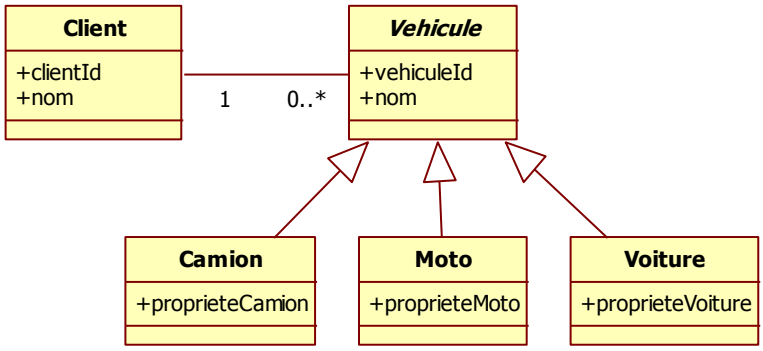


Figure 14. Architecture de l'expérimentation.

Comme vous pouvez le constater le use case est très simpliste et c'est voulu, car notre objectif n'étant pas d'implémenter toutes les situations possibles et imaginables de mapping, mais plutôt d'implémenter une situation habituelle qui s'appliquerait à une grande majorité des applications.

Fonctionnalités présentes dans le use case :

- Héritage
- CRUD

Le use-case est unique, mais pas son type d'implémentation. L'implémentation sera modifiée pour chaque stratégie de mapping, ce qui nous garantit d'avoir exactement une trace par stratégie.

Nous avons choisi l'association *OneToMany* entre le *Client* et le *Vehicule.* Cette dernière possède sa propre annotation ***@OneToMany*** du côté client et *ManyToOne* du côté véhicule. Voici ci-dessous le code exact que nous avons utilisé.

```
import javax.persistence.ManyToOne;

@Entity @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public abstract class Vehicule {

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int vehiculeId;
    private String nom;
    @ManyToOne
    private Client client;
    public Client getClient() {
        return client;
    }
```

Figure 15. Annotation *ManyToOne.* Classe *Vehicule.*

```
import javax.persistence.OneToMany;

@Entity
public class Client {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int clientId;
    private String nom;
    @OneToMany
    private Collection<Vehicule> listeVehicules = new ArrayList<Vehicule>();
    public int getClientId() {
        return clientId;
    }
```

Figure 16. Annotation *ManyToOne.* Classe *Client.*

En ce qui concerne les stratégies de mapping, avec Hibernate c'est la classe-mère V*ehicule* qui doit être réadaptée à chaque fois que l'on change de stratégie.

```
@Entity @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public abstract class Vehicule {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int vehiculeId;
```

Figure 17. Annotation de la stratégie *SINGLE TABLE* de la classe *Vehicule*.

```
@Entity @Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Vehicule {
    @Id @GeneratedValue(strategy=GenerationType.TABLE)
    private int vehiculeId;
```

Figure 18. Annotation de la stratégie *TABLE PER CLASS* de la classe *Vehicule*.

```java
@Entity @Inheritance(strategy=InheritanceType.JOINED)
public abstract class Vehicule {
    @Id @GeneratedValue(strategy=GenerationType.TABLE)
    private int vehiculeId;
```

Figure 19. Annotation de la stratégie JOINED de la classe *Vehicule*.

La stratégie de génération de l'identifiant doit être changée aussi comme vous avez pu le remarquer. Hibernate n'accepte pas la stratégie de génération d'identifiant *AUTO* pour les stratégies de mapping TABLE PER CLASS et JOINED.

4. Technique de la mise en œuvre de l'expérimentation

La mise en œuvre est divisée en deux phases :

1. Génération de traces d'exécution
2. Manipulation des informations obtenues avec l'outil d'analyse de traces DDRA

Le principe est très simple, nous allons d'abords télécharger le code source Hibernate (version 4.1.4) depuis la plateforme en ligne GitHub. A l'aide de Gradle[4], nous allons créer des projets Eclipse, ce qui rendra le code source Hibernate exploitable pour la future instrumentation. Une fois cela fait, nous générerons les archives instrumentés, toujours avec Gradle, et exécuterons des projets (nos use cases, qui utiliseront cette fois-ci la version Hibernate instrumentée) afin d'obtenir six traces, que nous examinerons dans la phase deux.

La phase deux est centrée autour de DDRA et des outils de présentation tels que Treemapper, TreeMap et le tableur Excel. Les fonctionnalités de DDRA nous fourniront des fichiers Excel, que nous allons manipuler avec Treemapper et TreeMap pour une meilleur présentation. Nous utiliserons aussi des formules Excel pour le traitement de certaines informations. Voici ci-dessous le workflow de notre expérimentation.

4 Outil de gestion de cycle de vie des projets Java.

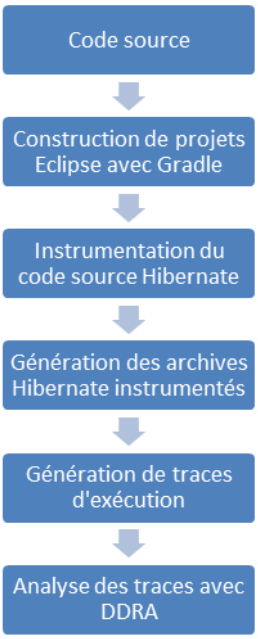


Figure 20. Technique de la mise œuvre de l'expérimentation

5. Technique d'analyse de résultats

L'analyse de résultats est entièrement basée sur des informations contenues dans les traces générées. Ces traces ne sont pas exploitables telles quelles, car une trace en soi n'est qu'un fichier texte avec un nombre de lignes compris entre cent et cent cinquante mille[5], ce qui est un nombre relativement faible pour un framework. Cela s'explique par le fait que Hibernate n'est un pas un framework de métier, son objectif est de supporter des applications de métier lorsqu'elles interagissent avec des bases de données relationnelles.

Étudions é présent le template de la trace, qu'est-ce qu'elle contient comme information ?

Entrée dans une méthode

[staticPackageName] [staticClassName] [packageName] [className] '[' [threadNumber] ']' [methodSignature] 'AS' [returnedType] '[' [timeStmp] ']' [parameterValues]

[staticPackageName]	nom complet du package de la classe statique correspondant à la méthode exécutée.
[staticClassName]	nom de la classe statique correspondant à la méthode exécutée.
[packageName]	nom complet du package de la classe dynamique correspondant à la méthode exécutée.
[className]	nom de la classe dynamique correspondant à la méthode exécutée.
[threadNumber]	numéro du thread dans lequel la méthode s'exécute.
[methodSignature]	nom de la méthode suivi du type complet des paramètres placés entre parenthèses, dans l'ordre et séparés par des virgules.
[returnedType]	type retourné par la méthode.
[timeStmp]	timeStamp de l'exécution de la méthode (en milisecondes écoulées depuis une référence absolue (par exemple 01.01.1901).
[parameterValues]	valeurs des paramètres de types primitif, dans l'ordre des déclaration de paramètres et séparées par des virgules. Si une valeur n'est pas de type primitif, elle est remplacée dans la liste pas le caractère underscode '_'.

Sortie de la méthode

'END' [staticPackageName] [staticClassName] [packageName] [className] '[' [threadNumber] ']' [methodSignature] 'AS' [returnedType] '[' [timeStmp] ']'

Figure 21. Template de la trace.

Et voici un exemple de la trace de la stratégie JOINED :

```
org.hibernate.type CharacterType org.hibernate.type CharacterType [1] getRegistrationKeys() AS java.lang.String[] [1350334603142]
org.hibernate.type CharacterType org.hibernate.type CharacterType [1] getName() AS java.lang.String [1350334603142]
END org.hibernate.type CharacterType org.hibernate.type CharacterType [1] getName() AS java.lang.String [1350334603142]
END org.hibernate.type CharacterType org.hibernate.type CharacterType [1] getRegistrationKeys() AS java.lang.String[] [1350334603142]
org.hibernate.type CharacterType org.hibernate.type CharacterType [1] getRegistrationKeys() AS java.lang.String[] [1350334603142]
org.hibernate.type CharacterType org.hibernate.type CharacterType [1] getName() AS java.lang.String [1350334603142]
END org.hibernate.type CharacterType org.hibernate.type CharacterType [1] getName() AS java.lang.String [1350334603142]
END org.hibernate.type CharacterType org.hibernate.type CharacterType [1] getRegistrationKeys() AS java.lang.String[] [1350334603142]
```

Figure 22. Exemple de la trace.

Comme vous le voyez, tirer quelconque information de ce format peut s'avérer être une tâche herculéenne. Il nous faut un outil qui doit être capable dans un premier temps d'analyser cette information brute, puis d'extraire des informations recherchées par l'utilisateur.

5 Ce nombre est bien évidemment propre à nos use case.

5.1.1 Outil d'analyse de traces DDRA

DDRA est une application développée dans l'équipe de professeur Dugerdil de la Haute École de Gestion à Genève. Ses fonctionnalités nous permettront d'analyser les traces et en tirer des informations suivantes :

- Toutes les classes et les packages utilisées par chaque use case
- L'occurrence temporelle dans la trace des classes étudiées
- Les clusters pour chaque use case
- Les classes dont l'occurrence temporelle est supérieure à N% dans toute la trace[6]

5.1.2 Treemapper et TreeMap

Ces logiciels font tous les deux partie de la famille des outils de représentation qui nous fournirons des images très parlantes regroupant les classes et les packages utilisées dans nos use cases. La particularité de Treemapper et TreeMap réside dans le fait qu'ils prennent en input des fichiers d'extensions, respectivement .csv et .xls. Nous avons beaucoup travaillé sur avec ces deux extensions de fichiers et nous cherchions justement un outil qui puisse donner une meilleur visualisation de nos tableaux. Ces deux outils nous ont donné beaucoup de satisfaction.

5.1.3 PLSQL Developer, Excel

DDRA nécessite une connexion à une base de données pour garder les informations des traces à analyser. Nous avons opté pour la base de données Oracle 10g Express Edition. Certaines informations ont été tirées directement depuis la base et exportés sous format CSV, pour ce faire nous avons utilisé PLSQL Developer.

Les résultats exportés depuis DDRA et PLSQL ne sont pas toujours très exploitables, nous avons beaucoup utilisé Excel et ses formules pour rendre nos résultats plus parlants.

6 Ce réglage nous permettra de détecter des classes qui sont appelées très peu ou trop souvent.

6. Instrumentation du code source Hibernate

6.1 Plateforme GitHub

Au moment d'écriture de ce travail, le code source d'Hibernate peut être téléchargé depuis le site web officiel de GitHub, qui est une plateforme de partage de codes sources pour de nombreuses applications. On y trouve les toutes dernières versions, les commentaires avec une modeste marche à suivre pour créer des projets. Hibernate est disponible sur GitHub sous forme d'un dossier zip ou si l'on le souhaite, on peut même télécharger des fichier/dossiers séparément. Il y a aussi une autre possibilité, c'est de télécharger le client GitHub qui s'installe sur votre ordinateur et permet d'avoir une agréable interface graphique pour gérer vos codes sources.

Sans installer le client GitHub, nous allons tout simplement télécharger Hibernate sous forme d'un fichier zip (la version utilisée dans le cadre de ce travail est la 4.1.4 Final). Le fichier zip contient toutes les classes d'Hibernate, mais cela ne nous suffit malheureusement pas. Nous devons construire de tout cela des projets Eclipse pour pouvoir utiliser notre Instrumentor pour instrumenter le code source Hibernate. Une fois cela est fait, nous générerons des fichiers .jar instrumentés que nous allons utiliser dans un vrai projet. Et qui pour finir nous génèrera la trace d'exécution Hibernate.

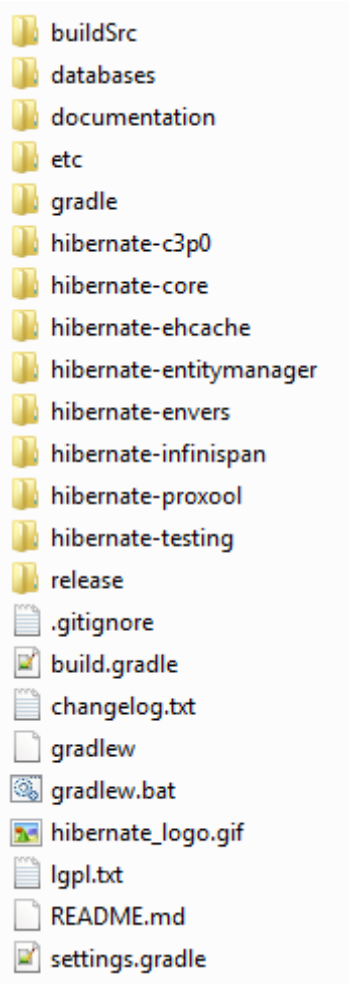


Figure 23. Code source Hibernate.

Dans la figure ci-dessus il y a deux répertoires qui nous intéressent, ce sont *hibernate-core* et *hibernate-entitymanager,* car les archives jar qui portent les mêmes noms font parties des librairies obligatoires à l'implémentation de Hibernate. Il y a aussi deux autres qui n'y figurent pas et que nous allons aussi instrumenter : *hibernate-jpa* et *hibernate-commons-annotaions.* La stratégie de choix des librairies à instrumenter est assez simple, nous avons consulté le répertoire « required » de Hibernate et nous avons choisi les archives qui contiennent le nom « *hibernate-* ».

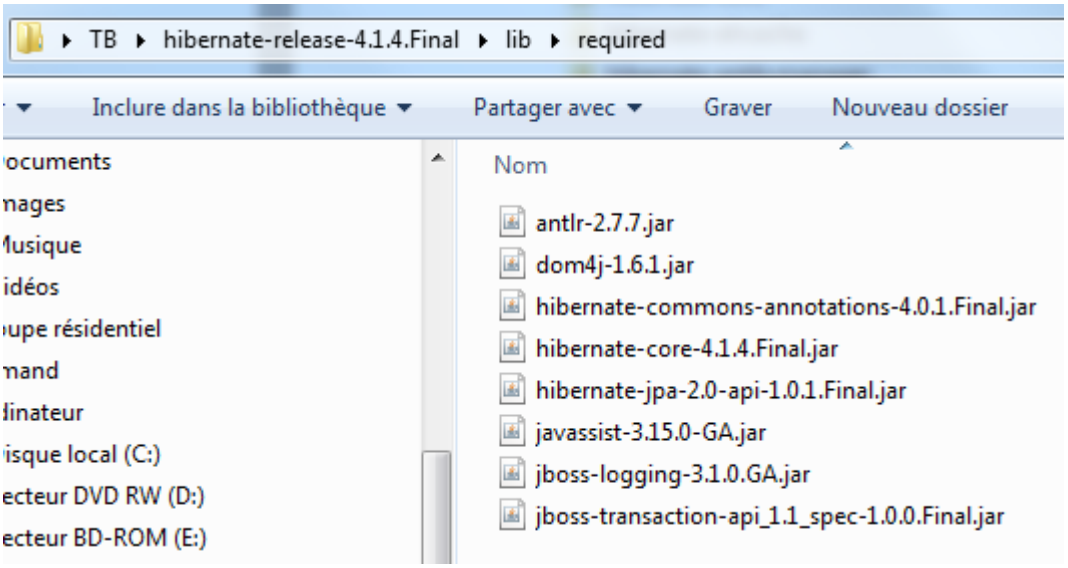


Figure 24. Librairies obligatoires du framework Hibernate.

Vous avez bien remarqué que dans le code source les librairies *hibernate-jpa* et *hibernate-commons-annotation* ne figurent pas. Il faut télécharger le code source de ces derniers depuis de nombreux site web qui le propose. Attention quand même à ne pas télécharger les mauvaises versions (consulter la figure ci-dessus).

6.2 Gradle, outil de construction de projets

Pour construire donc des projets, oui c'est bien des projets et pas un seul, car Hibernate est composé de plusieurs fichiers .jar, donc un projet par le fichier jar, nous utiliserons un outil qui s'appelle Gradle. Gradle est un logiciel de construction de projets Java. En deux mots, nous en avons besoin pour construire nos projets Eclipse et générer des .jar's instrumentés car le code source brute téléchargé depuis GitHub n'est pas exploitable directement. L'apprentissage ne posera pas de problèmes pour ceux qui connaissent déjà Maven, car Gradle suit plus ou moins les mêmes principes. Alors que pour des gens non-expérimentés, il va falloir lire un peu la documentation (disponible sur le site officiel) et s'habituer à utiliser la ligne de commande, car une grande partie de manipulations se passe là-dedans.

Le code source Hibernate vient avec un fichier « *gradlew* » (w pour Wrapper) qui nous permet d'utiliser des commandes Gradle sans l'installer sur la machine. Comment tout cela fonctionne ? La marche à suivre est la suivante. Premièrement, ouvrir la console Windows à la racine du code source téléchargé et lancer les deux commandes suivantes, l'une après l'autre :

gradlew clean build

gradlew eclipse

Chacune des commandes doit être terminée par un « SUCCESSFULL BUILD » dans la console. Une fois cela est fait, le code Hibernate est prêt pour être instrumenté. Nous allons donc importer des projets « hibernate-core » et « hibernate-entitymanager » dans notre workspace Eclipse.

Si le plugin Eclipse d'Instrumentor est installé avec succès nous pouvons procéder donc à l'instrumentation de ces deux projets. Pour ce faire il suffit de clic droit sur un package (ou le projet en entier) et de choisir l'option *« Instrument »*. Si l'instrumentation a été effectuée avec succès, le message comme ci-dessous doit apparaître :

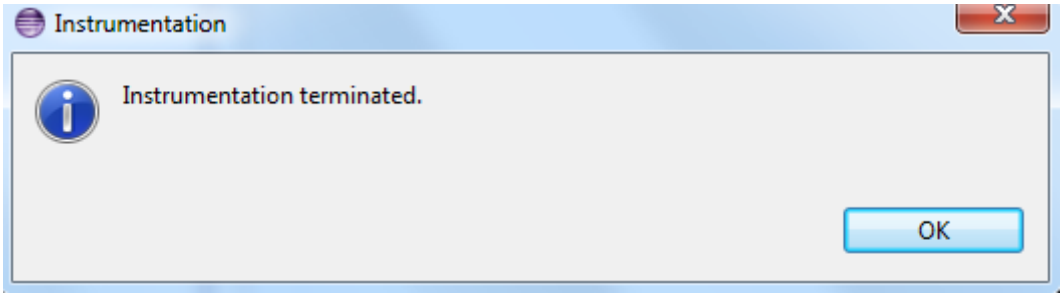


Figure 25. Message de confirmation de l'instrumentation.

Les packages que nous allons instrumenter sont entourés en rouge, ce sont des packages qui contiennent des classes .java :

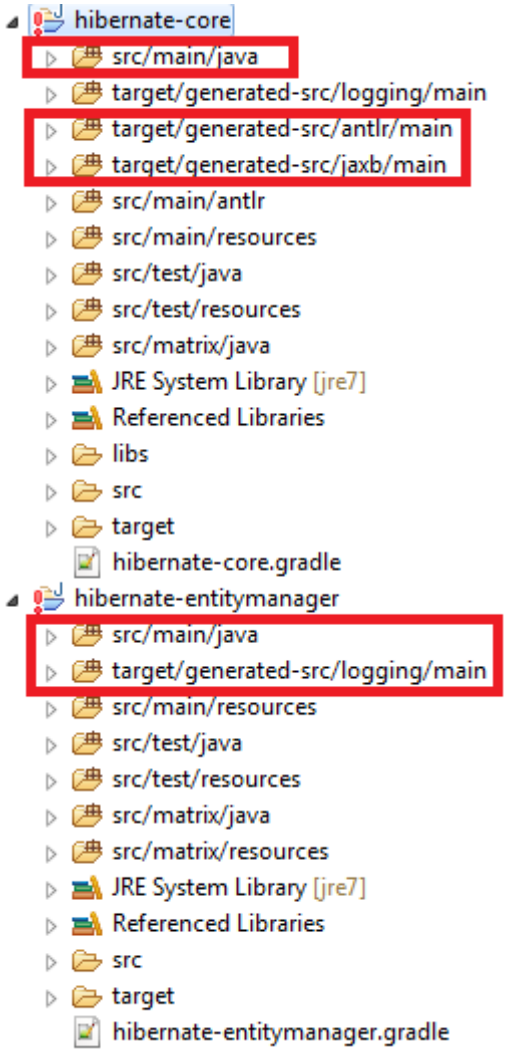


Figure 26. Les packages Hibernate à instrumenter.

Afin que l'instrumentation fonctionne, il faut inclure le fichier .jar « *intrumentor-runtime* » dans le BuildPath de chacun des deux projets. Il faut aussi l'indiquer à Gradle que nous allons utiliser un jar extérieure pour ces deux projets. Pour ce faire, il faut d'abord créer un répertoire libs à la racine des deux projets et y copier-coller le fichier .jar. Une fois cela est fait, il faut ajouter la ligne suivante dans les fichiers *hibernate-core.gradle* et *hibernate-entitymanager.gradle* :

```
dependencies {
    compile( libraries.jta )
    compile( libraries.dom4j )
    compile( libraries.commons_annotations )
    compile( libraries.jpa )
    compile( libraries.javassist )
    compile( libraries.antlr )
    compile files('libs/ch.hesge.csim2.instrumentor.runtime_1.0.0.jar')
    antlr( libraries.antlr )
```

Figure 27. Ajout d'une nouvelle dépendance *instrumentor-runtime* avec Gradle.

Et finalement cette commande gradle ci-dessous va générer des fichiers .jar instrumentés :

gradlew clean install

Cette commande a pour effet de générer un nouveau répertoire dans chacun des projets Hibernate (core et entitymanager). Ce répertoire s'appelle *target/libs* et il contient deux fichiers .jar, un qui contient le code source du projet et l'autre les classes compilées, c'est le deuxième qui nous intéresse.

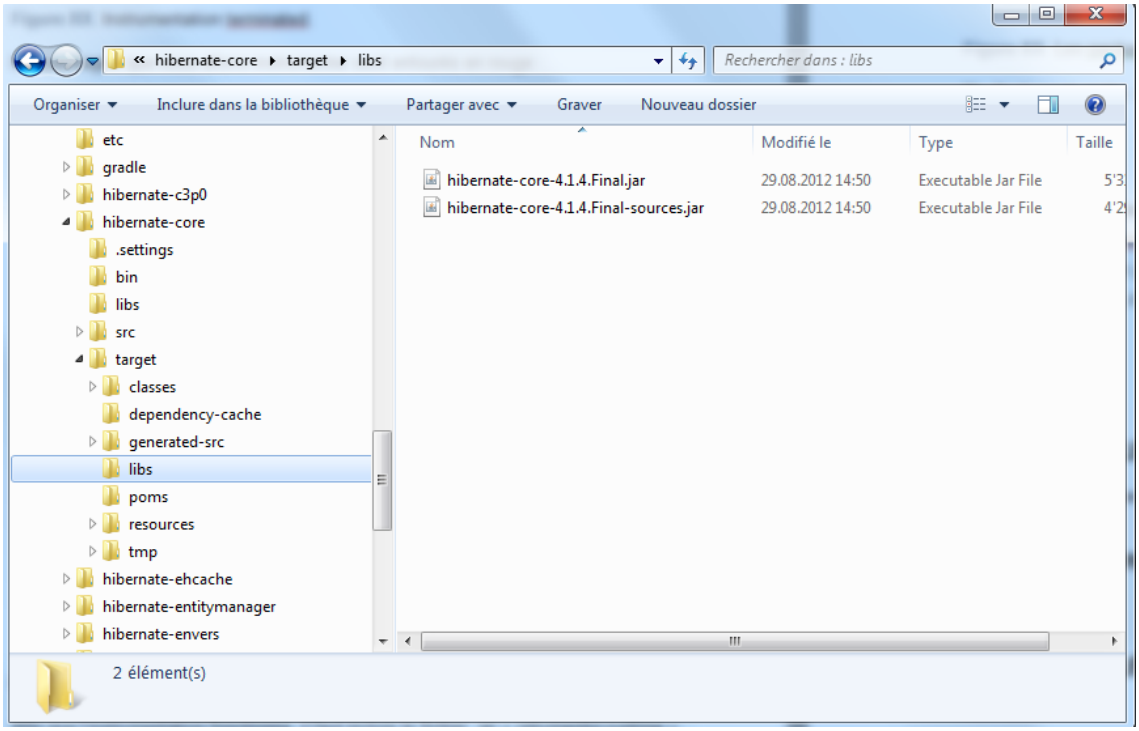


Figure 28. Répertoire *target/libs*.

