

Master Thesis Software Engineering

Wednesday, August 25, 2010

Finding Performance Issues in Hibernate Usage by Static Code Analysis

by

Jasper Alblas
University of Amsterdam

Supervisor: Paul Klint

Company supervisors: Yaroslav Usenko & Sabrina Filemon

Publication status: Public

Version: 1.0



Abstract

An ORM (Object/Relational Mapping) implementation handles the persistence of data in a software application. An ORM implementation is a complex beast [1]. It can occur that software developers make wrong choices while developing software that uses ORM. These wrong choices can result in poor performance of the whole system.

Hibernate is an often used ORM implementation. It would be very useful to know patterns, in the source code, of Hibernate usage that cause poor performance. If these patterns are known, a software developer can check for these patterns and optionally improve the performance of his implementation. If these patterns can be detected by static code analysis, the developer can automate this check.

In this paper we make a start by defining one pattern that can result in poor Hibernate performance. We show that this pattern occurs in real life software projects and we show that we can gain performance by replacing the source code by code that does not match the pattern.

1 Introduction

During our internship at KPMG CT we analyzed various Java software projects that uses Hibernate as ORM-tool and we studied the Hibernate documentation. We tried to define a pattern that is useful for developers to know if that pattern exists in their software. What is a useful pattern?

A useful pattern is: (1) a pattern that occurs in real life software project and (2) if you replace the source code by code that does not match the pattern, the Hibernate performance should improve.

If we succeed in defining a useful pattern we want to know if this pattern can be detected statically. We also wanted to know how accurate this detection is, in other words: How many of the found 'warnings' are actually performance issues?

We formulate the following research questions:

Research question 1: Is it possible to find a pattern of Hibernate usage in source code, which can affect the Hibernate performance, by static code analyses?

To answer this question we needed to answer two other research questions.

Research question 2: Is it possible to define a pattern that occurs in real life software projects and has performance improvement potential?

Research question 3: Is it possible to find this pattern by static code analysis with an acceptable amount of false positives?

We will focus on the fact that performance improvement is possible; we do not determine how much performance improvement is possible in a real life application.

1.1 Scoping

We scoped our static analyses to the Java files (not the Hibernate XML-mapping files) therefore we had to make a few assumptions about those mapping files.

A persistent object can be in four different states [1]:

1. Transient: objects that are instantiated using the new

operator. The object is only in memory not in the database.

2. Persistent: objects that have a database identity. Changing the object in memory results in a change in the database.
3. Detached: an uncoupled persistent object. Closing a session uncouples persistent objects. Changing the object in memory results not in a change in the database.
4. Removed: if an object is scheduled for deletion at the end of a unit of work but it is still managed by the persistent context until the end of the unit of work.

See [1] chapter 9 for more information about the different states of persistent objects. In our analyses we assume that all the persistent objects are always in the persistent state. We assume a worst-case scenario, every modification results in a database hit.

Hibernate has the functionality to retrieve (parts of) an object only when it is accessed in the source code. This is called lazy loading/fetching [1]. In our analyses we assume that all the persistent classes are mapped with lazy="false". This means that Hibernate retrieves the whole object from the database when you query for it.

Hibernate defines the following fetching strategies [4]:

- Join fetching: Hibernate retrieves the associated instance or collection in the same SELECT, using an OUTER JOIN.
- Select fetching: a second SELECT is used to retrieve the associated entity or collection. Unless you explicitly disable lazy fetching by specifying lazy="false", this second select will only be executed when you access the association.

- Subselect fetching: a second SELECT is used to retrieve the associated collections for all entities retrieved in a previous query or fetch. Unless you explicitly disable lazy fetching by specifying lazy="false", this second select will only be executed when you access the association.
- Batch fetching: an optimization strategy for select fetching. Hibernate retrieves a batch of entity instances or collections in a single SELECT by specifying a list of primary or foreign keys.

See [4] chapter 19 for more information about the different fetching strategies. In our analysis we assume select fetching, which is the default of Hibernate.

Hibernate has a first and a second level cache [4]. We assume that both the caches are empty or disabled if possible. Here again we assume the worst-case scenario.

1.2 Content

The remainder of this paper is organized as follows: Chapter 2 describes the previous research that is used to formulate the hypotheses and the pattern. Chapter 3 describes the Java optimisation framework Soot that we use to collect static information about the application under analyzes. Chapter 4 shows that the pattern has performance improvement potential. Chapter 5 focuses on the algorithm /method we used to detect this pattern statically. Chapter 6 describes the research we did to test our hypotheses. Chapter 7 describes the concluding remarks on this research, and finally we indicate future work in Chapter 8.