7. Analyse de traces

Suite à l'instrumentation du code Hibernate, six traces ont été générées, une pour chacune des trois stratégies de mapping, puis une pour chaque type de liaison entre les classes. Dans un premier temps, nous allons étudier les données dans l'ensemble. Nous étudierons le nombre de classes au total, le pourcentage des classes utilisées et les différents modules impliquées. Nous allons ensuite nous intéresser aux classes propres de nos use cases et déterminer les éléments suivants :

- Les couches et les packages Hibernate auxquelles elles appartiennent
- Ses responsabilités
- Leur occurrence temporaire dans la trace

Ces éléments nous permettrons de déterminer les parties de l'architecture Hibernate propre à chaque stratégie, ainsi que leur comportement spécifique par rapport aux autres stratégies.

Dans un deuxième temps, nous allons nous intéresser aux classes qui collaborent fortement entre elles afin de réaliser une des trois stratégies (ces classes forment ce qu'on appelle des clusters). Nous ferons ensuite le lien entre les clusters trouvés afin de déterminer s'ils sont différents ou non suite au changement de stratégies de mapping Cela nous permettra de déterminer la partie de l'architecture la plus impliquée à la réalisation de mapping minimal Hibernate.

Finalement nous nous intéresserons aux classes les plus appelées de chaque stratégie, ce qui nous permettra de compléter l'architecture interne de mapping Hibernate.

Une fois cela fait, nous pourrons « dessiner » l'architecture interne nécessaire à la réalisation de nos use cases et déterminer ainsi le cœur fonctionnel de mapping Hibernate.

7.1 Informations générales sur les traces

Nous nous sommes aperçu, après un bref analyse de traces, que la libraire *hibernate-entitymanager* n'est pas du tout utilisée dans le mapping Hibernate. Nous nous concentrerons donc sur la librairie *hibernate-core, hibernate-jpa* et *hibernate-common-annotations.*

Voici ci-dessous quelques informations sur les traces :

	SINGLE TABLE	TABLE PER CLASS	JOINED
number of calls processed	63853	63502	69503
number of function found	63852	63501	69502
number of procedures found	1	1	1
number of classes involved	402	411	414
end sequence	63854	63503	69504

Tableau 1. Informations générales sur les traces.

7.2 Packages utilisés dans le mapping Hibernate

Afin d'étudier l'architecture interne Hibernate nous avons besoin de connaître les classes et les packages utilisées lors de l'implémentation des trois stratégies de mapping. Nous allons nous intéresser, dans un premier temps, plus aux packages qu'aux classes elles-mêmes, afin d'avoir une vue plutôt macro sur l'ensemble. Nous pourrons ainsi tirer nos premières conclusions sur le nombre de packages impliqués dans l'implémentation par rapport au nombre total mis à disposition par le framework Hibernate. Cela nous permettrait de savoir à quel point l'architecture Hibernate est impliquée dans la réalisation de cette fonctionnalité.

Les annexes 1, 2 et 3 montrent respectivement les packages utilisés (en vert) des stratégies SINGLE TABLE, TABLE PER CLASS et JOINED. Les packages en rouge sont des packages non-utilisés lors de l'implémentation des use cases. Le chiffre à droite du nom de package représente le nombre de classes appelées dans la trace, qui font partie de ce même package.

Les annexes 4, 5 et 6 montrent la même information que les trois précédents, mais avec une métrique en plus. La taille de package représente le nombre de fois que les classes de ce même package ont été appelées. Autrement dit si un package A est plus grand que le B, cela veut dire que les classes de A ont été plus souvent sollicitées à l'exécution que les classes de B.

7.2.1 Observations

Nombre total de packages : 178

Nombre de packages utilisés : 72

Pourcentage d'utilisation : 40%

Notre première constatation est la suivante : presque la moitié de tous modules Hibernate sont impliqués dans la réalisation de la fonctionnalité de mapping.

Il est important de préciser que cette représentation ne veut surtout pas dire que ces packages sont dédiées uniquement à l'implémentation des trois stratégies de mapping. Nous ne pouvons pas non plus confirmer à quel point ces packages sont impliqués. Est-ce que l'on fait appel à une, deux, toutes les classes d'un des packages en vert ? Nous ne pouvons pas non plus dire à quelle fréquence et à quel moment, (plutôt au début de l'implémentation ou vers la fin) les appels sont effectués. La seule chose dont on peut être sûr, c'est que à un moment donné, Hibernate fait appel à une méthode de la classe qui se trouve dans un de ces packages en vert.

Le pourcentage d'utilisation de packages nous indique aussi que le mapping est plutôt une fonctionnalité majeure, dont l'implémentation est repartie entre la plus part des modules Hibernate. N'oublions pas que nos use cases n'exploitent pas en entier la puissance de mapping Hibernate, mais uniquement le strict minimum. Et ce strict minimum fait appel déjà à plus de 40% des ressources propres à Hibernate (pour rappel nous avons instrumenté uniquement les packages qui commencent par « *hibernate- »)*.

Notre deuxième constatation : les stratégies de mapping Hibernate utilisent toutes les mêmes grands modules de ce framework. Il y a quasiment pas de différences entre elles, si ce n'est que la stratégie SINGLE, qui est la seule à ne pas utiliser *org.hibernate.id.enhanced*. L'information que l'on peut tirer de cette constatation est la suivante : l'architecture fonctionnelle de mapping minimal Hibernate est la même pour les trois stratégies si on se place au niveau des packages. La véritable différence doit apparaître aux niveaux des classes appelées, si ce n'est toujours pas le cas, nous allons devoir descendre encore plus bas dans la hiérarchie et regarder au niveau des méthodes. Et c'est exactement ce que nous allons essayer de découvrir dans le chapitre qui suit.

7.3 Classes propres aux stratégies de mapping Hibernate

Nous avons constaté que la vue package ne suffit pas pour capter l'ampleur de l'architecture fonctionnelle impliquée pour chaque use case. Regardons donc au niveau des classes si les stratégies de mapping Hibernate sont implémentées différemment.

Classes propres à la stratégie SINGLE TABLE	
Package	Classe
org.hibernate.mapping	SingleTableSubclass

Les classes propres à la stratégie TABLE PER CLASS	
Package	Classe
org.hibernate.mapping	DenormalizedTable
org.hibernate.mapping	UnionSubclass
org.hibernate.persister.entity	UnionSubclassEntityPersister

Les classes propres à la stratégie JOINED	
Package	Classe
org.hibernate.cfg	FkSecondPass
org.hibernate.cfg	JoinedSubclassFkSecondPass
org.hibernate.mapping	JoinedSubclass
org.hibernate.persister.entity	JoinedSubclassEntityPersister
org.hibernate.sql	ANSICaseFragment
org.hibernate.sql	CaseFragment

Tableau 2. Classes propres des stratégies de mapping Hibernate.

Pour trouver des résultats ci-dessus nous avons utilisé Microsoft Excel 2010, la marche à suivre peut être consultée en Annexe N° 7.

7.3.1 Observations

Première observations qui saute aux yeux est la faible quantité de classes utilisées par Hibernate pour la stratégie de mapping SINGLE TABLE. Seulement une classe propre pour cette dernière, c'est donc la stratégie la moins spécifique de Hibernate. Pour information, cette stratégie est celle utilisée par défaut en Hibernate.

L'idée de trouver des classes propres à chaque stratégie cache en fait un autre objectif. Ce que nous recherchions avant tout c'est de savoir si les trois stratégies de mapping utilisent plus ou moins les mêmes classes. Si le nombre de classes propres était trop important par rapport au nombre de classes total appelées par la stratégie donnée, nous pourrions affirmer que l'architecture fonctionnelle de cette stratégie est assez indépendante par rapport aux deux autres. Ce qui n'est visiblement pas le cas d'après les résultats ci-dessus. Le nombre total de classes appelées varie entre 402 et 414, alors que le nombre total des classes propres varie entre 1 et 6, ce qui représente au maximum 1.5% des classes appelées d'une stratégie.

La stratégie JOINED est la stratégie la plus gourmande en termes de nombre de classes spécifiques (6) par rapport aux autres stratégies. C'est la stratégie la plus spécifique de toutes les trois.

7.4 Clusters

Dans le cadre de ce travail un cluster est un ensemble de classes qui figurent ensemble dans des segments différents dans la trace et qui collaborent fortement pour livrer une fonctionnalité. Pour rappel, un segment est une séquence d'appels des méthodes des classes. Pour mieux comprendre le concept de cluster, prenons un exemple d'une trace qui contient cinq segments de taille trois. Imaginons qu'un cluster est formé par deux classes A et B. Cela veut dire que les méthodes de ces classes sont souvent évoquées conjointement dans les segments tout au long de la trace.

Ce « souvent » est en fait un réglage DDRA que l'on peut paramétrer, il s'agit de la taille du segment. Si la taille est grande, plus de méthodes contiendraient les segments et donc potentiellement plus il y aurait de classes dont les méthodes apparaissent ensemble dans le même segment et vice-versa. La taille n'est pas paramétrable directement, DDRA utilise un coefficient que l'on appelle le « Multiplier », qui sert dans le calcul du nombre de segments. Voici la formule pour déterminer le nombre de segments dans une trace.

Nombre de segments = Multiplier * Nombre de classes dans la trace

Plus de segments il y a dans la trace, plus ces segments seront petits et vice-versa.

Il se peut bien sûr que les méthodes de nos classes A et B soient appelées séparément dans d'autres segments, ou bien même qu'elles figurent séparément dans d'autres clusters (classe A, par exemple, peut en même temps figurer dans un autre cluster avec une autre classe). L'équipe de Professeur Dugerdil a introduit à cet effet un indicateur de corrélation qui indique le pourcentage de corrélation entre les classes. Pour nos classes A et B, cette corrélation est à 100%, car les méthodes de ces classes apparaissent tout le temps dans les mêmes segments. Nous utiliserons cet indicateur dans le chapitre consacré à l'occurrence temporelle des classes des mêmes clusters qui couplé à cette dernière, nous permettra de savoir avec qui une classe collabore le plus souvent dans le cas où, par exemple, elle figure dans plusieurs clusters.

Pour connaître l'architecture Hibernate impliquée lors de l'exécution de nos use cases, nous allons dans un premier temps déterminer quelles sont les classes qui collaborent le plus activement entre elles pour la livraison de la fonctionnalité de mapping. Ces classes seront déterminées grâce à la fonctionnalité de recherche de clusters de DDRA.

Dans un deuxième temps nous allons consulter les occurrences temporelles des classes constituant les clusters trouvés afin de comprendre à quel moment de l'exécution ces collaborations sont les plus fréquentes.

Finalement nous nous intéresserons aux classes les plus sollicitées à l'exécution de nos use cases. En plus des clusters, cela nous permettra de déterminer le noyau fonctionnel de l'architecture Hibernate nécessaire au mapping des classes.

7.4.1 Clusters de mapping Hibernate

Il est important de savoir que les clusters dépendent fortement du nombre de segments présents dans la trace (cf. le chapitre précèdent). Les résultats ci-dessous ont été trouvé avec un Multiplier optimal qui est égale à 16 (conseillé par Professeur Ph. Dugerdil). Les couleurs servent à indiquer les clusters communs aux trois stratégies de mapping.

Informations générales			
	Stratégie SINGLE	Stratégie TABLE PER CLASS	Stratégie JOINED
Nb segments dans la trace	6432	6576	6624
Taille de segments	9	9	10
Nb Clusters	11	14	17
Nb Clusters communs avec autres stratégies	11	14	14
Nb Clusters propres	0	0	3

Tableau 3. Informations générales sur les clusters.

Stratégie SINGLE		Stratégie TABLE PER CLASS		Stratégie JOINED	
N° cluster	Classe	N° cluster	Classe	N° cluster	Classe
1	EJB3DTDEntityResolver	10	EJB3DTDEntityResolver	1	EJB3DTDEntityResolver
1	DTDEntityResolver	10	DTDEntityResolver	1	DTDEntityResolver
1	ErrorLogger	10	ErrorLogger	1	ErrorLogger
1	XMLHelper	10	XMLHelper	1	XMLHelper
2	JavaAnnotationReader	12	JavaAnnotationReader	8	JavaAnnotationReader
2	JPAMetadataProvider	12	JPAMetadataProvider	8	JPAMetadataProvider
3	StandardRandomStrategy	6	StandardRandomStrategy	11	StandardRandomStrategy
3	UUIDTypeDescriptor.ToStringTransformer	6	UUIDTypeDescriptor.ToStringTransformer	11	UUIDTypeDescriptor.ToStringTransformer
6	Collections	13	Collections	17	Collections
6	FlushVisitor	13	FlushVisitor	17	FlushVisitor
9	BoundedConcurrentHashMap.HashEntry	11	BoundedConcurrentHashMap.HashEntry	13	BoundedConcurrentHashMap.HashEntry
9	BoundedConcurrentHashMap.Segment	11	BoundedConcurrentHashMap.Segment	13	BoundedConcurrentHashMap.Segment
11	NoFormatImpl	7	NoFormatImpl	12	NoFormatImpl
11	DatabaseExporter	7	DatabaseExporter	12	DatabaseExporter

5	JavassistLazyInitializer	4	JavassistLazyInitializer	10	JavassistLazyInitializer
5	JavassistProxyFactory	4	JavassistProxyFactory	10	JavassistProxyFactory
10	StandardAnsiSqlAggregationFunctions	9	StandardAnsiSqlAggregationFunctions		
10	TypeInfoExtracter	9	TypeInfoExtracter		
7	JavaMetadataProvider			7	JavaMetadataProvider
7	XMLContext			7	XMLContext
		2	ConcurrentReferenceHashMap.HashIterator	9	ConcurrentReferenceHashMap.HashIterator
		2	ConcurrentReferenceHashMap.Values	9	ConcurrentReferenceHashMap.Values
		5	SqlExceptionHelper	2	SqlExceptionHelper
		5	DatabaseExporter	2	DatabaseExporter
		8	AbstractReturningWork	14	AbstractReturningWork
		8	WorkExecutor	14	WorkExecutor
		14	JdbcTransaction	15	JdbcTransaction
		14	AbstractTransactionImpl	15	AbstractTransactionImpl
				3	EventListenerGroupImpl
				3	EventListenerRegistryImpl
				5	StandardAnsiSqlAggregationFunctions
				5	StandardSQLFunction
				6	DirectPropertyAccessor
				6	PropertyAccessorFactory
4	Version	1	Version	4	Version
4	Version	1	Version	4	Version
4	Log_.logger	1	Log_.logger	4	Log_.logger
4	LoggerFactory	1	LoggerFactory	4	LoggerFactory
8	Camion	3	Camion	16	Camion
8	Moto	3	Moto	16	Moto

Tableau 4. Clusters de mapping minimal d'Hibernate.

7.4.2 Observations

Avant d'avancer dans nos observations, nous pouvons tout de suite exclure de notre analyse les deux derniers clusters. Le Logger, comme son nom l'indique, est un outil qui permet d'écrire dans le log, par conséquent, il ne nous intéresse pas car il est présent d'office lorsqu'on utilise Hibernate. En ce qui concerne le cluster Camion-Moto, ces classes sont propres à notre application, on peut donc aussi les exclure de l'analyse.

Après avoir consulté le code source des classes constituant les clusters trouvés, nous allons dresser un tableau regroupant leurs responsabilités. Nous n'avons choisi que les cluster communs aux trois stratégies.

Classe	Responsabilités

EJB3DTDEntityResolver DTDEntityResolver ErrorLogger XMLHelper	Lecture et mise en cache des fichiers de configuration
JavaAnnotationReader JPAMetadataProvider	Lecture des annotations des classes à persister
StandardRandomStrategy UUIDTypeDescriptor.ToStringTransformer	Mise en œuvre de la stratégie de génération de l'identifiant
Collections FlushVisitor	Persistance des collections d'objets
BoundedConcurrentHashMap.HashEntry BoundedConcurrentHashMap.Segment	Class appropriée par Hibernate, initialement prévue pour Infinispan qui a pour objectif d'exposer des structures de données concurrentes
NoFormatImpl DatabaseExporter	Formatage des requêtes SQL
JavassistLazyInitializer JavassistProxyFactory	Implémentation de Javassist proxy

Tableau 5. Responsabilités des classes des clusters communs aux 3 stratégies de mapping d'Hibernate.

D'après les résultats du premier tableau, nous pouvons constater que Hibernate utilise principalement les mêmes classes pour réaliser nos use cases, qui consistent, pour rappel, à implémenter les stratégies de mapping minimal Hibernate. En effet, sur 16 clusters trouvés, 7 sont communs à toutes les stratégies, ce qui représente quasiment la moitié.

Nous connaissons désormais les collaborations les plus fortes qui sont présentes dans les trois stratégies de mapping Hibernate, les voici ci-dessous.

Package	Classe
org.hibernate.cfg	EJB3DTDEntityResolver
org.hibernate.internal.util.xml	DTDEntityResolver
org.hibernate.internal.util.xml	ErrorLogger
org.hibernate.internal.util.xml	XMLHelper
org.hibernate.annotations.common.reflection.java	JavaAnnotationReader
org.hibernate.cfg.annotations.reflection	JPAMetadataProvider
org.hibernate.id.uuid	StandardRandomStrategy
org.hibernate.type.descriptor.java	UUIDTypeDescriptor.ToStringTransformer
org.hibernate.engine.internal	Collections
org.hibernate.event.internal	FlushVisitor
org.hibernate.internal.util.collections	BoundedConcurrentHashMap.HashEntry
org.hibernate.internal.util.collections	BoundedConcurrentHashMap.Segment
org.hibernate.engine.jdbc.internal	NoFormatImpl
org.hibernate.tool.hbm2ddl	DatabaseExporter
org.hibernate.proxy.pojo.javassist	JavassistLazyInitializer

org.hibernate.proxy.pojo.javassist	JavassistProxyFactory

Tableau 6. Clusters communs aux trois stratégies.

Pouvons-nous à ce stade considérer que les packages suivant constituent le cœur de l'implémentation minimale du mapping Hibernate ? Ce qui nous emmène, en partie, à penser le contraire ce sont les noms de certaines classes et de leurs packages. Le package *org.hiberante.internal.util.collection* n'a rien à voir sémantiquement avec le mapping. En effet, la présence des attributs de type *java.util.Collection* dans notre use case peuvent enduire à penser que les collaborations de cette classe ont quelque chose à voir avec l'implémentation de mapping Hibernate, ce qui n'est pas du tout le cas.

Afin de déterminer le cœur fonctionnel de mapping Hibernate nous avons aussi besoin d'analyser l'occurrence temporelle des classes qui constituent ces clusters. Car les classes ci-dessus peuvent être appelées depuis un même segment seulement une fois dans toute la trace. Ce que nous recherchons ce sont plutôt des classes qui apparaissent régulièrement dans toute la trace ou du moins, qui sont concentrées abondamment au début, au milieu ou vers la fin de la trace. Le fait de trouver ce genre de clusters pourrait nous emmener à la conclusion que ces classes sont vraiment importantes dans la réalisation de mapping Hibernate.

Nous allons donc analyser chaque trace et éliminer les clusters dont l'occurrence temporelle est trop faible.

7.5 Occurrence temporelle

Nous allons de nouveau utiliser DDRA et plus particulièrement sa fonction de recherche d'occurrence temporelle des classes sélectionnées. Avant de procéder à l'analyse temporel de chaque cluster de nos use cases, nous allons exclure de l'affichage les classes dont l'occurrence temporelle est inférieur à 5% par rapport à toute la trace (réglage DDRA). Cela va éliminer d'office les classes qui n'apparaissent que très rarement. Ces classes ne sont pas intéressantes pour nous, car ce que nous recherchons ici ce sont plutôt des classes avec une forte activité temporelle. Les classes qui resteront après l'application du filtre, nous donnerons un indice sur quoi les éléments de l'architecture Hibernate sont le plus concentrées.

L'occurrence temporelle peut nous révéler une autre information très intéressante qui est la distribution du travail lors d'une implémentation d'une fonctionnalité. En diminuant l'occurrence temporelle des classes dans les traces, le nombre total de classes va diminuer. Ce qui serait intéressant de voir, c'est justement l'effet sur le nombre de classes restantes. Si ce nombre est petit,

cela voudrait dire que, dans nos traces, il y a beaucoup de classes intermédiaires qui ne font que déléguer le travail aux autres classes.

Ceci peut être expliqué en observant la répartition du travail, qui est représentée par des appels des méthodes dans la trace et les occurrences temporelles (OT) des classes. Cette répartition peut être homogène ou non. Dans le cas où elle serait non homogène, la plus grande partie du travail serait repartie entre la minorité des classes. En programmation orientée objet cela s'explique par une délégation (appel à une autre méthode depuis une méthode) et des appels répétitifs qui déclenchent une série d'autres appels. Il ne faut par contre pas confondre l'héritage avec la délégation, ce sont des notions différentes. Avec la délégation on confie une tâche à une autre classe, alors qu'avec l'héritage on utilise la méthode déclarée dans les classes étendues.

Le taux de délégation peut être donc observé en diminuant les apparitions des classes tout en observant la quantité des classes restantes. Ce qu'il faut étudier en fait c'est la relation entre l'OT et le pourcentage des classes restantes, qui nous donnera une indication sur le type de distribution de travail.

Distribution plutôt forte : on augmente l'OT et on observe que le % des classes restantes est une minorité.

Distribution proportionnelle : la diminution du pourcentage des classes restantes est proportionnelle à la diminution de l'OT.

Distribution plutôt faible : on augmente l'OT et on observe que le % des classes restantes est une majorité.

Nous avons, pour commencer, diminué l'occurrence temporelle de 5, puis de 10 et de 15%. Le nombre de classes a littéralement fondu après l'application du premier filtre. Voici les résultats :

Stratégies	Nb de classes sans le filtre temporel	Occurrence Temporelle <5%	Occurrence Temporelle <10%	Occurrence Temporelle <15%
SINGLE TABLE	402	139	99	74
TABLE PER CLASS	411	143	101	76
JOINED	414	150	102	79

Tableau 7. Résultats d'application du filtre d'occurrence temporelle.

Plus 65% des classes appelées n'apparaissent que très peu durant l'exécution de nos use cases. Cela veut dire que nos traces sont remplies par des appels des méthodes des classes qui font partie de ces 65%. Il reste donc 35% qui sont une majorité et nous avons augmenté l'OT seulement de 15%. Nous sommes donc en présence d'une distribution plutôt forte.

Est-ce que les classes restantes, la minorité donc, peut être considérée comme un noyau fonctionnel de notre use-case, car ce sont leurs méthodes qui sont sollicitées le plus souvent dans la trace et pas celles de la majorité ? Il nous faut plus d'arguments pour affirmer cela, car nous ne savons toujours pas qui sont ces classes ni leurs responsabilités. Nous allons étudier ces classes plus en détails dans un chapitre ultérieur.

Après avoir appliqué le filtre OT nous avons également observé la situation du côté des clusters. Nous constatons, d'après le tableau ci-dessous, que quasiment toutes les classes des clusters trouvés précédemment ont disparu de notre tableau, cela veut dire que leur occurrence temporelle était inférieure à 5% dans toute la trace[7]. Nous concluons que ces classes disparues ne collaboraient, en grande partie, qu'entre elles.

Stratégie SINGLE		Stratégie TABLE PER CLASS		Stratégie JOINED	
N°cluster	Classe	N°cluster	Classe	N°cluster	Classe
1	JavaAnnotationReader	1	JdbcTransaction	1	DirectPropertyAccessor
1	JPAMetadataProvider	1	AbstractTransactionImpl	1	PropertyAccessorFactory
		2	JavaAnnotationReader	2	JdbcTransaction
		2	JPAMetadataProvider	2	AbstractTransactionImpl
				3	EventListenerGroupImpl
				3	EventListenerRegistryImpl
				4	JavaAnnotationReader
				4	JPAMetadataProvider

Tableau 8. Clusters restant, filtre occurrence temporelle = 5%.

Les classes rarement sollicitées sont désormais éliminées de notre analyse, nous pouvons donc nous concentrer sur les classes restantes et étudier leur occurrence temporelle, afin de comprendre si elles apparaissent en même temps ou non en fonction de la stratégie choisie.

Le cluster en vert étant commun à toutes les stratégies est le plus intéressant à étudier, si les collaborations de ces deux classes sont identiques au niveau temporel dans les trois stratégies, nous pourrons affirmer que ces deux classes contribuent grandement à la réalisation de mapping Hibernate indépendamment de la stratégie choisie.

7 Nous ne sommes pas allés plus loin que 5% pour les clusters.

Nous allons également étudier le cluster en rouge qui est commun aux stratégies TABLE PER CLASS et JOINED. Si on arrive à démontrer que l'occurrence temporelle des classes qui le constituent est semblable pour ces deux stratégies, nous pourrons affirmer que la stratégie JOINED s'appuie sur les mêmes collaborations que la stratégie TABLE PER CLASS, à condition que le cluster commun (vert) soit identique au niveau temporel dans les trois traces.

Pour finir, nous étudierons également les clusters N°1 et N°3 de la stratégie JOINED afin d'avoir un panorama complet de l'architecture fonctionnel de nos use cases.

7.5.1 Cluster commun aux trois stratégies de mapping Hibernate

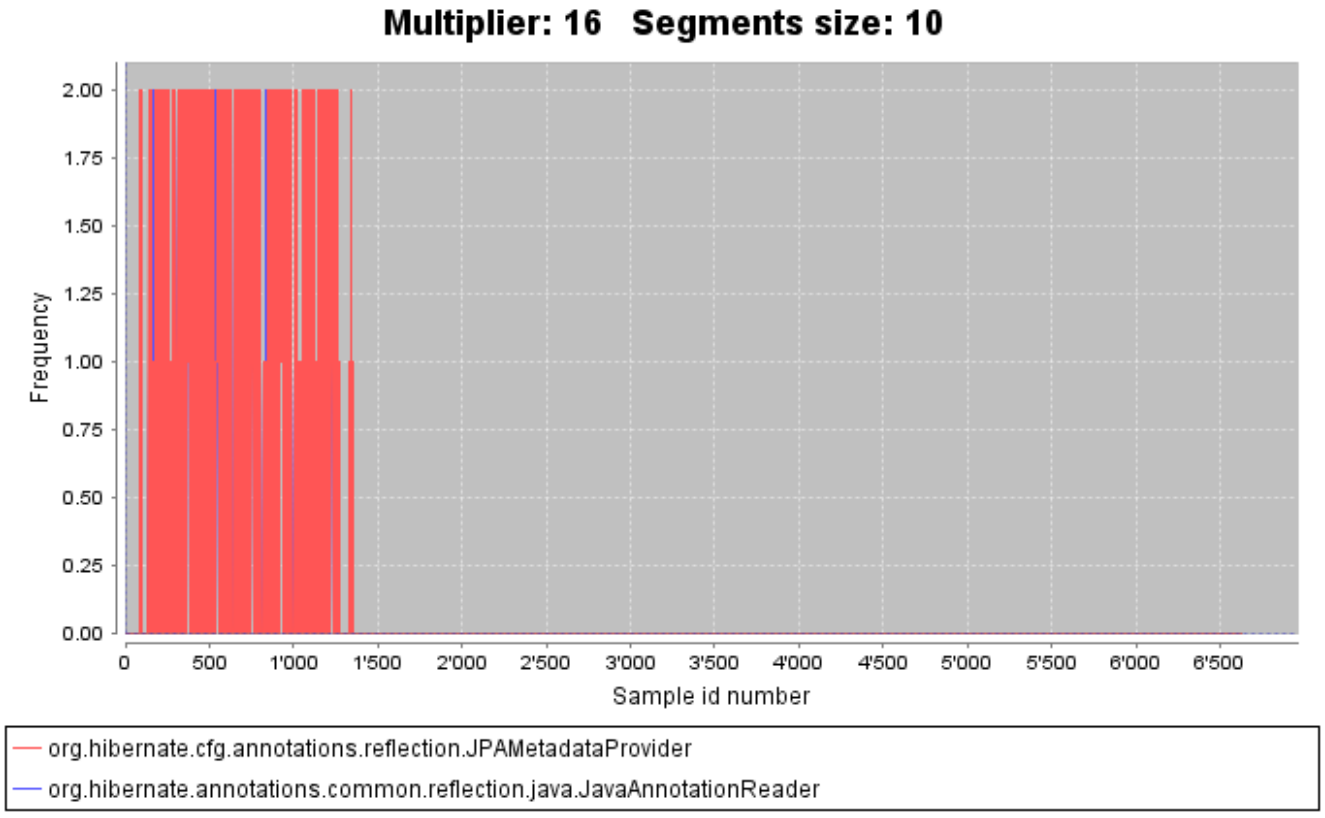
Les graphiques qui suivent doivent se lire de la manière suivante :

Abscisse : numéro de segment.

Ordonnée : fréquence d'appel dans le segment.

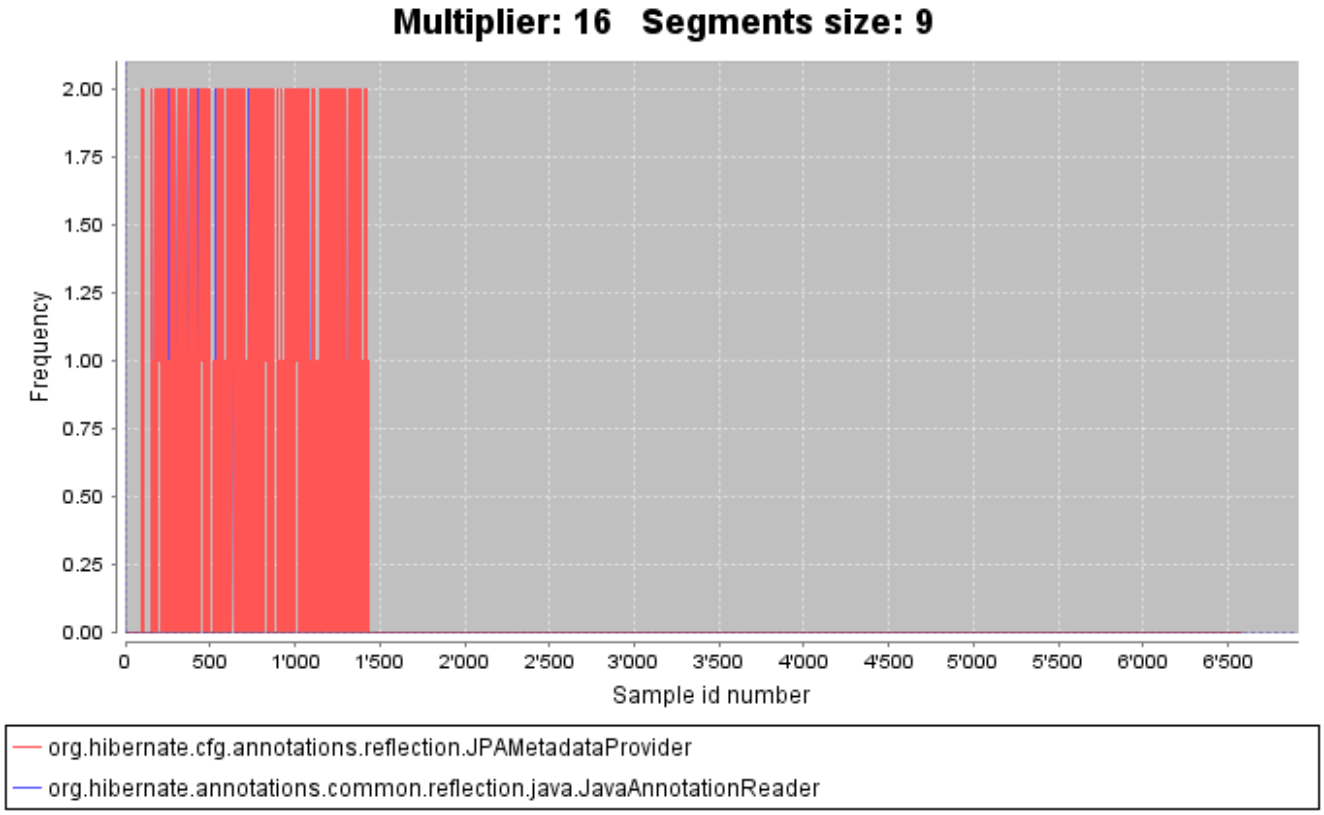
Couleur : chaque classe a sa propre couleur.

org.hibernate.annotations.common.reflection.java	JavaAnnotationReader
org.hibernate.cfg.annotations.reflection	JPAMetadataProvider



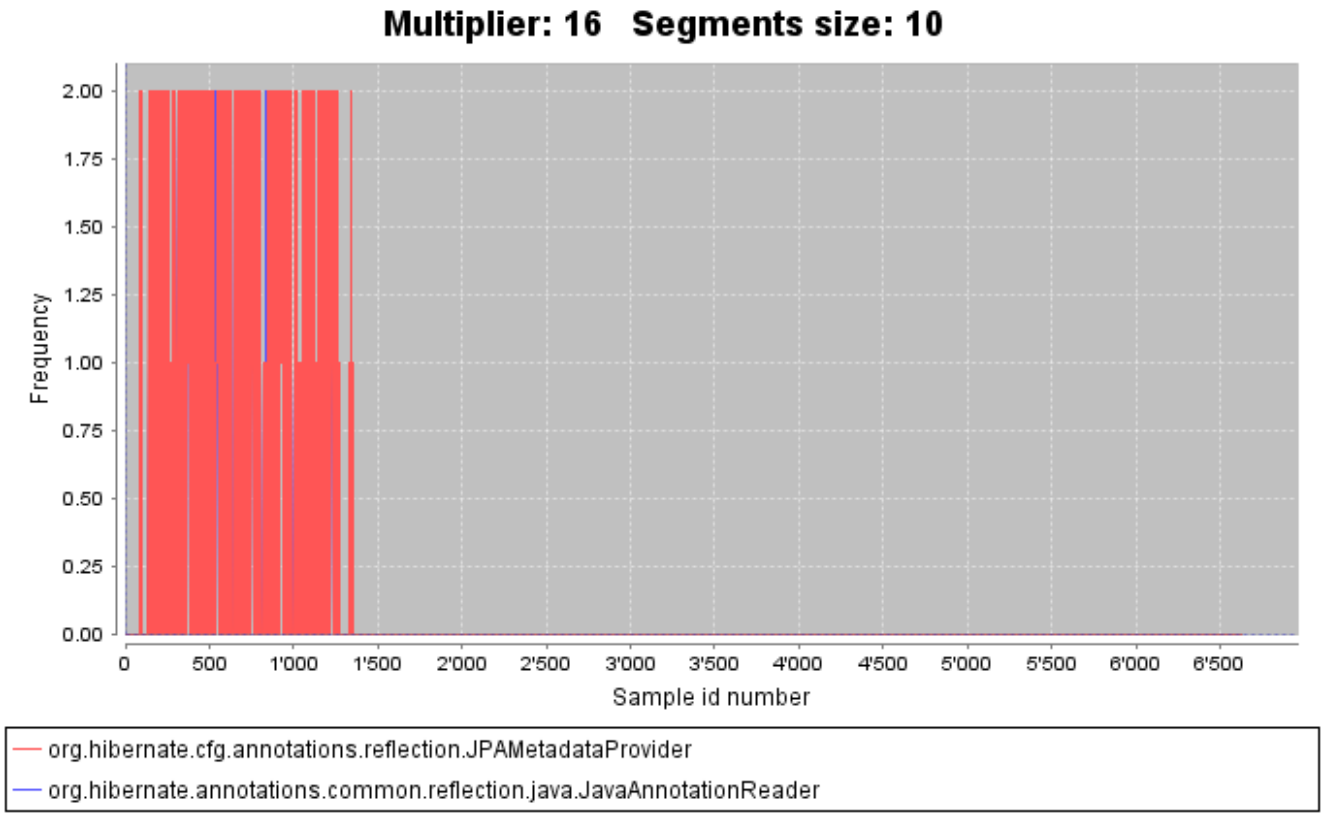
Figures 29. Cluster commun. Occurrence temporelle stratégie SINGLE TABLE.

Corrélation : 92%



Figures 30. Cluster commun. Occurrence temporelle stratégie TABLE PER CLASS.

Corrélation : 92%



Figures 31. Cluster commun. Occurrence temporelle stratégie JOINED.

Corrélation : 94%

7.5.1.1 Responsabilités des classes et observations

org.hibernate.cfg.annotations.reflection.JPAMetadataProvider

JAVADoc description: MetadataProvider aware of the JPA Deployment descriptor.

Afin de comprendre à quoi sert exactement cette classe, nous avons effectué une requête dans la base de données de DDRA et avons extrait les méthodes appelées de cette classe. Voici le pseudo code de la requête :

SELECT signatureMethode FROM nomTrace WHERE classeDynamque et classeStatique ='JPAMetadataProvider'

Sur environs mille appels pour chaque trace, il n'y a que les méthodes *getDefaults()* qui occupe les deux premiers appels et le reste, ce sont des appels de la méthode *getAnnotationReader(java.lang.reflect.AnnotatedElement)*. D'après la sémantique des méthodes, nous pouvons déduire qu'il s'agit du fait que Hibernate parcourt le code de nos use cases et recherche des éléments annotés afin de les mapper par la suite.

org.hibernate.annotations.common.reflection.java.JavaAnnotationReader

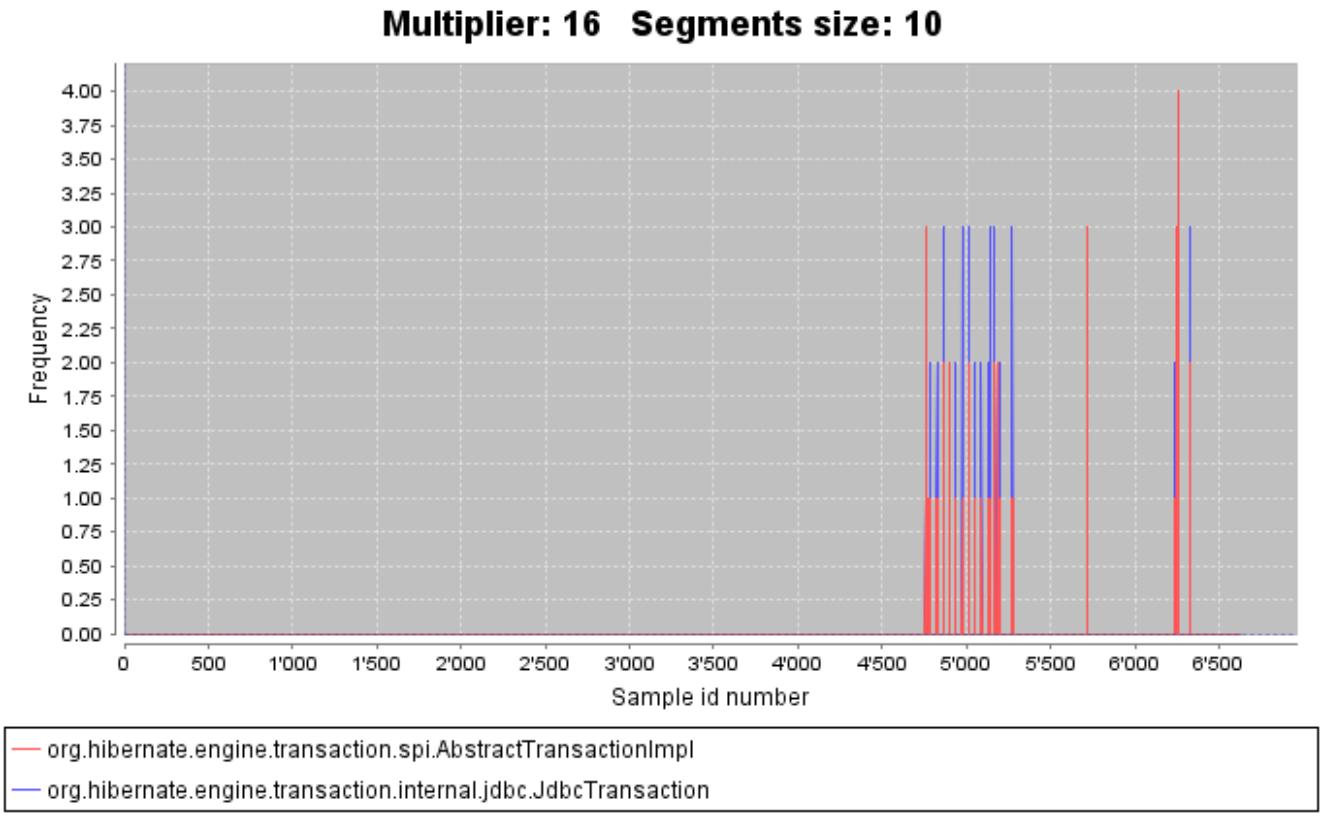
JAVADoc description: Reads standard Java annotations.

Méthodes les plus appelées : *getAnnotation(java.lang.Class)* et *isAnnotationPresent(java.lang.Class).* Tout comme la classe précédente, il s'agit clairement ici d'une sorte de préparation où l'on se renseigne sur comment les classes sont annotées afin de les mapper correctement.

Il n'est donc pas du tout étonnant que ces classes sont surtout sollicitées au début de la trace, car c'est à ce moment qu'il est nécessaire de définir la manière dont les classes devront être mappées par la suite.

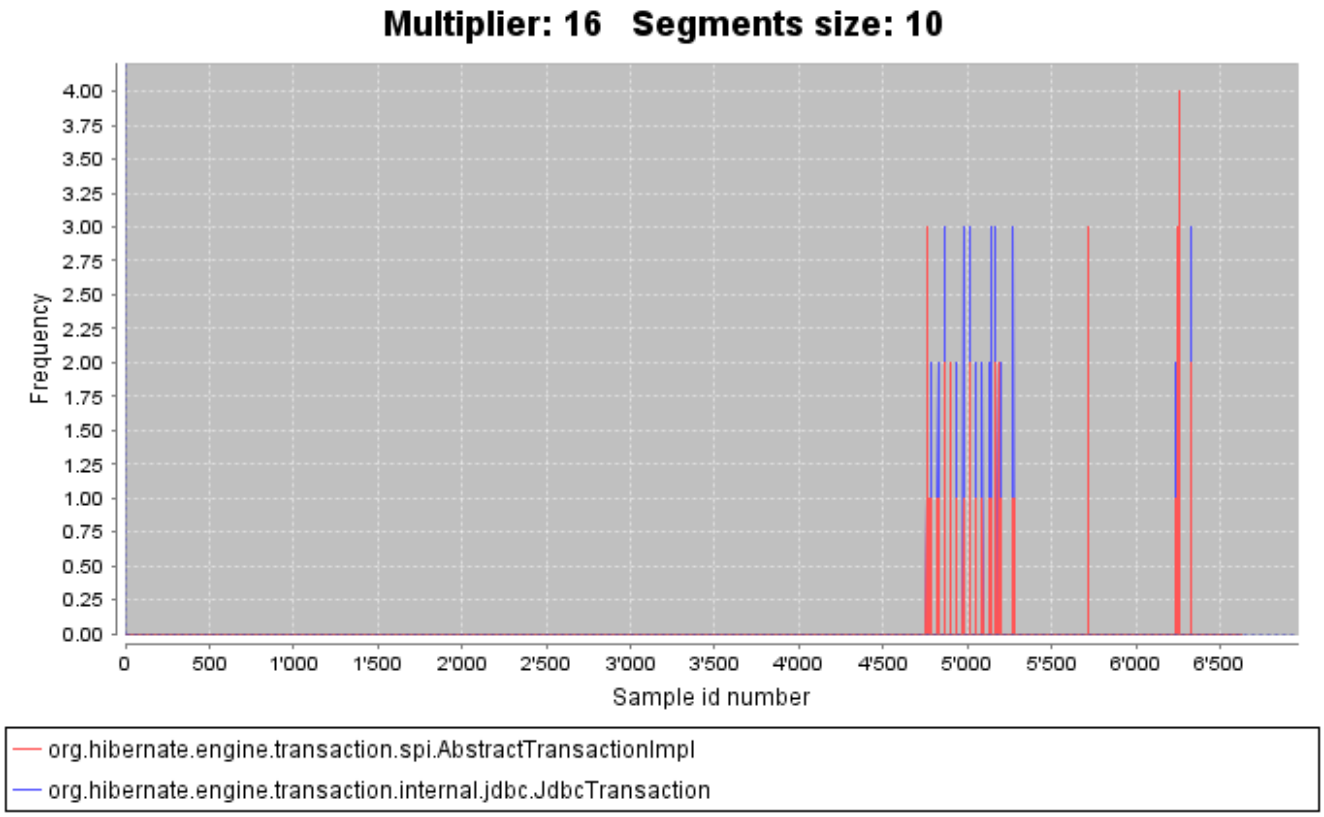
7.5.2 Cluster commun aux stratégies TABLE PER CLASS et JOINED

org.hibernate.engine.transaction.internal.jdbc	JdbcTransaction
org.hibernate.engine.transaction.spi	AbstractTransactionImpl



Figures 32. Cluster commun aux stratégies TABLE PER CLASS et JOINED.

Corrélation : 86%



Figures 33. Cluster commun aux stratégies TABLE PER CLASS et JOINED.

Corrélation : 86%

7.5.2.1 Responsabilités des classes et observations

org.hibernate.engine.transaction.internal.jdbc.JdbcTransaction

JAVADoc description: implementation based on transaction management through a JDBC. This is the default transaction strategy.

org.hibernate.engine.transaction.spi.AbstractTransactionImpl

JAVADoc description: Abstract support for creating implementations.