2 Previous research

A lot of research is done on increasing performance in software projects. Some of these researches include finding performance issues or bugs (whatever you like to call them, we prefer the term performance issues) by using static code analysis.

We based the hypotheses and the pattern basically on two pieces of work. We focused on the work of Christian Bauer and Gavin King [1] and the Hibernate Reference Documentation [4]. Both references contain a lot of information about how Hibernate works and how a software developer should use Hibernate. Besides that, these references also describe how a software developer should not use Hibernate.

To improve our knowledge about ORM performance tuning we studied Stanley et al. [11] and Pieter van Zyl et al. [12, 14]. They taught us that Hibernate settings such as lazy loading, cascading and fetching strategies have a big influence on the performance of Hibernate. We had to define those variables in our pattern or we had to make assumptions about them. We chose the latter option (see section 1.1).

Pieter van Zyl et al. [14] shows that a query can be written in a different way to improve the retrieval speed.

Jeroen Bach has shown some performance improvement opportunities, relating to Hibernate in his Master thesis [2]. However, all of Bach's performance improvements need dynamic analysis and were therefore unachievable in this research.

In the following sections we formulate our hypotheses. Every hypothesis is linked to a research question. In section 2.2 we define the pattern.

2.1 Static analysis

Doing static Java code analysis with SOOT gives us information about the

application under analysis. SOOT has been used in the past to analyze Java projects [5]. We expect that SOOT will give us the information we need to detect the pattern. Therefore we formulate the following hypothesis.

Hypothesis 1: It is possible to find a pattern of Hibernate usage in source code, which can affect the Hibernate performance, by static code analyses in a software project by analyzing the information about that application that is given to us by SOOT (Research question 1).

We describe what information is given to us by SOOT in the following chapter. In chapter 5, we describe our pattern detection algorithm. The pattern is described in the following section.

2.2 The pattern

There are various ways to affect Hibernate performance, two important points are: (1) Don't fetch data you don't need and (2) use the right ratio between the number of queries sent and the amount of data you query in one query [1].

The first point has all to do with a good mapping plan and a good fetching strategy. This mapping plan and fetching strategy is set in the XML-mapping files (or annotations). You have to check if this plan and strategy are matching with the way you use the (persistent) data in your code [1, 4].

To check your software for non-optimal performance relating to point (2), you have to check all the locations in the code where queries are sent, especially at the locations in the code where multiple queries are sent.

Hypothesis 2: When searching for places in the application where a query is sent/generated in a loop

we search for a useful¹ pattern that we can detect statically (Research question 2).

The pattern of “sending/generating a query in a loop” can cause poor Hibernate performance [1]. This means that it is not always wrong to use this pattern in your software, but in some cases it is. This brings us to the following subject: False Positives.

2.3 False Positives

When searching for pattern matches in the application we agreed that having false positives is acceptable. These false positives are warnings that point to source code that matches the pattern but this source code does not imply a performance issue. This can be the case for various reasons; in chapter 6 we divide the false positives in groups.

To make the result of the static analysis useful for a software reviewer we had to reduce the number of false positives. Therefore we formulate the following hypothesis.

Hypothesis 3: At least 25% of the found matches with the pattern are performance issues and the performance can be improved by removing the pattern (Research question 3).

Reducing the number of false positives is an important task to make the warning document useful. Further in this paper we divide these false positives in groups in order to increase the chance of eliminating those false positives in future research and implementations. Now let's take a look at the methods that send or generate a query.

¹ See chapter 1 for the definition of 'useful'.

2.4 Database-hit methods

There are various general structures how one can generate and/or send multiple queries by using a loop. To statically detect this we have to know which Hibernate methods send or generate a query.

Certain methods of the Session and Query interface will result in sending and generating a query; in the next section we name and describe those methods and we mention why they cause poor performance when you call them in a loop.

According to Christian Bauer and Gavin King [1] it results in sending or generating a query if you call some methods of the Session and Query interface or a setter of a field of a persistent class in a loop. The set of methods that send or generate a query contains the following methods:

- Object Session.get(...)
- Object Session.load(...)
- Serializable
Session.save(...)
- Void
Session.saveOrUpdate(...)
- Void Session.update(...)
- Void Session.delete(...)
- List Query.list()
- Object
Query.uniqueResult()
- ScrollableResults Query.
Scroll(...)
- Iterator Query.iterate()
- All the setters of fields of persistent classes

In section 2.4.1, 2.4.2 and 2.4.3 we look at each method specifically.