Après avoir consulté la base de données et les méthodes appelées de ces classes, nous avons constaté qu'il n'y a pas de méthodes qui sont appelées plus que les autres. Nous avons également consulté le code source et avons constaté que *JdbcTransaction* hérite de la classe *AbstractTransactionImpl* et que cette dernière implémente l'interface *TransactionImplementor.* Cette interface est assez bien commentée et nous constatons qu'elle hérite d'une autre interface qui est *Transaction.* Cette dernière est encore mieux décrite :

```
* Defines the contract for abstracting applications from the configured underlying means
of transaction management.
```

```
 * Allows the application to define units of work, while maintaining abstraction from the
underlying transaction
 * implementation (eg. JTA, JDBC).
 * <p/>
 * A transaction is associated with a {@link Session} and is usually initiated by a call
to
 * {@link org.hibernate.Session#beginTransaction()}.  A single session might span multiple
transactions since
 * the notion of a session (a conversation between the application and the datastore) is
of coarser granularity than
 * the notion of a transaction.  However, it is intended that there be at most one
uncommitted transaction associated
 * with a particular {@link Session} at any time.
 * <p/>

 * Implementers are not intended to be thread-safe.
```

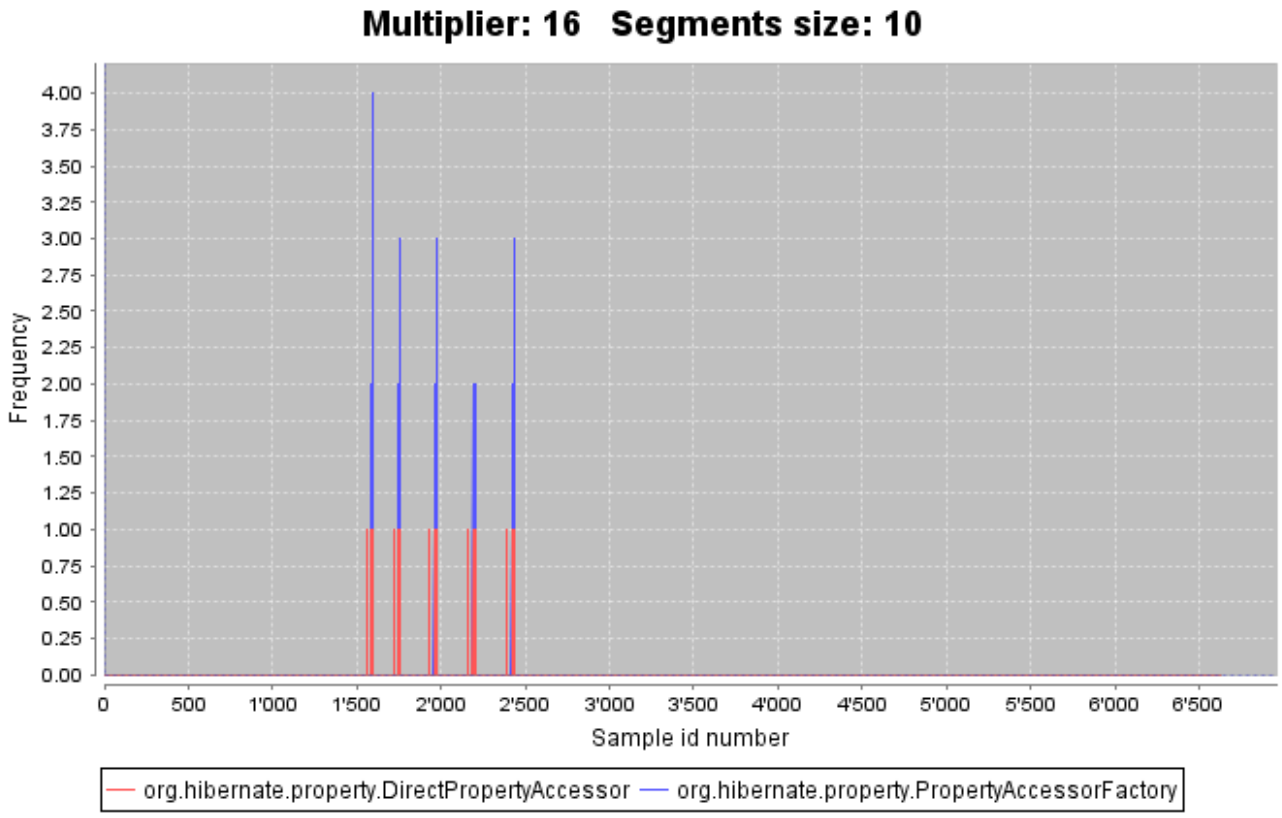
Nous avons déjà vu, dans le chapitre Architecture Hibernate, que l'interface *Transaction* fourni des méthodes qui permettent la synchronisation des données d'une *Session* avec la base de données et tout cela dans la transparence totale pour l'application cliente. Ce qui est étonnant, c'est pourquoi ce cluster n'est pas présent pour la stratégie *SINGLE TABLE* ? Il y a aussi une interface et on communique aussi avec la base de données. La réponse est dans notre manière de calculer les clusters. Rappelez-vous que pour les trouver DDRA s'appuie sur une formule qui détermine le nombre de segments, et les clusters sont très dépendants de ce nombre. Nous avons utilisé le Multiplier = 16 et en le mettant à 14[8], nous retrouvons bel et bien un nouveau cluster qui contient ces deux classes.

Comme les appels se font plutôt vers la fin des traces, cela nous laisse imaginer que les transactions avec la base de données se font plutôt à ce moment.

7.5.3 Clusters propres à la stratégie JOINED

org.hibernate.property	DirectPropertyAccessor
org.hibernate.property	PropertyAccessorFactory

8 Conséquence : moins de segments, segments sont plus grands.



Figures 34. Cluster N°1 propre à la stratégie JOINED.

Corrélation : 88%

7.5.3.1 Responsabilités des classes et observations

org.hibernate.property.DirectPropertyAccessor

JAVADoc description: Accesses fields directly.

org.hibernate.property.PropertyAccessorFactory

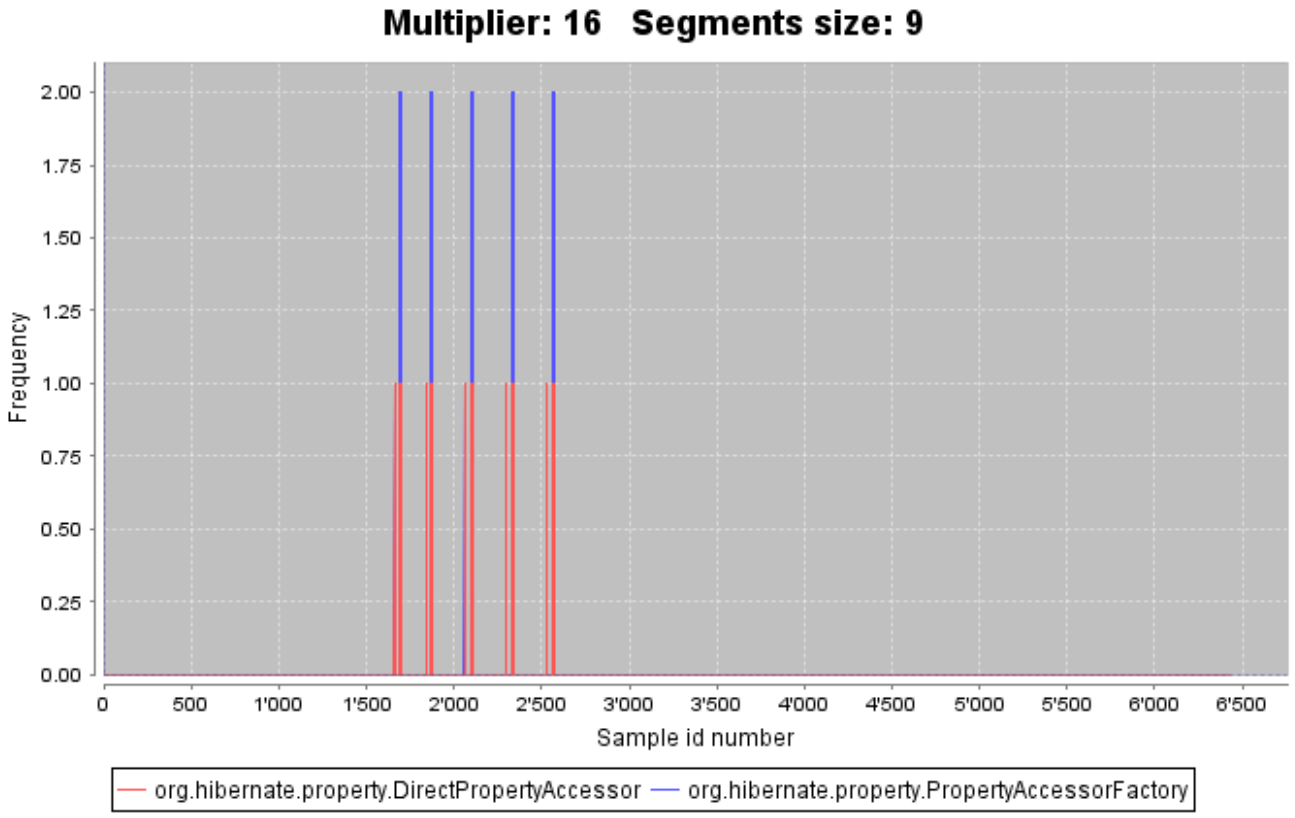
JAVADoc description: A factory for building/retrieving PropertyAccessor instances.

Les méthodes les plus appelées sont :

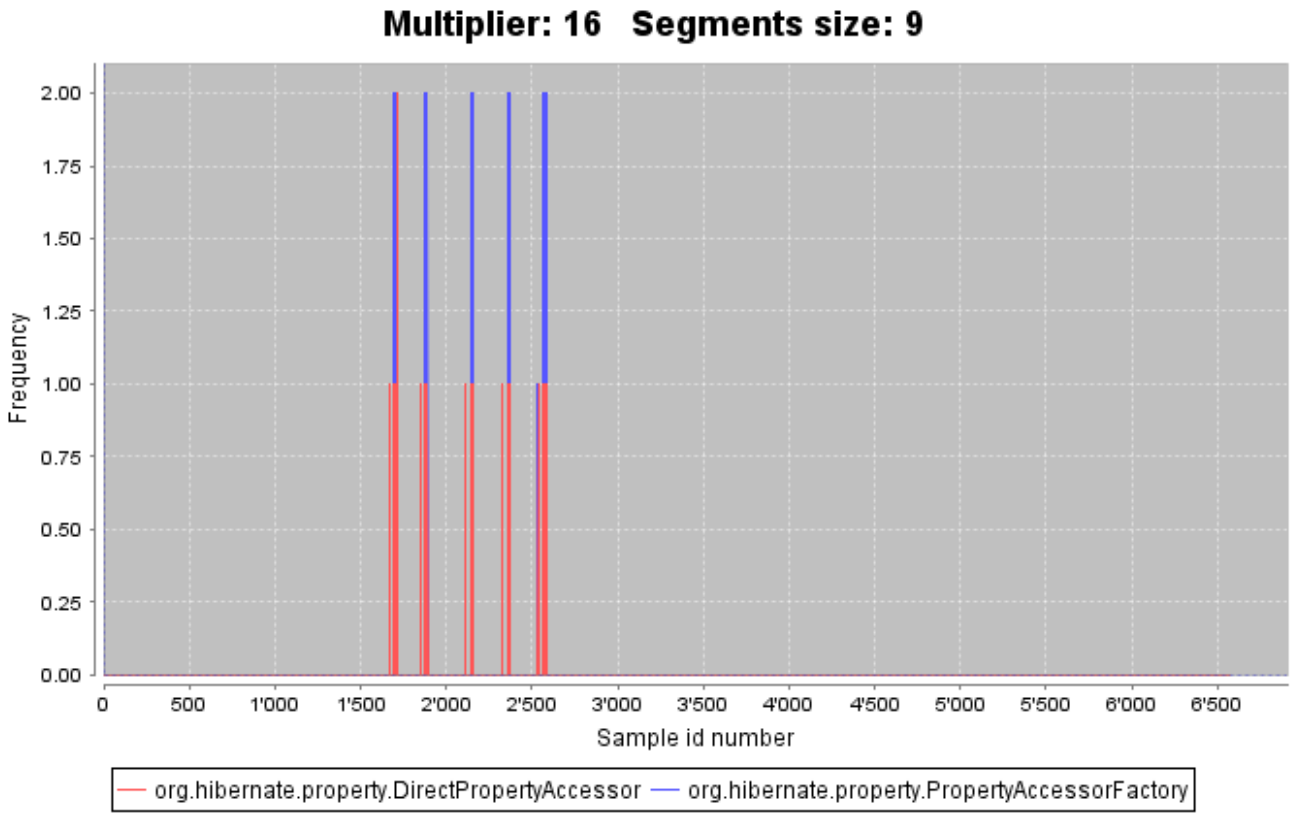
- DirectPropertyAccessor
  - getSetter(java.lang.Class,java.lang.String) : org.hibernate.property.Setter
  - getGetter(java.lang.Class,java.lang.String) : org.hibernate.property.Getter
- PropertyAccessorFactory
  - getPropertyAccessor(java.lang.Class,java.lang.String) : org.hibernate.property.PropertyAccessor
  - getPropertyAccessor(java.lang.String) : org.hibernate.property.PropertyAccessor

Nous nous rapprochons du milieu de la trace et visiblement Hibernate tente de récupérer les accesseurs des attributs des classes à persister, afin de pouvoir y accéder plus tard. De nouveau, il est très curieux de savoir pourquoi ce cluster n'apparaît pas dans d'autres stratégies ?

Tout comme avec le cluster précédent, nous avons diminué le Multiplier à 15 et ce cluster apparaît alors dans les traces des deux autres stratégies avec le même positionnement temporel.

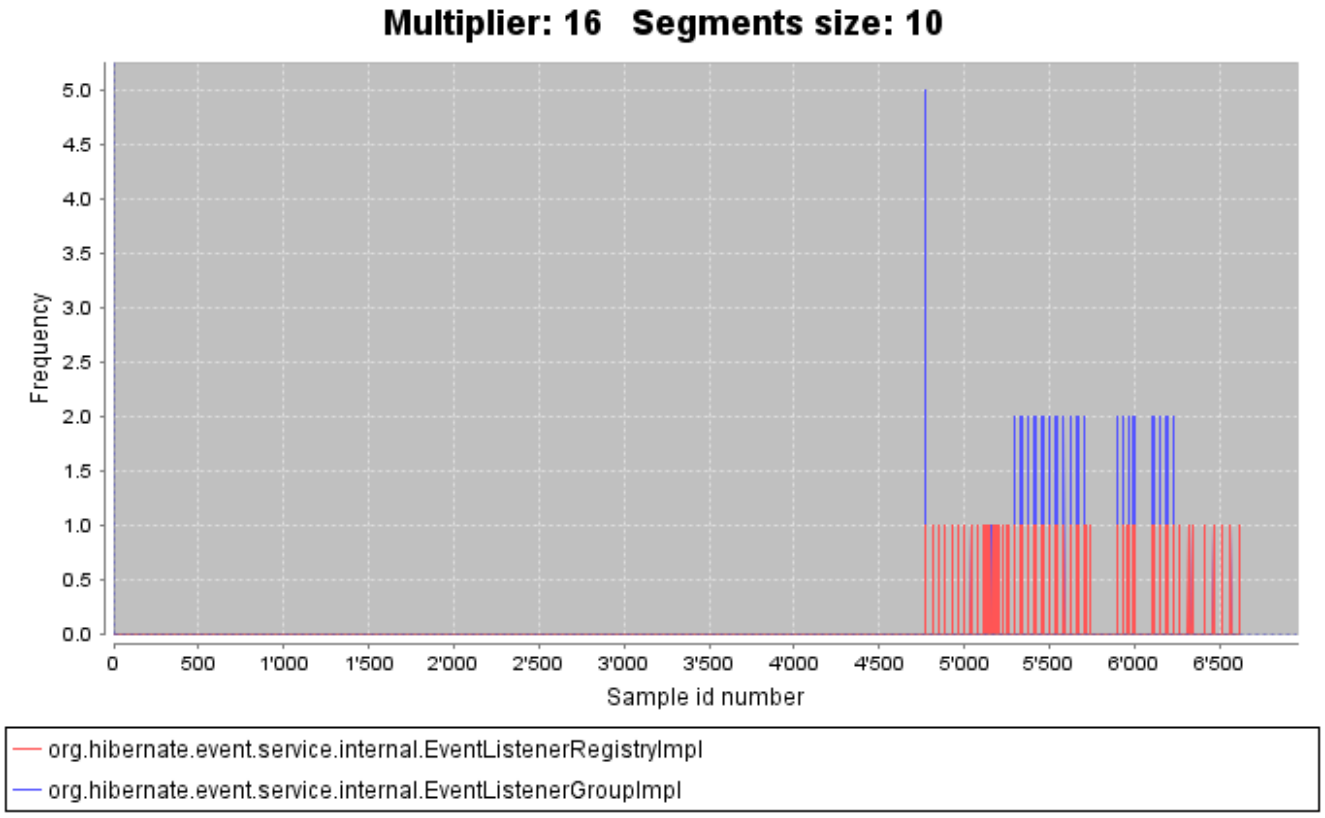


Figures 35. Stratégie SINGLE. Cluster org.hibernate.property.



Figures 36. Stratégie TABLE PER CLASS. Cluster org.hibernate.property.

org.hibernate.event.service.internal	EventListenerGroupImpl
org.hibernate.event.service.internal	EventListenerRegistryImpl



Figures 37. Cluster N°2 propre à la stratégie JOINED.

Corrélation : 80%

7.5.3.2 Responsabilités des classes et observations

org.hibernate.event.service.internal.EventListenerGroupImpl

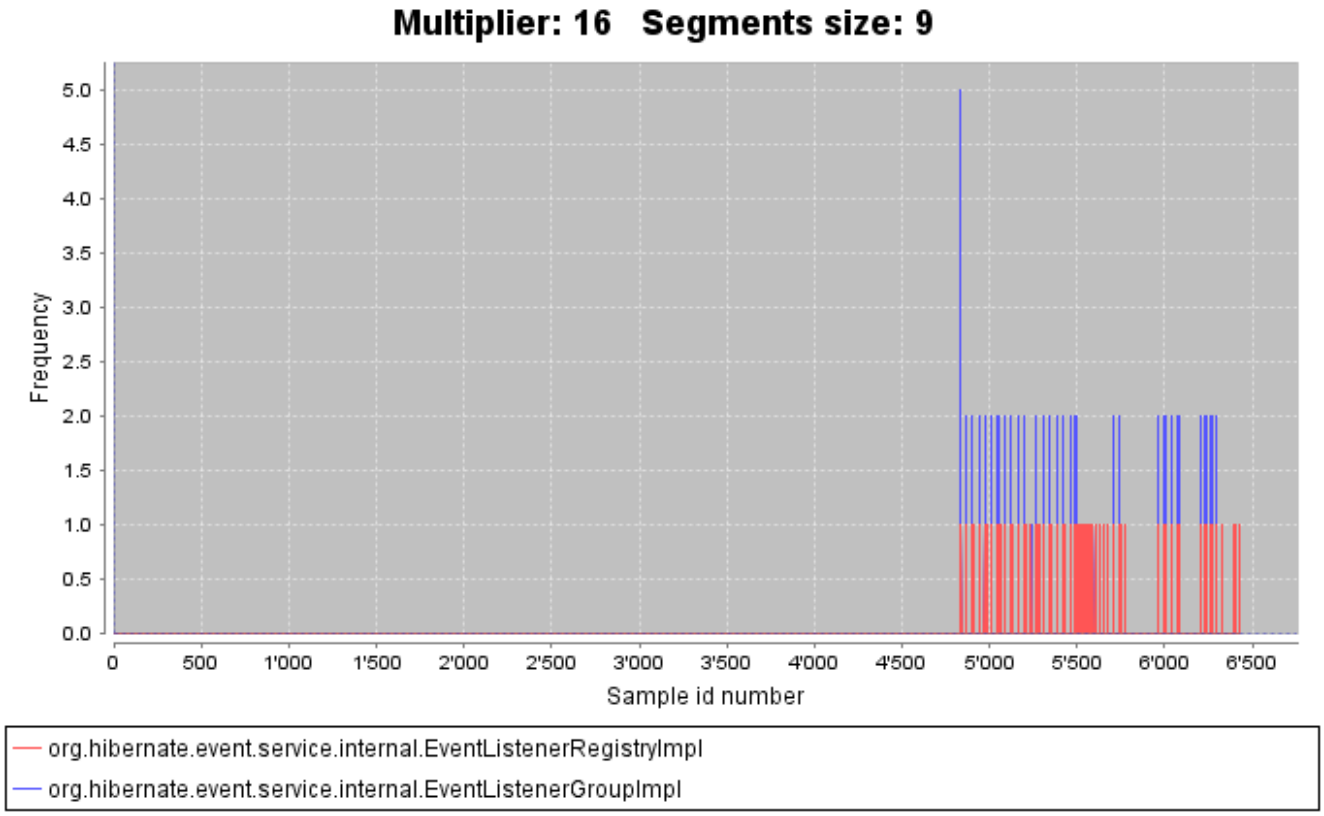
JAVADoc description: non mentionnée.

org.hibernate.event.service.internal.EventListenerRegistryImpl

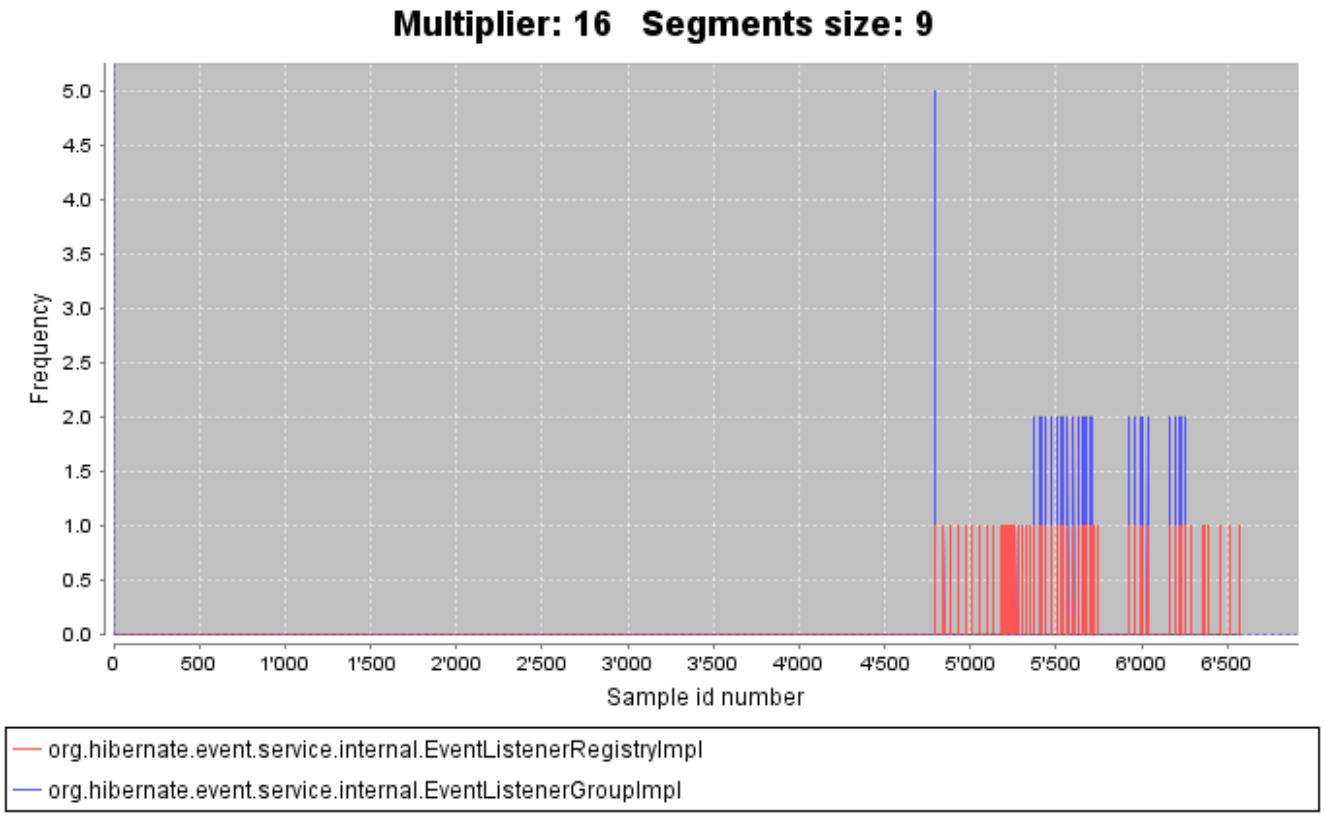
JAVADoc description: non mentionnée.

Nous n'avons malheureusement pas trouvé de description dans la JAVADoc Hibernate, mais en étudiant les interfaces implémentées et les méthodes appelées, nous arrivons à une conclusion que Hibernate tente de récupérer les listeners des évènements internes à son fonctionnement.

Nous avons également modifié le Multiplier pour voir si ce cluster est présent dans d'autres stratégies, il s'avère que lui aussi est bel et bien présent au même endroit dans la trace.



Figures 38. Stratégie SINGLE. Cluster org.hibernate.event.service.internal.



Figures 39. Stratégie TABLE PER CLASS. Cluster org.hibernate.event.service.internal.

7.6 Classes les plus sollicitées

Pour comprendre l'architecture fonctionnelle nous avons vu jusqu'à présent les classes propres à chaque stratégie, leurs collaborations les plus fortes et leurs occurrences temporelles dans les traces. Cette information est, certes, importante pour comprendre le fonctionnement, mais sommes-nous sûrs d'avoir assez d'éléments pour dessiner l'architecture fonctionnel de mapping Hibernate ? La réponse est non, les classes propres et les clusters ne nous disent rien sur qui détient véritablement l'intelligence nécessaire à la réalisation de nos use cases. Car ces classes ne constituent qu'une petite partie par rapport à la totalité des classes utilisées. Ce qui est utile lorsqu'on étudie une architecture fonctionnelle de n'importe quelle application c'est de savoir s'il y a des classes dont les méthodes sont plus souvent utilisées que les autres. Cette information peut aider à comprendre les besoins auxquels l'architecture fonctionnelle essaie de répondre le plus souvent. À condition bien sûr qu'il y ait une sémantique dans les signatures des méthodes et dans les noms des classes appelées.

Nos traces possèdent les informations sur les classes appelées dynamiquement et statiquement. Par une classe dynamique, nous appelons une classe dont l'instance exécute la méthode appelée dans la trace, alors qu'une classe statique est une classe où cette méthode a été réellement déclarée. Si A

hérite de B et que l'on appelle une méthode déclarée dans B depuis l'instance de A, la classe A serait une classe dynamique et B statique.

Nous allons comparer les vues statique et dynamique de nos traces, ce qui va nous indiquer si l'héritage est fortement utilisé ou non. Si les deux vues d'une trace sont très différentes l'une par rapport à l'autre, cela voudrait dire que l'héritage est fortement utilisé et vice-versa.

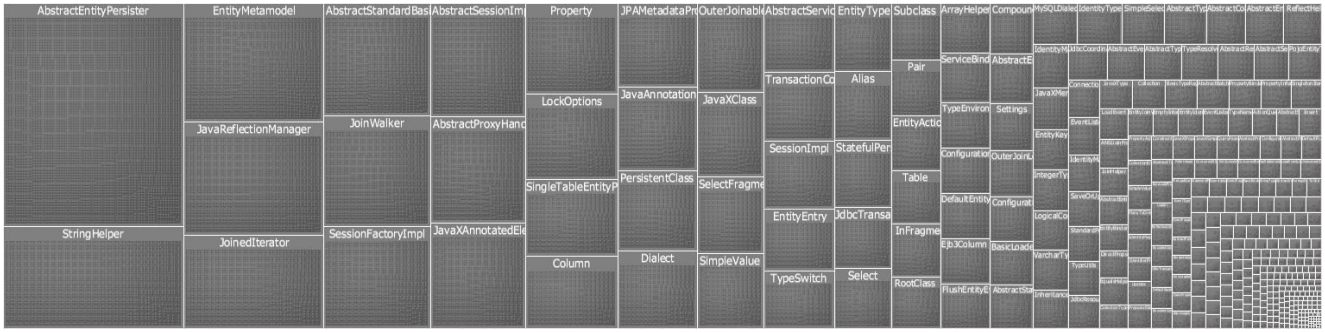
Pour information, les classes statiques et plus précisément les classes abstraites ne peuvent pas apparaître dans la vue dynamique, car dans cette vue il n'y a que des classes instanciées. Et comme nous le savons si bien, une classe abstraite ne peut pas être instanciée.

Pour donner du poids à notre raisonnement, nous allons donner un simple exemple de la trace SINGLE TABLE, vue de deux manières différentes, dynamique et statique. Grâce à DDRA, et plus particulièrement à sa base ORACLE qui contient les informations sur les traces, nous allons faire deux requêtes SQL qui nous retourneront les informations suivantes :

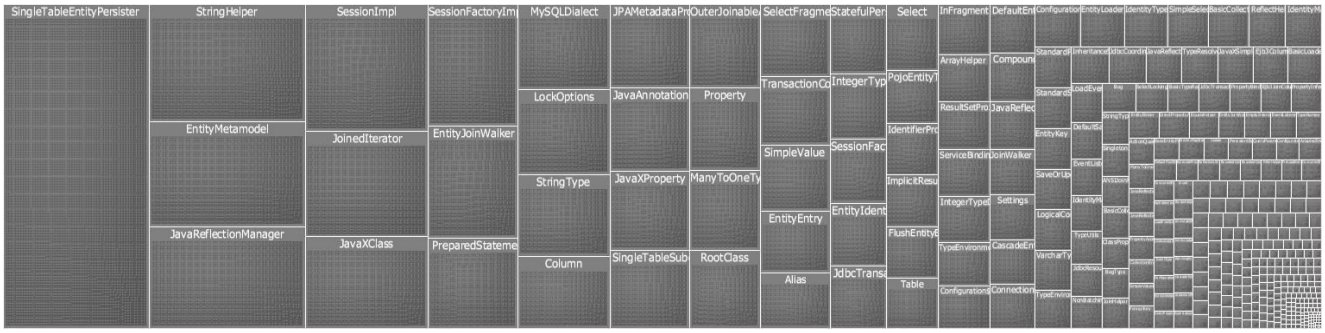
- Toutes les classes appelées dynamiquement
- Toutes les classes appelées statiquement

Lecture des images qui suivent doit se faire comme suit. Chaque carré représente une classe qui figure dans la trace. La taille de chaque carré représente le nombre de fois que cette classe a été enregistrée (appelée) dans la trace.

Voici ci-dessous les résultats :

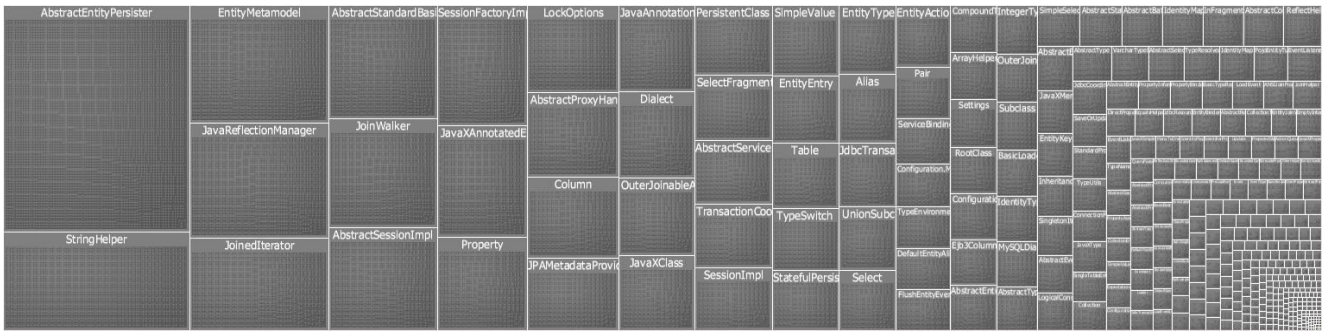


Figures 40. Classes où les méthodes appelées sont déclarées. Trace SINGLE TABLE.

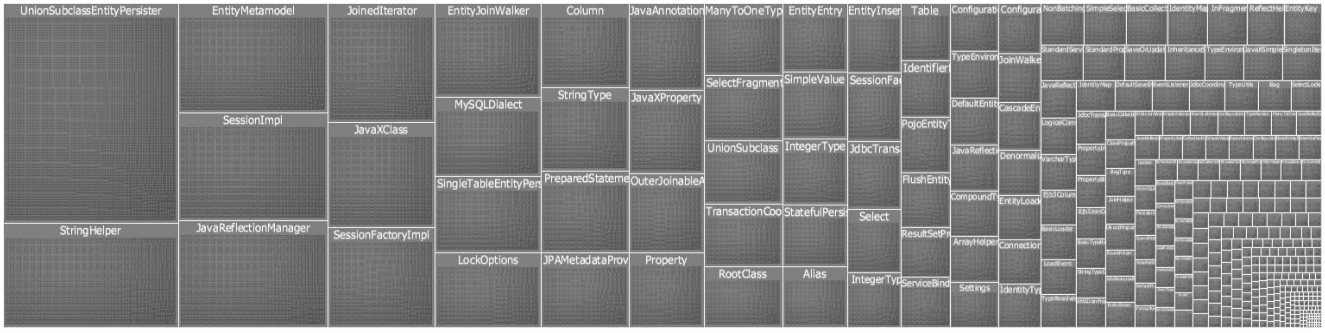


Figures 41. Classes appelées réellement. Trace SINGLE TABLE.

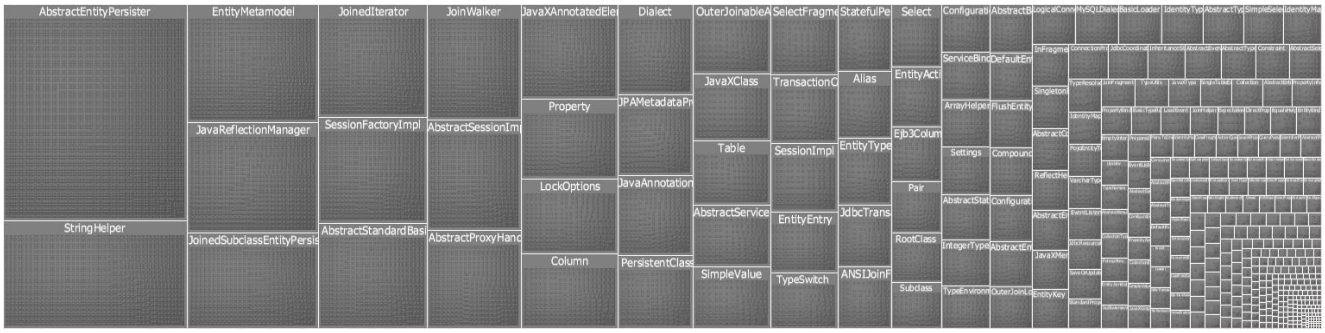
Comme vous pouvez le constater les vues statique et dynamique sont très différentes sur la même trace. Prenons par exemple les classes les plus appelées dans cette trace : *org.hiberante.persister.entity.SingleTableEntityPersister* et *org.hiberante.persister.entity .AbstractEntityPersister.* Il se trouve que la table *SingleTableEntityPersister* hérite de la classe *AbstractEntityPersister.* Comment peut-on être sûr que la classe *SingleTableEntityPersister,* qui est la plus appelée statiquement n'a pas hérité son comportement de *AbstractEntityPersister* ? En effet, cela est difficile à dire sans comparer les deux vues et c'est exactement ce que nous allons faire.



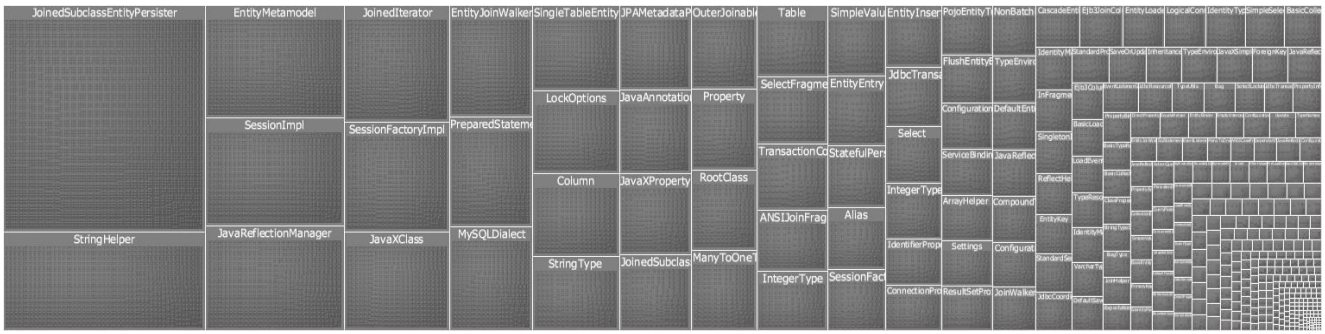
Figures 42. Classes appelées statiquement dans la trace TABLE PER CLASS.



Figures 43. Classes appelées dynamiquement dans la trace TABLE PER CLASS.



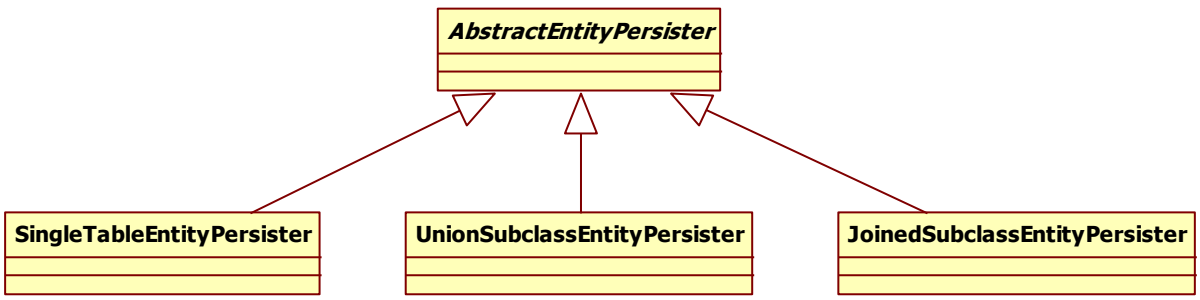
Figures 44. Classes appelées statiquement dans la trace JOINED.



Figures 45. Classes appelées dynamiquement dans la trace JOINED.

7.6.1 Observations

Tout comme les packages observés au début de cette analyse, nous constatons que les classes les plus appelées aussi bien dynamiquement que statiquement sont quasiment les mêmes pour les trois stratégies, à quelques exceptions près. Une très grande différence réside dans les classes de type *XXXEntityPersister* et leur classe mère *AbstractEntityPersister.*



Figures 46. Classes XXXEntityPersister.

Ce sont des classes les plus sollicitées dynamiquement dans les trois traces et chacune d'elle correspond à une stratégie de mapping, alors que la classe AbstractEntityPersister est celle dont les méthodes sont les plus sollicitées à l'exécution. C'est elle qui est la plus sollicitée dynamiquement, ce

qui est très logique, car les trois classes XXXEntityPersister sont ses classes-filles. Les occurrences temporelles de ces classes peuvent être consultées en Annexes de 8 à 13.

Au niveau temporel, la classe *AbstractEntityPersister* occupe donc la place centrale dans la trace, c'est elle qui est la plus sollicitée au milieu de la trace juste après les classes du cluster commun du début de la trace.

Voici ci-dessous la JAVADoc de ces classes :

org.hibernate.persister.entity.AbstractEntityPersister

```
/**
 * Basic functionality for persisting an entity via JDBC
 * through either generated or custom SQL
 *
 * @author Gavin King

 */
```

org.hibernate.persister.entity.SingleTableEntityPersister

```
/**
 * The default implementation of the <tt>EntityPersister</tt> interface.
 * Implements the "table-per-class-hierarchy" or "roll-up" mapping strategy
 * for an entity class and its inheritence hierarchy.  This is implemented
 * as a single table holding all classes in the hierarchy with a discrimator
 * column used to determine which concrete class is referenced.
 *
 * @author Gavin King

 */
```

org.hibernate.persister.entity.UnionSubclassEntityPersister

```
/**
 * Implementation of the "table-per-concrete-class" or "roll-down" mapping
 * strategy for an entity and its inheritence hierarchy.
 *
 * @author Gavin King

 */
```

org.hibernate.persister.entity.JoinedSubclassEntityPersister

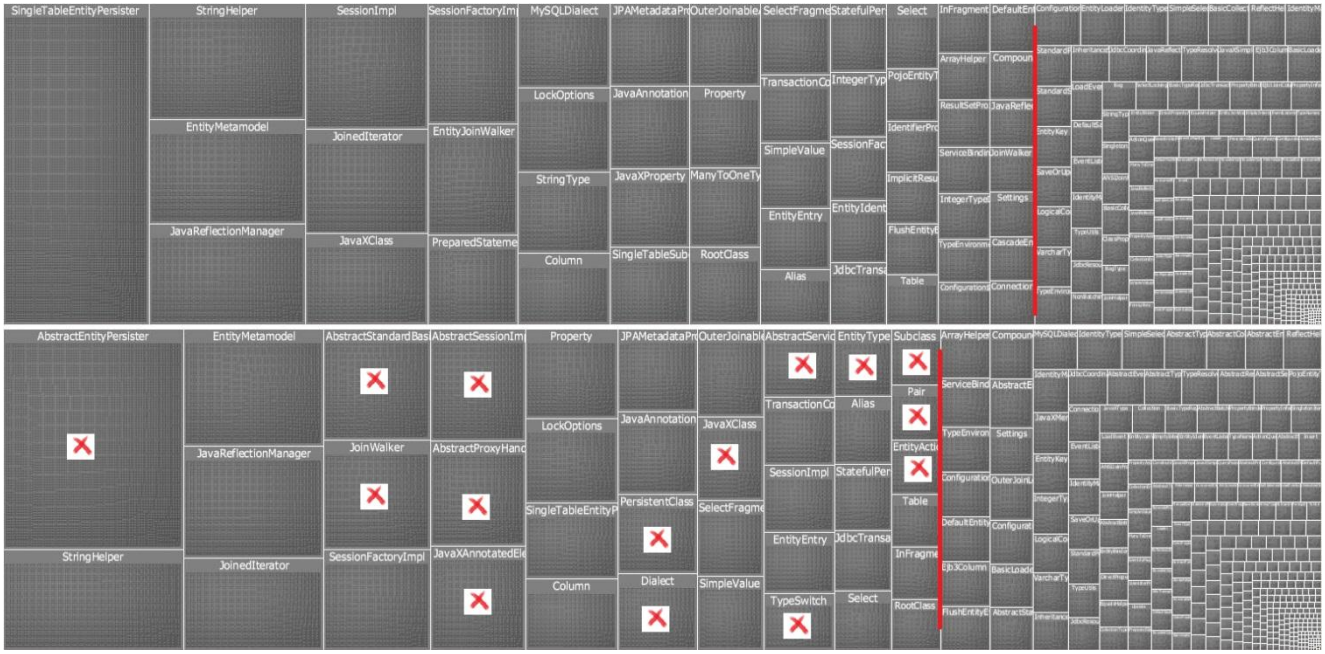
```
/**
 * An <tt>EntityPersister</tt> implementing the normalized "table-per-subclass"
 * mapping strategy
 *
 * @author Gavin King

 */
```

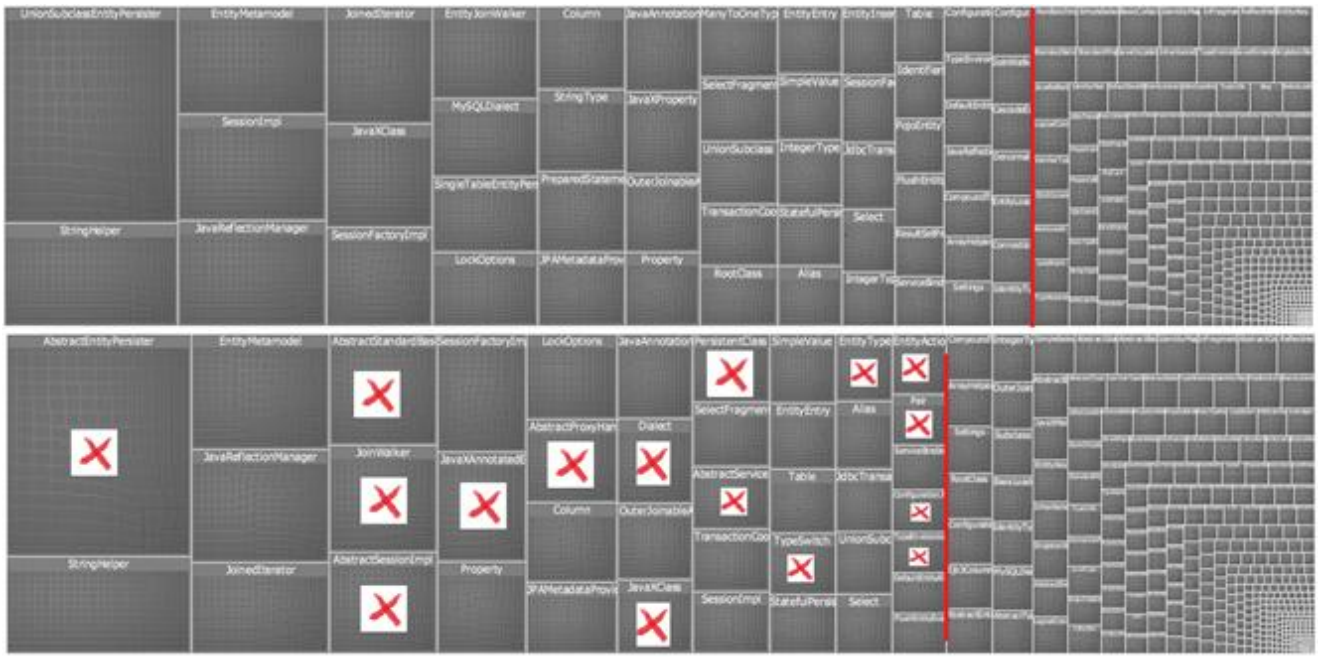
Rappelons-nous notre deuxième objectif, qui est de découvrir si l'héritage est fortement utilisé ou non. Pour ce faire, nous allons marquer les classes qui n'apparaissent pas dans la vue dynamique

avec des croix rouges, le plus souvent ce sont des classes abstraites, mais pas toujours. Il se peut qu'une classe concrète fasse un appel à une de ses méthodes héritée d'une autre classe qui est elle est aussi concrète. Nous pouvons donc avoir une situation où ces classes peuvent apparaître toutes les deux dans la vue dynamique. Et s'il y a beaucoup de croix, cela confirmera notre hypothèse comme quoi l'héritage est abondamment utilisé dans le mapping minimal Hibernate.

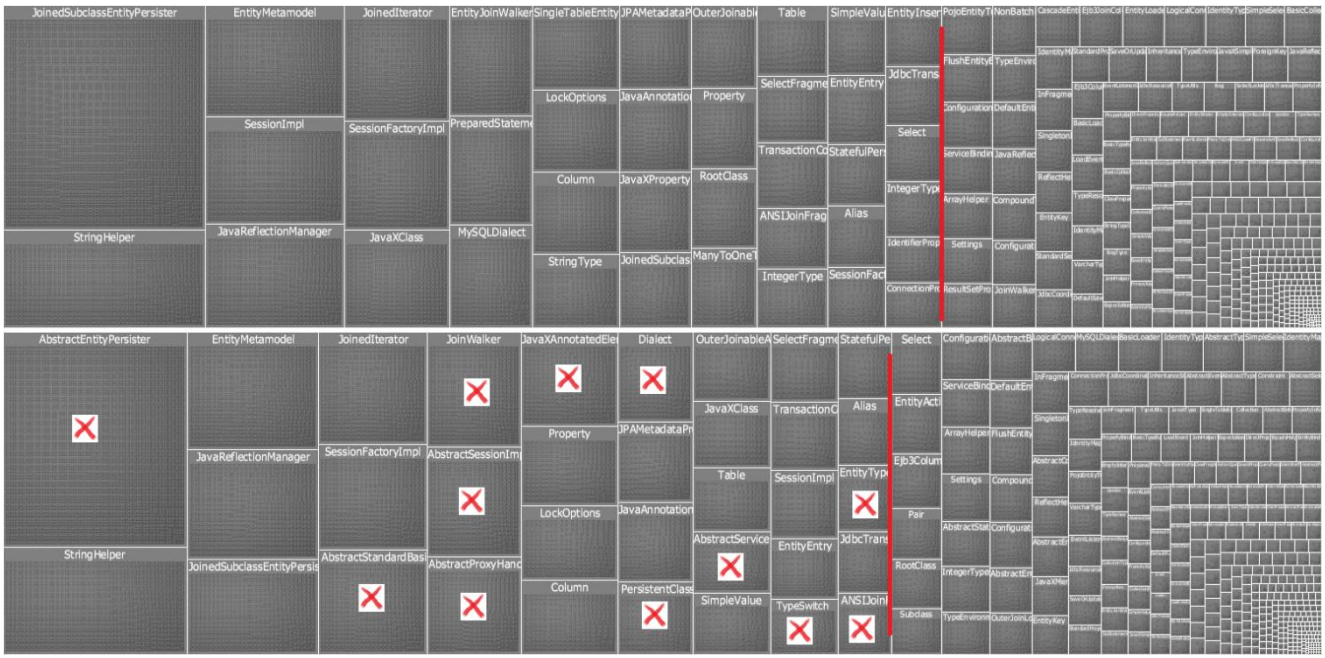
Pour des résultats ci-dessous, nous n'avons pas consulté toutes les classes des traces. Nous nous sommes concentrés uniquement sur des classes dont l'occurrence temporelle est vraiment importante. La ligne rouge sépare les classes consultées des autres classes, afin de voir si déjà à cette étape-là il y a beaucoup de différence ou non.



Figures 47. Différence vue statique et dynamique. Stratégie SINGLE TABLE.



Figures 48. Différence vue statique et dynamique. Stratégie TABLE PER CLASS.



Figures 49. Différence vue statique et dynamique. Stratégie JOINED.

Comme vous pouvez le constater la différence est plutôt considérable entre les deux vues, un peu moins de la moitié des classes statiques, les plus sollicitées, ne sont pas présentes dans la vue dynamique de la même trace. Cela veut dire qu'à l'exécution, certaines méthodes de ces classes ont été appelées très abondamment par une de leurs classe fille, qui elle figure très certainement quelque part dans la vue dynamique. Cela confirme notre hypothèse de départ et nous pouvons

affirmer que l'architecture requise pour le mapping minimal Hibernate exploite l'héritage de manière considérable.

7.6.2 Noyau fonctionnel de mapping minimal Hibernate

Un de nos objectifs de départ était de déterminer quels sont les packages et les classes les plus importantes dans l'implémentation de mapping Hibernate. Nous sommes désormais en possession de certains éléments qui peuvent nous permettre de « dessiner » cette architecture interne. Ces éléments sont les suivants :

- Les classes propres à chaque stratégie de mapping
- Les clusters communs et propres à chaque stratégie de mapping
- Les occurrences temporelles des classes qui constituent les clusters étudiés
- Les classes les plus sollicitées statiquement et dynamiquement

L'image ci-dessous regroupe tous ces éléments dans des packages marqué en vert.

Figures 50. Noyau fonctionnel Hibernate.

Consulter l'annexe N°14 pour l'image avec une meilleure résolution.

8. Difficultés rencontrées

8.1.1 Manipulation du code source

La toute première difficulté que j'ai rencontrée était l'obtention de code source d'Hibernate et sa manipulation. Depuis le site officiel d'Hibernate, il est disponible via un téléchargement simple, mais il y a une nuance.

Le code source n'est pas exploitable tel quel, on peut, certes, voir le contenue de toutes les classes, mais dans le cadre de mon travail je devais instrumenter ce code en utilisant Eclipse, car le plugin qui permet l'instrumentation n'est disponible qu'avec cet IDE. Le problème que j'ai eu alors c'est comment construire un ou plusieurs projets Eclipse afin de pouvoir les instrumenter par la suite ? Dans le jargon on dit qu'il faut « builder » des projets, c'est-à-dire de créer des fichiers : **.classpath**, **.project**, dossiers **/bin**, **/lib** pour que ce code source soit exploitable par Eclipse.

C'est à ce moment-là que Gradle entre en jeu. Hibernate utilise cette application pour builder des projets Eclipse. L'apprentissage peut prendre beaucoup de temps, car la documentation de cet outil est plus que complète et pas facilement compréhensible à tout le monde. Il m'a fallu 1-2 semaine pour bien comprendre comment utiliser Gradle pour builder mes projets sous Eclipse.

Mais revenons au code source, il est bel et bien disponible sur le site officiel d'Hibernate, mais n'est exploitable que si Gradle est préinstallé sur la machine locale. Par contre si on le télécharge depuis GitHub, il est exploitable tout de suite. La nuance réside dans le fait que Gradle laisse le choix à l'utilisateur d'utiliser une ou l'autre manière pour builder des projets, et cela dépend du fait s'il le code source est fournie ou pas avec un certain fichier *gradlew,* il l'est si on utilise GitHub. Ce fichier permet à l'utilisateur de construire des projets sans que Gradle soit préinstallé sur la machine locale. Cette manière est même très conseillée, car si la version installée de Gradle est différente de celle qui a été utilisée pour construire le projet à partir du code source, les commandes Gralde (l'outil est utilisé via la ligne de commande) ne vont pas s'exécuter correctement. Alors que si on utilise Gradle Wrapper (gradlew), il n'y a plus de problème de version. Ce wrapper a été conçu dans l'idée d'intégration continue des projets JAVA, s'il est inclus dans le projet, cela veut dire qu'en l'utilisant, cela aura comme effet de télécharger automatiquement sur votre machine locale la bonne version de Gradle à utiliser pour construire un projet JAVA.

«When you start a Gradle build via the wrapper, Gradle will be automatically downloaded and used to run the build. »[9]

8.1.2 Apprentissage de Gradle

Il est douloureux et plein de pièges, mais avec un peu de patience, un peu de lecture et de tests tout est possible. Gradle fournit à l'utilisateur une vingtaine de commandes (tasks) utilisables via la ligne de commande et pour les manipuler correctement, il faut avoir une bonne compréhension du cycle de vie d'un projet Java. Pour donner un exemple, je cherchais au départ un moyen pour générer des fichiers .jar après avoir instrumenté le code source. Et j'ai tout de suite repéré la commande ***gradle jar***, qui a pour effet de générer des archives jar des classes compilées. Non seulement il m'a fallu du temps pour trouver l'endroit où sont sauvegardés ces archives, mais surtout, j'ai mis du temps à comprendre que mon code instrumenté doit être d'abord compilé avec une autre commande et seulement après il faut utiliser la commande ***gradle jar***. Si on ne compile pas d'abord les classes, la commande ***gradle jar*** génère les mêmes archives qu'au départ. Pour finir, j'ai trouvé une autre commande ***gradle clean install***, qui fait tout en même temps. Là encore, c'est bien beau d'avoir un *SUCCESSFULL BUILD* à la fin de la ligne de commande, mais si on ne nous dit pas l'endroit où se trouvent ces archives, on peut mettre pas mal de temps à chercher.

8.1.3 Erreurs de la compilation

J'ai rencontré également beaucoup de problèmes lors de la compilation des classes instrumentés. Ces erreurs sont dues au fait que la version de base d'instrumentor n'était pas conçu à travailler avec des classes paramétrées et Hibernate les utilise abondement.

Il ne m'était pas possible par exemple de connaître le type renvoyé par une méthode, car l'instrumentor ajoutait *« .class »* à la fin de *« <T> »* et cela causait une erreur de compilation.

J'ai rencontré plus de mille erreurs à la compilation, ce qui m'empêchait de poursuivre mon travail. Corriger ces erreurs à la main n'était pas évidement la meilleure des solutions. J'ai donc rapidement communiqué ce problème aux auteurs du plugin, Philippe Dugerdil et David SENNHAUSER. Plusieurs version ont été éditée depuis et le problème est désormais résolu.

---

[9] Site officiel Gradle. Documentation. Consulté le 21.10.2012 http://gradle.org/docs/current/userguide/gradle_wrapper.html

8.1.4 Librairies non instrumentées

Tout au début de ce travail, nous avons instrumentés uniquement *hiberante-core* et *hibernate-entity-manager*, car dans le code source téléchargé depuis GitHub, il n'y a que ces deux projets qui figurent parmi les projets obligatoires au fonctionnement de Hibernate (cf. Figure 23 Code source Hibernate). Nous nous sommes dit que les jars « required » (cf. Figure 24 Librairies obligatoires du framework Hibernate), tels que *hibernate-jpa* et *hibernate-commons-annotations* seront générés et instrumentés en cours de « build » des projets principaux. Mais ce n'était pas le cas après l'analyse des premières traces. Il a fallu donc aussi instrumenter ces librairies, car nous avons utilisé que des annotations pour mapper les classes de nos use cases. Nous avons donc rajouté deux dépendances supplémentaires à chaque projet Eclipse qui correspondaient à un de nos use case. Ce sont en fait deux projets qui contiennent que le code source des deux librairies mentionnées. Nous les avons instrumentés et ajoutés en tant que dépendance, après quoi nous avons régénéré les traces.

9. Conclusion

Rappel

Rappelons-nous l'objectif de ce travail de diplôme qui est l'étude de l'architecture fonctionnelle de mapping minimal Hibernate. Nous avons voulu déterminer les composants clés dans l'implémentation de cette fonctionnalité. Pour ce faire, nous avons créé un use case générale et simpliste qui implémente le CRUD complet et qui possède quelques liens d'héritage dans son modèle d'objets de domaine. De là, nous avons étendu ce use case en créant trois spécialisations, qui sont les stratégies de mapping objet/relationnel.

Afin de connaître les composants clés de l'architecture Hibernate, nous avons d'abord instrumenté le code source, ce qui nous a permis de passer à l'étape suivante, qui est la génération de traces d'exécution. Finalement, ces traces ont été analysées avec une application d'analyse de trace d'exécution DDRA.

Nous nous sommes intéressés d'abord aux packages impliqués, puis aux classes qui participent à la réalisation de la fonctionnalité étudiée. Nous avons étudié les occurrences temporelles des classes les plus sollicitées afin de comprendre qui est appelé quand et à quelle fréquence. Pour finir, nous avons exposé les vues statique et dynamique de nos use cases, ainsi que la comparaison entre ces dernière.

Complexité de framework Hibernate

Nous avons vu dès le départ de notre expérimentation que le nombre de classes impliquées dans la réalisation de nos use cases est assez important. Environs 400 classes ont été appelées pour mapper cinq classes et effectuer quelques opérations CRUD. N'est-il pas surprenant ce nombre ? Est-ce que l'on peut tirer une quelconque information de cette constatation ?

Il ne nous est malheureusement pas possible d'affirmer quoi que ce soit à partir de cette information, car nous n'avons pas fait de comparaison avec d'autres framework de persistance. Néanmoins à nos yeux, ce nombre reste important pour le peu que nous avons exigé d'Hibernate. Si l'on suit notre logique, on peut supposer que la complexité du framework Hibernate est un élément à prendre en compte lorsque l'on décide d'utiliser cet API open source au sein d'une application JAVA. Il est d'ailleurs mentionné dans Java Persistance et Hibernate de PATRICIO Anthony *que « la gratuité d'une implémentation de Java Persistence telle qu'Hibernate ne doit pas faire illusion. <…> pour un développeur moyennement expérimenté, la courbe d'apprentissage <...> est généralement estimée de quatre à six mois pour en maitriser les 80% de fonctionnalités les plus utilisées. »*

Un autre point important qui en découle est la compréhension et le débogage du code source qui sont des éléments importants lorsque l'on se retrouve face à un problème lié à la manière dont Hibernate est développé. Avec une telle complexité, le débogage peut s'avérer très couteux en terme de temps et au final de l'argent. Ce coût serait encore plus important si la compréhension n'est pas acquise.

La complexité du framework Hibernate n'est finalement pas vraiment une surprise si on prend la peine de bien étudier son architecture, qui se veut être en couche (cf. chapitre sur l'architecture Hibernate). Ce type d'architecture a pour objectif de regrouper les composants, avec des responsabilités proches, du reste de l'application en définissant des interfaces comme moyen de communication intra couches. Cette isolation permet une meilleure spécialisation de ces composants, qui une fois isolés, peuvent se concentrer entièrement sur leurs propres responsabilités, sans se soucier de ce qui se fait ailleurs et ainsi donc d'améliorer la qualité du travail fourni. Cette qualité a malheureusement a un prix qui est justement cette complexité, que nous avons pu observer grâce au traçage dynamique de notre simple use case.

Forte distribution et délégation

Au cours de notre analyse, nous nous sommes rendu compte que Hibernate fait appel à une grande majorité des classes qui ne sont que très rarement sollicitées. Cette information nous donne une indication précieuse sur la manière dont l'architecture Hibernate est construite. Plutôt que faire un appel à une ou deux classes bien précises, nous passons par une multitude de classes qui s'enchaînent l'une après l'autre.

Cette complexité dans les appels ne laisse aucun doute sur le fait que la délégation et la distribution de tâches entre les classes est un élément clé de l'architecture Hibernate. Cela nous permet aussi, mais dans la moindre mesure car nous n'avons utilisé qu'un cinquième de toutes les classes Hibernate, de soupçonner une forte spécialisation de classes de ce framework. Car ces deux concepts sont des caractéristiques d'un environnement fortement spécialisé.

Architecture des stratégies de mapping Hibernate

La spécialisation nous permet de rebondir sur un autre point important qui a été observé au cours de notre analyse. Il s'agit de la faible différence entre les trois manières de mapper les classes avec Hibernate, qui réside principalement dans les trois classes de package o*rg.hibernate.persister.entity* .dont chacune est propre à une stratégie : *SingelTableEntityPersister, UnionSubclassEntityPersister* et *JoinedSubclassEntityPersister.* Cette faible différence peut être vue sous un autre angle, qui est la forte spécialisation (une classe par stratégie) de chaque stratégie de mapping Hibernate. Cela veut dire que dans l'ensemble, le mapping Hibernate peut être vu comme un processus assez générique

orchestré par des stratégies de mapping dont l'intelligence réside principalement dans les classes citées. Cette affirmation est certes très osée et manque d'arguments, mais cela s'avère être le cas dans le mapping minimal Hibernate implémenté dans le cadre de ce travail de Bachelor.

Les occurrences temporelles de ces classes, tout comme celle de leur classe mère abstraite *org.hibernate.persister.entity.AbstractEntityPersister*, sont les plus importantes parmi les classes observées et sont quasiment identiques. Hibernate commence par faire appel aux méthodes de ces classes peu avant le milieu de toutes les traces (cf. les annexes 8-13) avec en moyenne 3 appels par segment, constitué de 9 à 10 appels de méthodes. Seules classes qui peuvent rivaliser avec elles sont : *org.hibernate.annotations.common.reflection.java.JavaAnnotationReader* et *org.hibernate.cfg.annotations.reflection.JPAMetadataProvider*, appelées au tout début de l'exécution. Grâce à ces informations sur les occurrences temporelles nous savons ce à quoi Hibernate emploie le plus de ressource afin de réaliser le mapping de notre use case.

Dans le cadre de ce travail de Bachelor nous avons exploré l'architecture dynamique de la fonctionnalité sans doute la plus utilisée de ce framework. Toutes fois, seulement une faible partie de cette fonctionnalité a été étudiée. Le mapping Hibernate est en réalité beaucoup plus puissant et permet de faire beaucoup plus qu'un simple CRUD et la gestion de quelques liens d'héritage. Nous étions très ambitieux au début de ce travail, mais peu à peu nous étions obligés de restreindre notre champ de recherche, notamment à cause des difficultés techniques que nous avons rencontré pour construire et instrumenter le code source Hibernate. La complexité Hibernate et la moyenne expérience du monde JAVA de l'auteur ont aussi joué leur rôle, tout comme son emploi du temps assez chargé vers la fin de ce projet.

À présent, de nouvelles portes s'ouvrent et l'expérience vécue durant l'écriture de ce travail de Bachelor permettra à l'auteur de progresser, de développer sa curiosité et son goût pour de nouveaux challenges sans lequel il n'aurait jamais choisi ce sujet.

10. Acronymes

API – Application Programming Interface

CRUD – Create, Read, Update, Delete

JDBC – Java Data Base Connectivity

ORM – Object/Relational Mapping

SQL – Sequence Query Language

Varchar – Varying character

XML – Extensible Markup Language

CVS – coma separated values

JTA – Java Transaction API

11. Bibliographie

[1] HIBERNATE. Hibernate Developer Guide. *Guide pour les développeurs Hibernate* [en ligne]. http://docs.jboss.org/hibernate/orm/4.1/devguide/en-US/html_single/ (consulté le 09.11.2012).

[2] HIBERNATE. Javadoc. *Site de Javadoc de la version Hibernate 4.1* [en ligne]. http://docs.jboss.org/hibernate/orm/4.1/javadocs/ (consulté le 09.11.2012).

[3] HIBERNATE. Hibernate Getting Started Guide. *Guide de démarrage Hibernate* [en ligne]. http://docs.jboss.org/hibernate/orm/4.1/quickstart/en-US/html_single/ (consulté le 09.11.2012).

[4] HIBERNATE. Wiki hibernate. *Site wiki de la communauté Hibernate* [en ligne]. https://community.jboss.org/en/hibernate (consulté le 09.11.2012).

[5] GRADLE. Gradle User Guide. *Guide d'utilisation Gradle* [en ligne]. http://www.gradle.org/docs/current/userguide/userguide_single.html (consulté le 09.11.2012).

[6] BAUER, Christian. KING, Gavin. *Hibernate*. Paris : CampusPress, 2005. 421 p.

[7] PATRICIO, Anthony. *Java Persistence et Hibernate.* Paris : Eyrolles, 2008. 365 p.

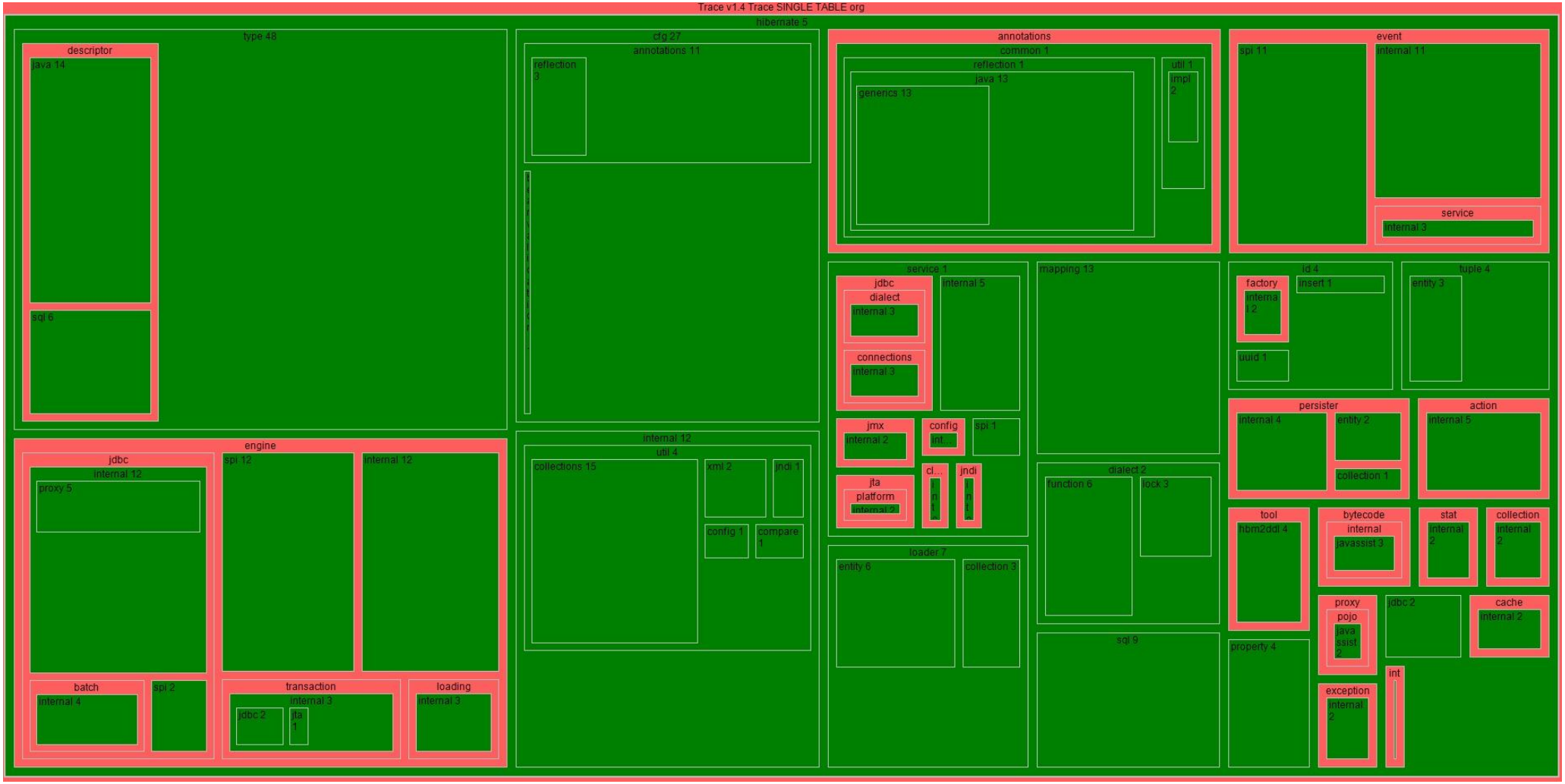
[8] KOUSHKS. *Javabrains Java Tutorials* [en ligne]. http://www.youtube.com/user/koushks (consulté le 09.11.2012).

Annexe 1
Stratégie SINGLE TABLE

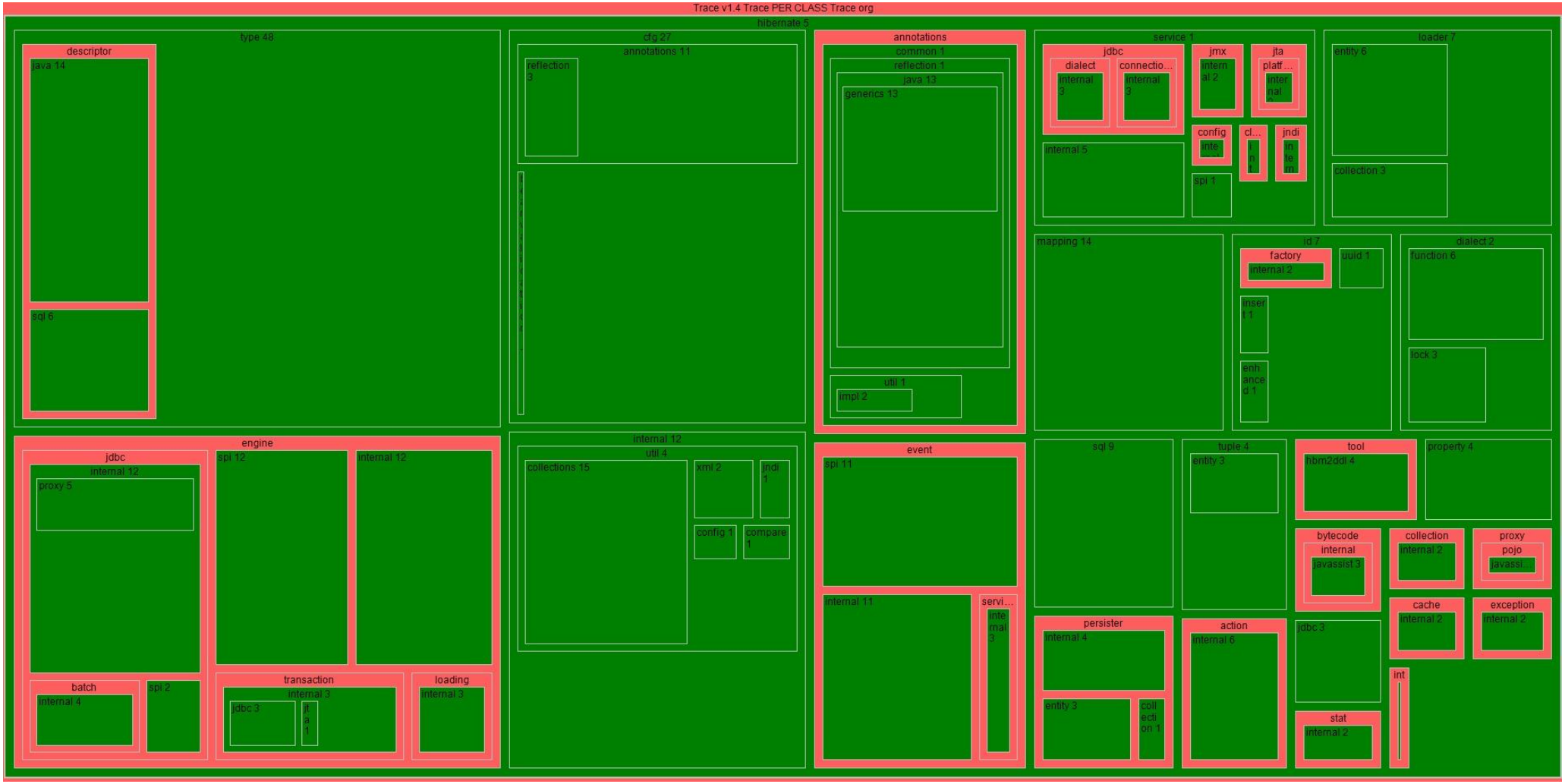
Annexe 2
Stratégie TABLE PER CLASS

Annexe 3
Stratégie JOINED

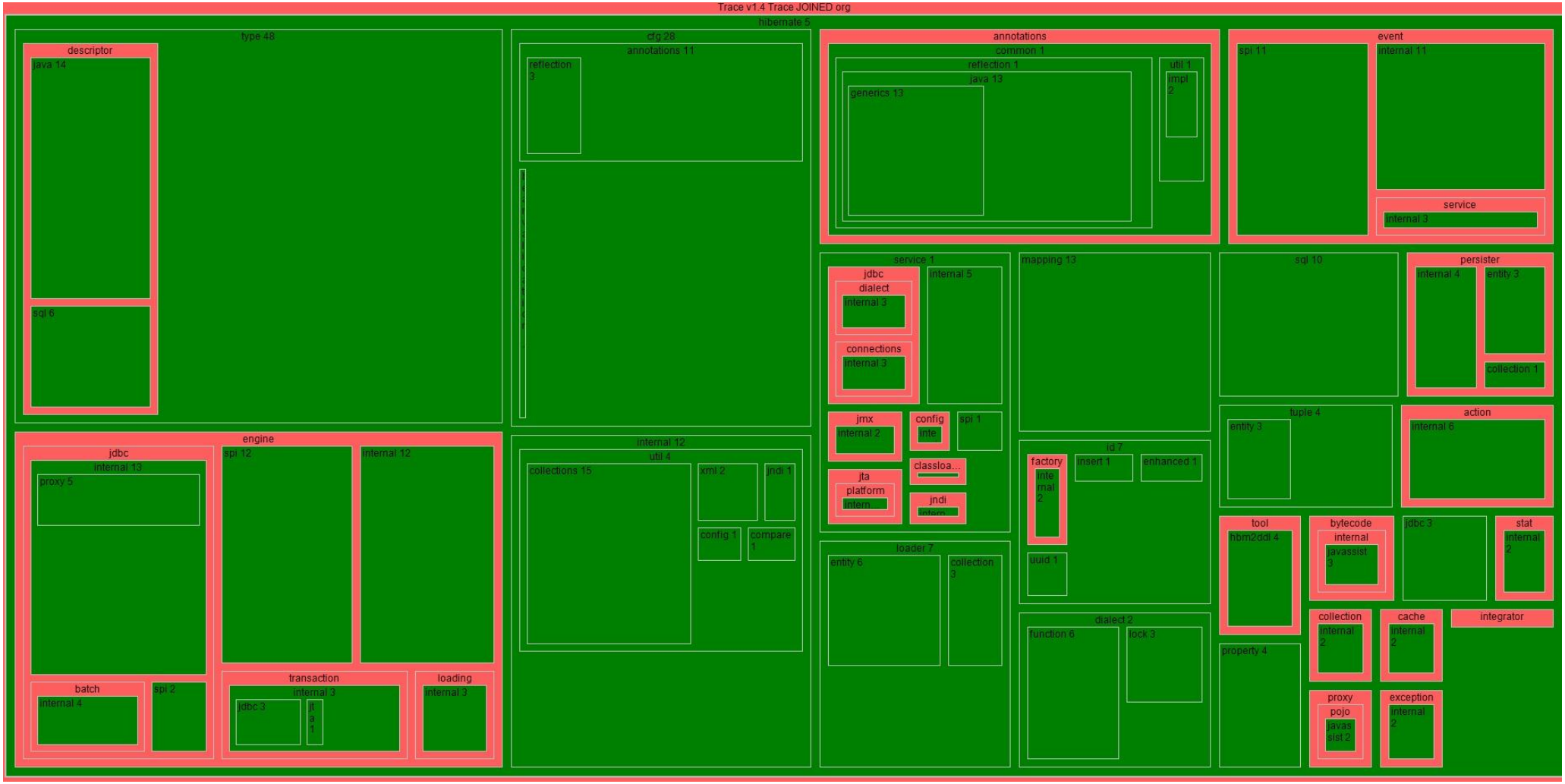
Annexe 4
Stratégie SINGLE TABLE, taille des packages proportionnelle au nombre d'appels



Annexe 5
Stratégie TABLE PER CLASS, taille des packages proportionnelle au nombre d'appels



Annexe 6
Stratégie JOINED, taille des packages proportionnelle au nombre d'appels



Annexe 7
Recherche de valeurs uniques dans un tableau Excel

Tout d'abord nous avons inséré les trois traces dans une feuille Excel, chaque trace occupe deux colonnes, une contient le nom de la classe et l'autre le nom du package de cette classe.

Trace 1		Trace 2		Trace 3	
Classe	Package	Classe	Package	Classe	Package
XXX	XXX	XXX	XXX	XXX	XXX
XXX	XXX	XXX	XXX	XXX	XXX
XXX	XXX	XXX	XXX	XXX	XXX
…	…	…	…	…	…

Pour trouver les classes qui ne figurent pas dans d'autres colonnes, et donc dans d'autres traces, nous avons utilisé la formule *RECHERCHEV*. Voici sa description Excel :

« Cherche une valeur dans la première colonne à gauche d'un tableau, puis renvoie une valeur dans la même ligne à partir d'une colonne spécifiée. Par défaut le tableau doit être trié par ordre croissant. »

Les paramètres de la formule :

RECHERCHEV(valeur_cherchée ; table_matrice ; index_col ; boolean)

valeur_cherchée : la valeur recherchée dans **table_matrice**, dans notre cas il s'agit du nom de la classe.

table_matrice : la sélection dans laquelle on recherche notre **valeur_cherchée**, dans notre cas il s'agit de la colonne « Classe » des deux autres traces.

index_col : le numéro de la ligne à partir de laquelle la recherche sera effectuée. Explication : la sélection dans laquelle on effectue la recherche d'une valeur contient plusieurs lignes, il faut donc indiquer le départ de la recherche.

boolean : paramètre un peu particulier, Excel propose deux valeurs pour ce paramètre : VRAI ou FAUX. VRAI pour correspondance approximative et FAUX pour correspondance exacte. C'est la correspondance exacte qui nous intéresse, choisissons donc FAUX.

Fonctionnement de la formule

On cherche le nom de la classe dans une colonne qui contient les classes d'une autre trace. À l'aide de la recopie incrémentée d'Excel nous pouvons facilement appliquer cette formule à une colonne entière des classes d'une trace. L'output de la formule peut prendre deux formes :

- Le nom de la classe recherchée, si cette dernière figurent dans la sélection.
- *« N/A »* si le nom de la classe ne figurent pas dans la sélection.

Ce sont des valeurs de type *« N/A »* qui nous intéressent, car ce que nous recherchons c'est le fait que la classe soit propre à la trace et donc on ne veut pas qu'elle figure dans les autres traces.

La recherche pour chaque trace est effectuée en trois temps. Soit la trace A correspondant à la stratégie JOINED et les traces B et C respectivement pour les deux autres stratégies de mapping.

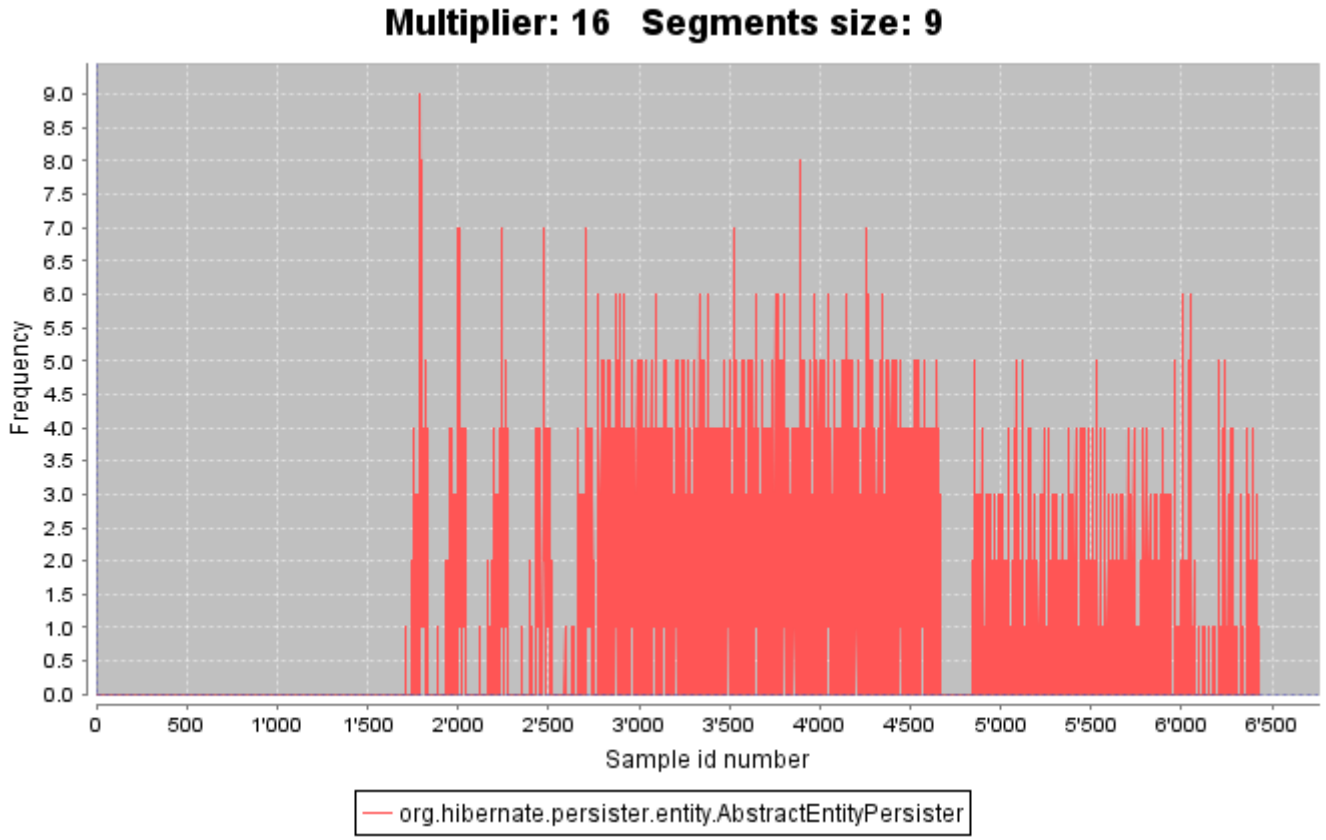
Étape 1 : Appliquer la formule à la trace B.

Étape 2 : Appliquer la formule à la trace C.

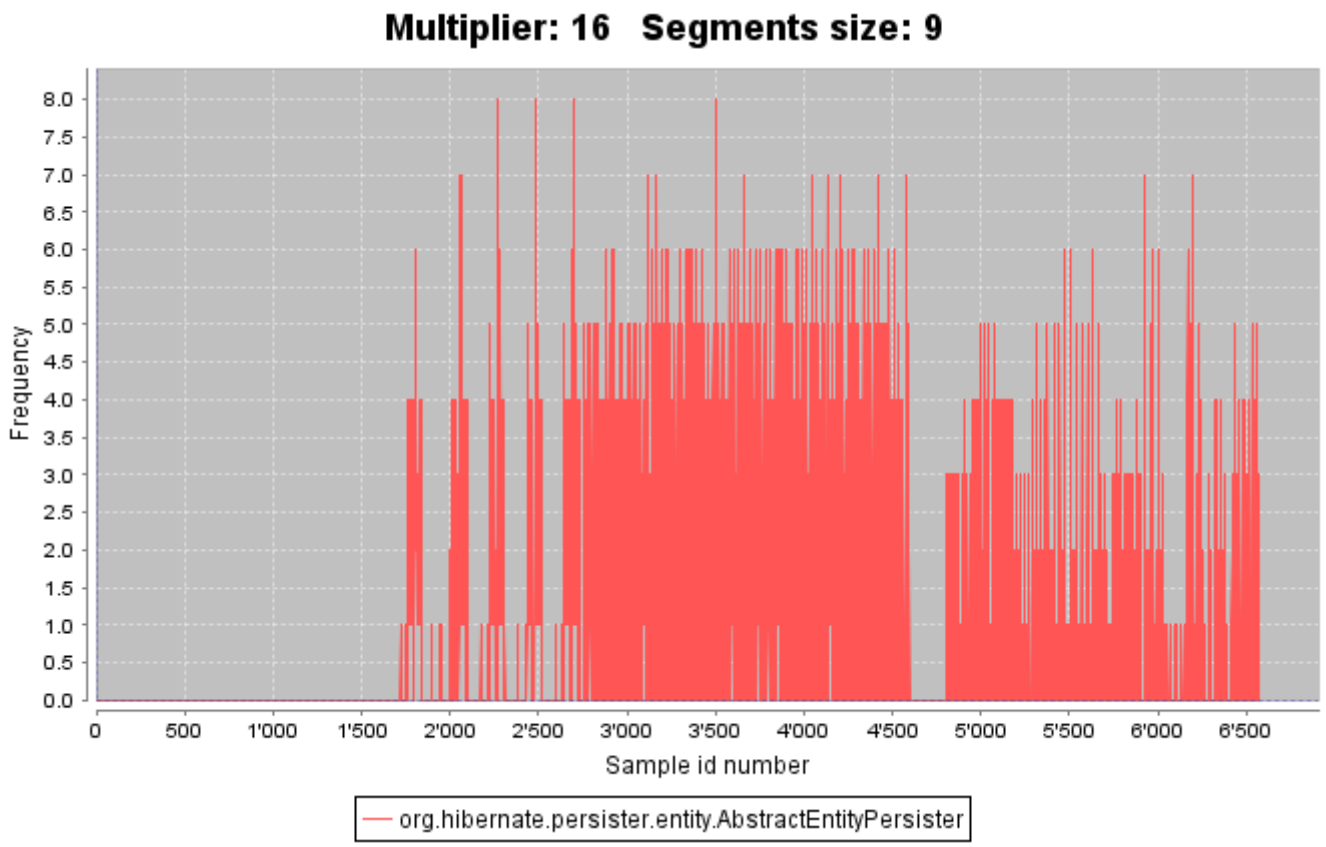
Étape 3 : Comparer à l'œil les différences.

La dernière étape est assez simple, il suffit tout simplement de repérer les *« N/A »* pour la même classe dans les deux output de la formule (étape 1 et 2). Si nous avons *« N/A »* dans les deux cas pour une classe quelconque, cela veut dire qu'elle ne figure ni dans la trace B ni dans la trace C.

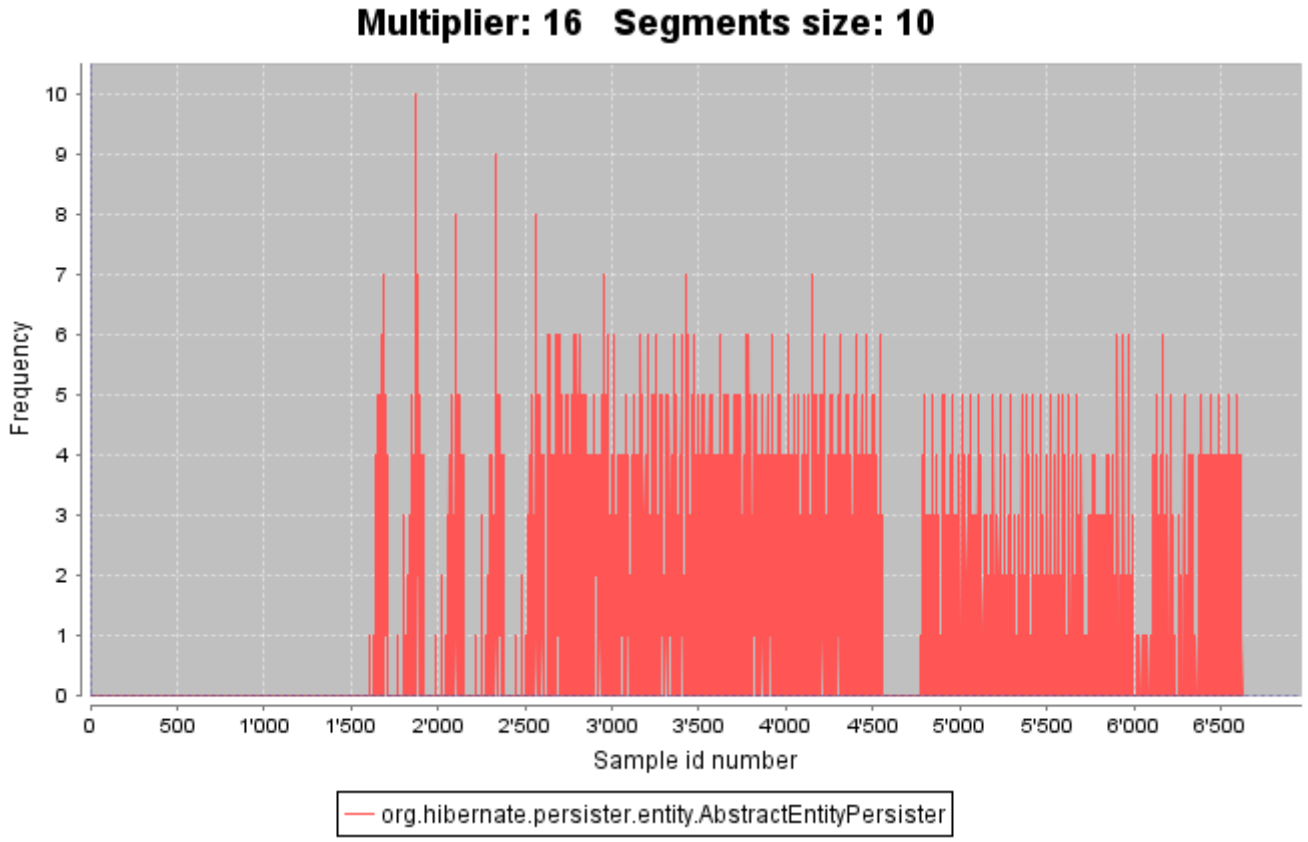
Annexe 8
Occurrence temporelle AbstractEntityPersister, stratégie SINGLE TABLE



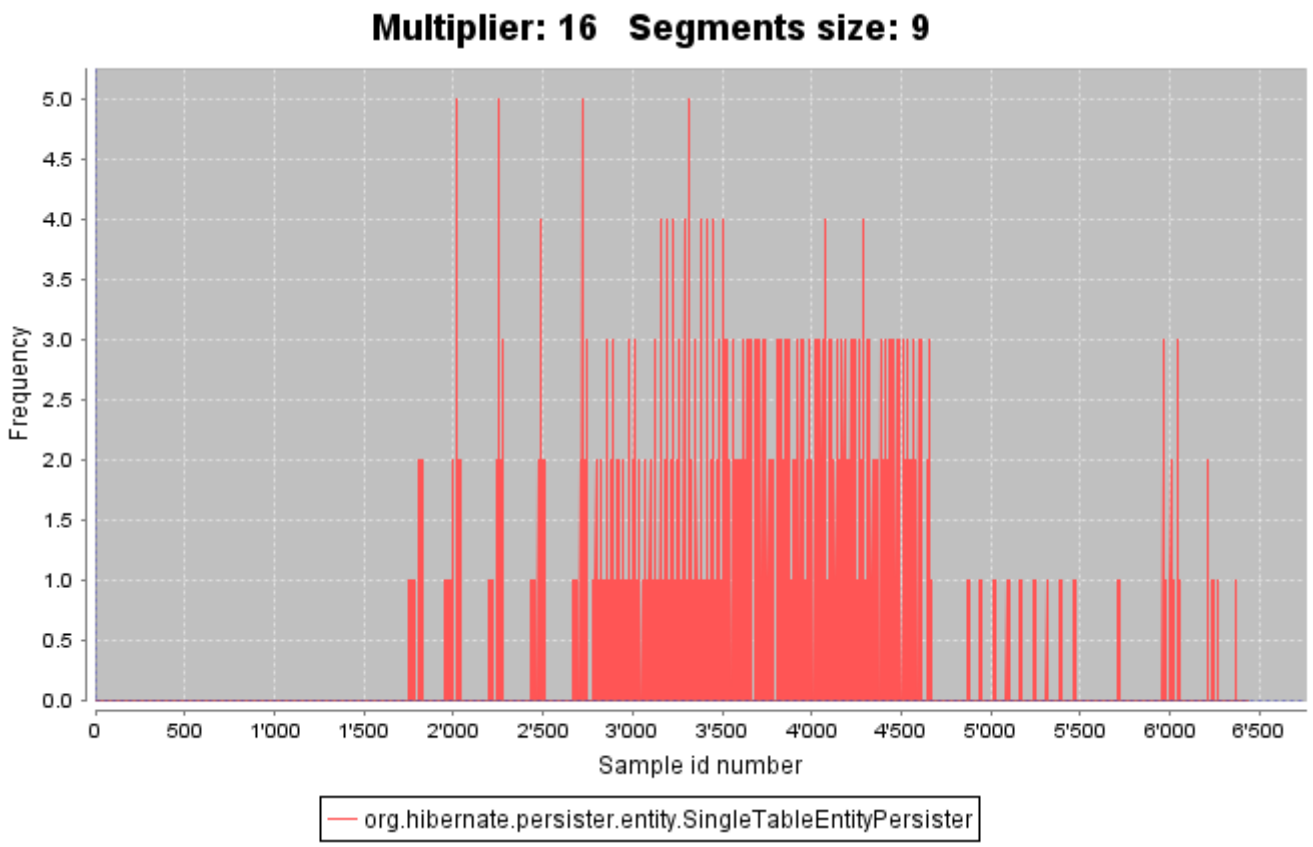
Annexe 9
Occurrence temporelle AbstractEntityPersister, stratégie TABLE PER CLASS



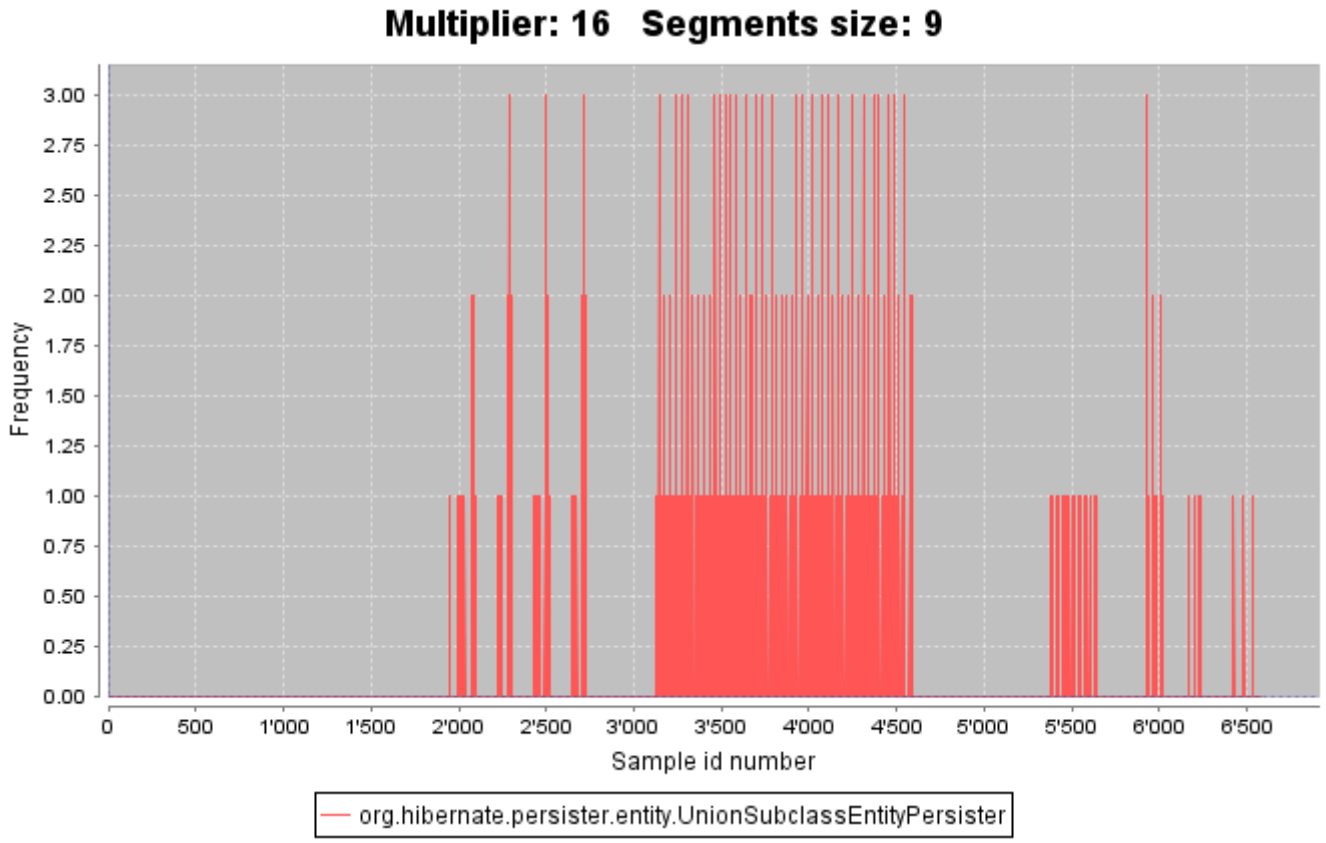
Annexe 10
Occurrence temporelle AbstractEntityPersister, stratégie JOINED



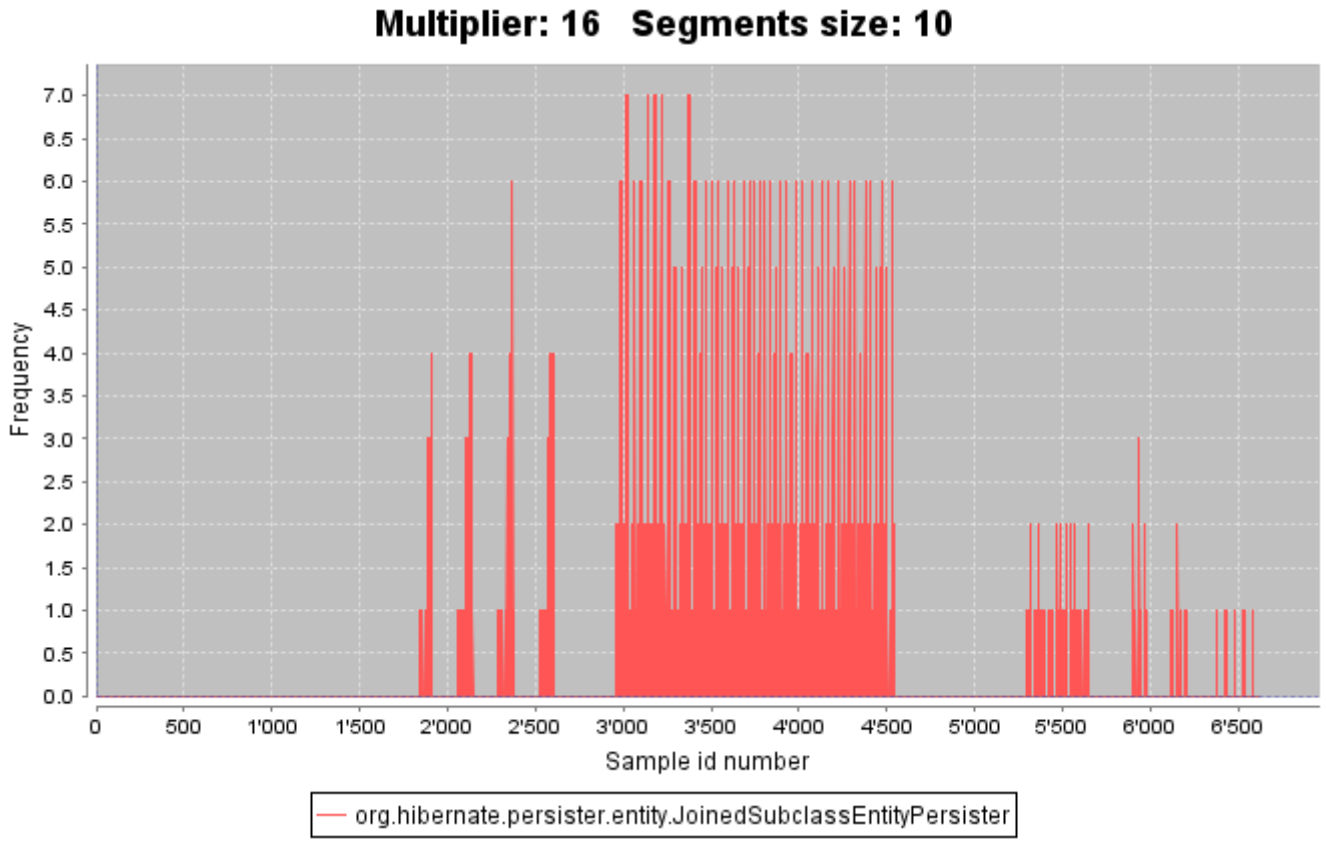
Annexe 11
Occurrence temporelle SingleTableEntityPersister, stratégie SINGLE TABLE



Annexe 12
Occurrence temporelle UnionSubclassEntityPersister, stratégie SINGLE TABLE

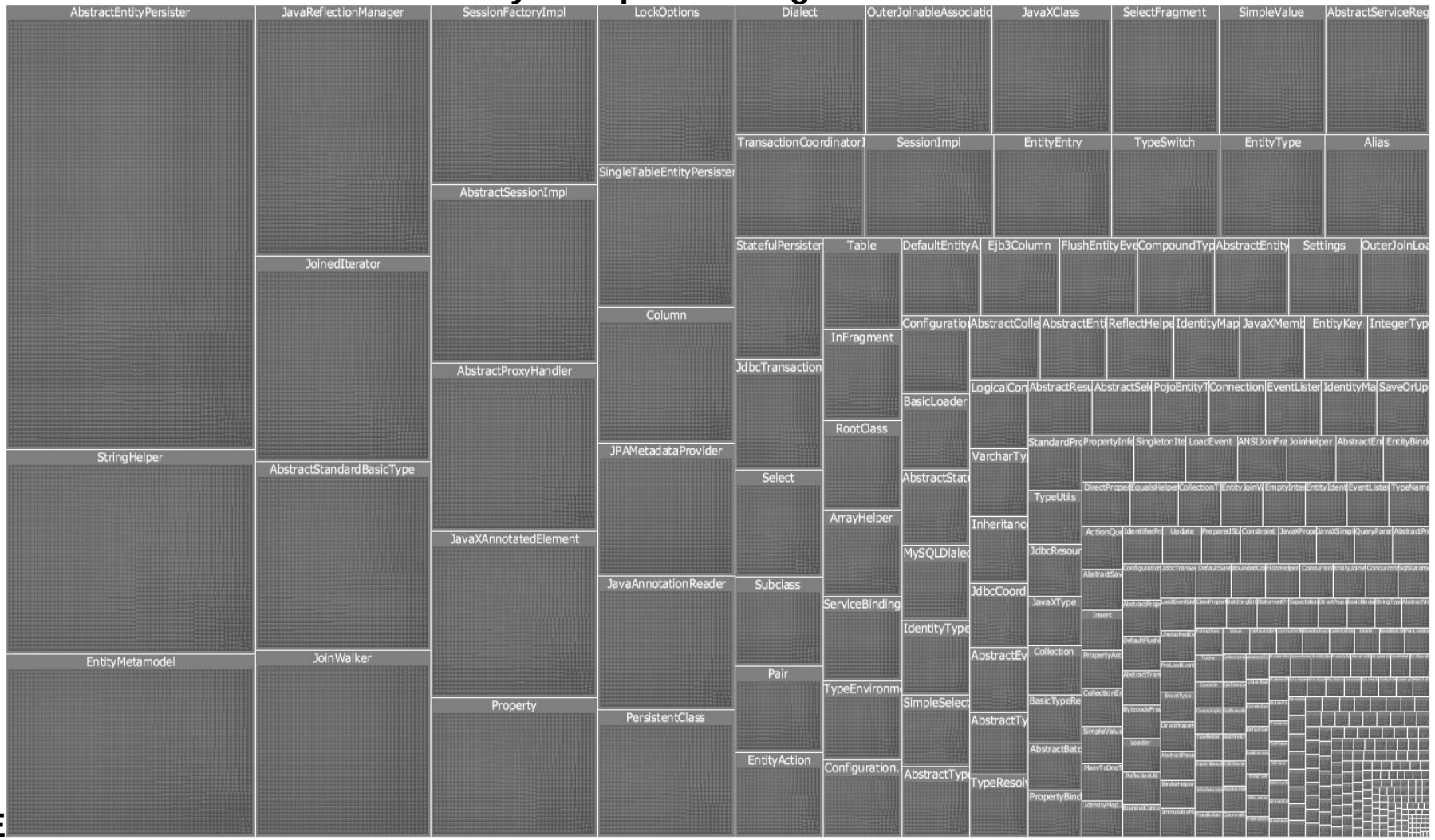


Annexe 13
Occurrence temporelle JoinedSubclassEntityPersister, stratégie SINGLE TABLE



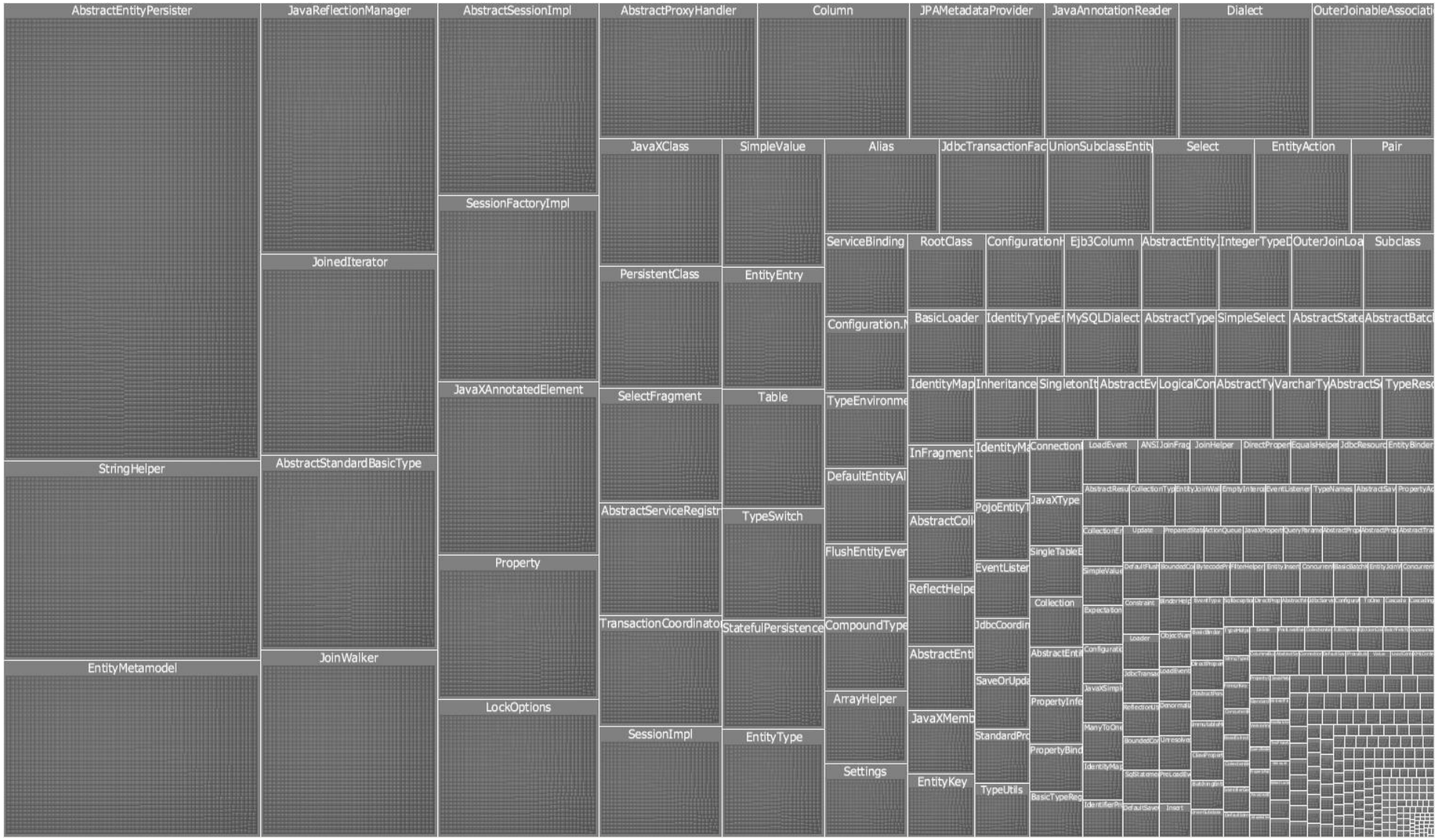
Annexe 14
Noyau fonctionnel de mapping Hibernate

Annexe 15
Classes dynamiques stratégie SINGLE

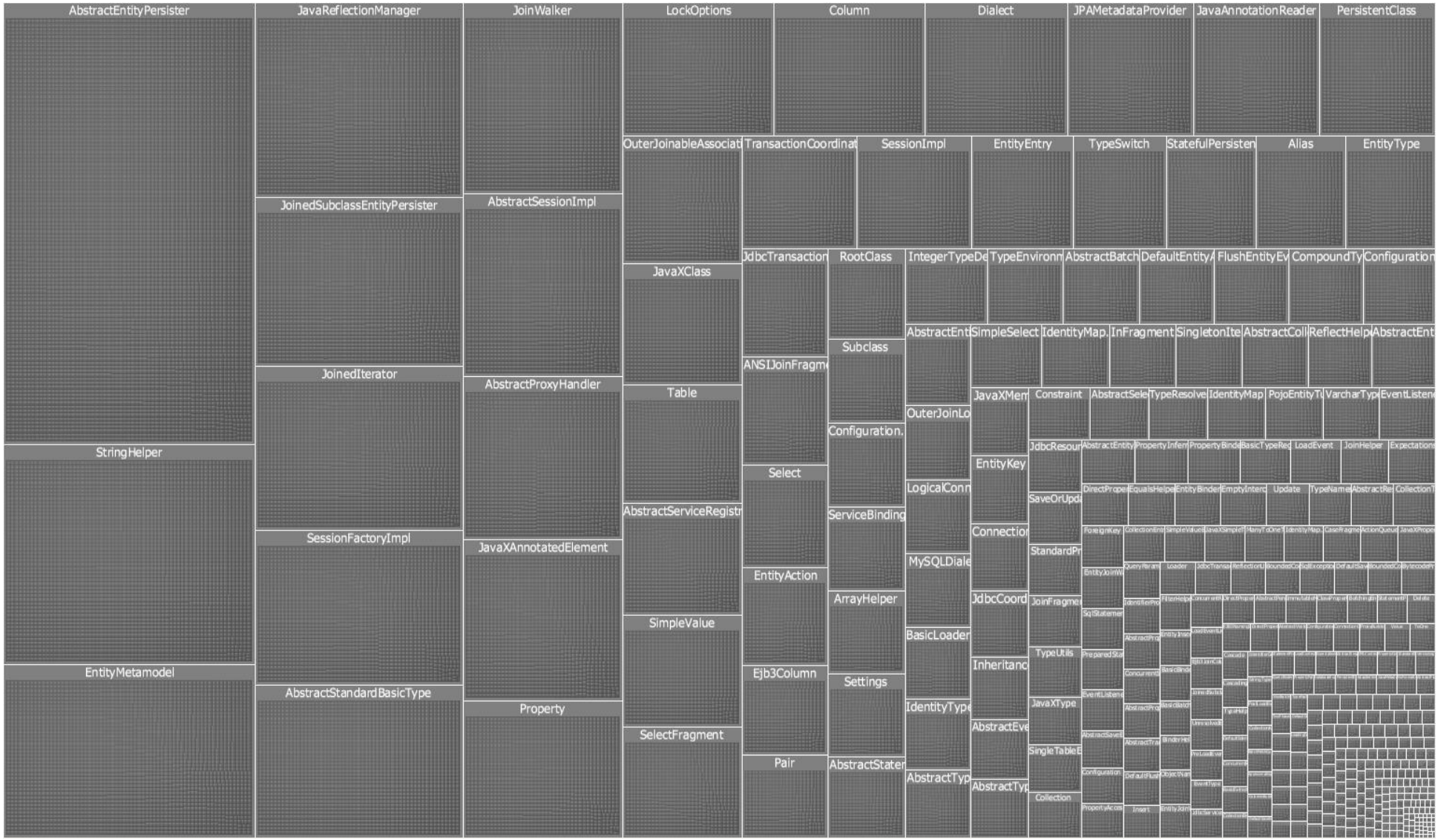


TABLE

Annexe 16
Classes dynamiques stratégie TABLE PER CLASS



Annexe 17
Classes dynamiques stratégie JOINED



Annexe 18
Code source Stratégie SINGLE TABLE

Nous avons mis à disposition seulement l'implémentation d'une seule stratégie, le reste du code peut être consulté sur CD.

```
package gomand.main;

import java.util.ArrayList;
import java.util.Iterator;


import gomand.dom.Client;
import gomand.dom.Moto;
import gomand.dom.Vehicule;
import gomand.dom.Camion;
import gomand.dom.Voiture;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class GarageMain {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		ArrayList<Vehicule> listeCamions = new ArrayList<Vehicule>();
		ArrayList<Vehicule> listeMotos = new ArrayList<Vehicule>();
		ArrayList<Vehicule> listeVoitures = new ArrayList<Vehicule>();
		Client client = new Client();
		client.setNom("Mark Favon");
		for (int i = 1; i <= 3; i++) {
			Camion camion = new Camion();
			camion.setNom("Camion numero " + i);
			camion.setProprieteCamion("camion");
			listeCamions.add(camion);
			Moto moto = new Moto();
			moto.setNom("Moto numero " + i);
			moto.setProprieteMoto("moto");
			listeMotos.add(moto);
			Voiture voiture = new Voiture();
			voiture.setNom("Voiture numero " + i);
			voiture.setProprieteVoiture("voiture");
			listeVoitures.add(voiture);
			client.getListeVehicules().add(camion);
			client.getListeVehicules().add(moto);
			client.getListeVehicules().add(voiture);
		}
		@SuppressWarnings("deprecation")
		SessionFactory sessionFactory = new Configuration().configure()
				.buildSessionFactory();
		Session session = sessionFactory.openSession();
		session.beginTransaction();
		for (int i = 0; i < listeMotos.size(); i++) {
```

```
                Moto moto = (Moto) listeMotos.get(i);
                session.save(moto);
            }
            for (int i = 0; i < listeCamions.size(); i++) {
                Camion camion = (Camion) listeCamions.get(i);
                session.save(camion);
            }
            for (int i = 0; i < listeVoitures.size(); i++) {
                Voiture voiture = (Voiture) listeVoitures.get(i);
                session.save(voiture);
            }
            for (int i = 1; i <= 5; i++) {
                Vehicule vehiculeFromBdd = (Vehicule) session
                        .get(Vehicule.class, i);
                vehiculeFromBdd.setNom("Modified");
                session.update(vehiculeFromBdd);
            }
            for (int i = 0; i < listeCamions.size(); i++) {
                client.getListeVehicules().removeAll(listeCamions);
                session.delete(listeCamions.get(i));
            }
            session.save(client);
            session.getTransaction().commit();
            session.close();
            session = sessionFactory.openSession();
            session.beginTransaction();
            Client clientFromBDD = (Client) session.get(Client.class, 1);
            System.out
                    .println("Nombre de véhicules du client de la base de données :
"
                            + clientFromBDD.getListeVehicules().size());
            Iterator itrFromBdd = clientFromBDD.getListeVehicules().iterator();
            while (itrFromBdd.hasNext()) {
                Vehicule vehicule = (Vehicule) itrFromBdd.next();
                System.out.println(vehicule.getNom());
            }
            session.getTransaction().commit();
            session.close();
        }
}
```

```
package gomand.dom;

import java.util.ArrayList;
import java.util.Collection;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;

@Entity
public class Client {
	@Id @GeneratedValue(strategy=GenerationType.AUTO)
	private int clientId;
	private String nom;

	@OneToMany
	private Collection<Vehicule> listeVehicules = new ArrayList<Vehicule>();

	public int getClientId() {
		return clientId;
	}
	public void setClientId(int clientId) {
		this.clientId = clientId;
	}
	public String getNom() {
		return nom;
	}
	public void setNom(String nom) {
		this.nom = nom;
	}
	public Collection<Vehicule> getListeVehicules() {
		return listeVehicules;
	}
	public void setListeVehicules(ArrayList<Vehicule> listeVehicules) {
		this.listeVehicules = listeVehicules;
	}
}
```

```
package gomand.dom;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;

@Entity @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public abstract class Vehicule {

	@Id @GeneratedValue(strategy=GenerationType.AUTO)
	private int vehiculeId;
	private String nom;
	public int getVehiculeId() {
		return vehiculeId;
	}
	public void setVehiculeId(int vehiculeId) {
		this.vehiculeId = vehiculeId;
```

```
	}
	public String getNom() {
		return nom;
	}
	public void setNom(String nom) {
		this.nom = nom;
	}

}
```

```
package gomand.dom;

import javax.persistence.Entity;

@Entity
public class Camion extends Vehicule {
	private String proprieteCamion;

	public String getProprieteCamion() {
		return proprieteCamion;
	}

	public void setProprieteCamion(String proprieteVelo) {
		this.proprieteCamion = proprieteVelo;
	}

}
```

```
package gomand.dom;

import javax.persistence.Entity;

@Entity
public class Moto extends Vehicule {
	private String proprieteMoto;

	public String getProprieteMoto() {
		return proprieteMoto;
	}

	public void setProprieteMoto(String proprieteMoto) {
		this.proprieteMoto = proprieteMoto;
	}
}
```

```
package gomand.dom;

import javax.persistence.Entity;

@Entity
public class Voiture extends Vehicule {
```

```
	private String proprieteVoiture;

	public String getProprieteVoiture() {
		return proprieteVoiture;
	}

	public void setProprieteVoiture(String proprieteVoiture) {
		this.proprieteVoiture = proprieteVoiture;
	}

}
```

<!--FICHIER HIBERNATE.CFG.XML-->

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->
        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://127.0.0.1/garage</property>
        <property name="connection.username">root</property>
        <property name="connection.password"></property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>

        <!-- Disable the second-level cache  -->
        <property
name="cache.provider_class">org.hibernate.cache.internal.NoCacheProvider</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">create</property>

             <mapping class="gomand.dom.Moto"></mapping>
             <mapping class="gomand.dom.Vehicule"></mapping>
             <mapping class="gomand.dom.Camion"></mapping>
             <mapping class="gomand.dom.Voiture"></mapping>
             <mapping class="gomand.dom.Client"></mapping>
    </session-factory>

</hibernate-configuration>
```