2.4.1 Session interface

2.4.1.1 get and load

The methods `get` and `load` both query for data from the database. If you call the method `load` and you give the type and id as parameter it will return

the object from the database from the right table with the matching id.

The difference between the `get` and `load` methods is as follows: `get` always returns the real object and `load` returns a proxy object² if possible. A proxy object is fully initialized if you call any other method than the `getId` method of the object.

The method `get` will always generate and send a query. The `load` method will only send a query when you access any of his fields other than the `id`-field. This means that multiple queries will be sent when using `get` in a loop and using `load` in a loop (and accessing fields of the loaded object).

2.4.1.2 `save`, `update` and `saveOrUpdate`

Calling the method `save` or `update`, inserts or updates the given object in the database. Both methods generate a query. The query is sent when the method `flush` (from the session interface) is called. Calling one of these methods in a loop will also result in multiple queries to the database.

However with these set of methods it is almost impossible to replace the source code with code that does not match the pattern. Because it often requires refactoring that reduces the readability of the software.

2.4.1.3 `delete`

Calling the method `delete`, deletes the given object from the database. The method will generate and send the query to the database. Calling the `delete` method in a loop will result in multiple queries to the database.

² An object that looks like the real thing but only contains the `id` value [1]. No database hit is needed to retrieve this proxy object.

2.4.2 Query interface

2.4.2.1 `list`, `uniqueResult`, `scroll` and `iterate`

When calling these methods you execute the `query` object. It triggers a sending query action. They return a list of objects, a single object, a `ScrollableResult` of objects or an iterator over the objects from the database [7]. Calling one of these methods in a loop will result in multiple queries to the database.

2.4.3 Setters

Hibernate keeps persistent objects that are in a persistent state synchronized with the database. Setting a field of a persistent object (that is in a persistent state) will result in a query to the database. Setting one of these fields in a loop will result in multiple queries to the database.

Now that we know what (which pattern) we are actually looking for in the application we take a look at our input data, provided by SOOT.

3 SOOT

SOOT is a Java optimisation framework [5, 9]. We only used some specific functionality from SOOT. We used SOOT to extract information from the Java source code by doing static analysis. We used part of the Scene data model (including the Jimple body) and the call graph that is generated by SOOT.

3.1 *SOOT Scene*

The SOOT Scene manages the SootClasses of the application being analyzed [8]. It is a model of the source code from the application being analyzed. The Scene has information present such as: Which classes contain which methods, and which classes extend which classes, the Jimple body of a method, etc. Because we don't need all

the information the scene provides, we filter the information and only save the information we need. We filter and restructure some of this information to improve speed when analyzing the application.

In chapter 5 we describe which information we use from the Scene.

```
public int stepPoly(int x)
{
    if(x < 0)
    {
        System.out.println("foo");
        return -1;
    }
    else if(x <= 5)
        return x * x;
    else
        return x * 5 + 16;
}
```

Listing 1 - Java method

```
public int 'stepPoly'(int)
{
    Test r0;
    int i0, $i1, $i2, $i3;
    java.io.PrintStream $r1;

    r0 := @this;
    i0 := @parameter0;
    if i0 >= 0 goto label0;

    $r1 = java.lang.System.out;
    $r1.println("foo");
    return -1;

label0:
    if i0 > 5 goto label1;

    $i1 = i0 * i0;
    return $i1;

label1:
    $i2 = i0 * 5;
    $i3 = $i2 + 16;
    return $i3;
}
```

Listing 2 - Jimple method

3.2 Jimple body

Jimple is an unstructured representation for Java bytecode. It is developed to allow optimizations and analyses to be performed on Java bytecode at the most appropriate level [9]. It is an ideal form for performing analyses [9].

We found the Jimple format very suitable for detecting loops and

analyzing its content. In Listing 1 a Java method is shown, its corresponding Jimple Body is shown in Listing 2.

3.3 Call graph construction

SOOT has the functionality of constructing a call graph from the application. SOOT can construct different kinds of call graphs. The simplest call graph is obtained through Class Hierarchy Analysis (CHA). CHA is simple in the fact that it assumes that all reference variables can point to any object of the correct type [6].

SOOT also has the functionality of constructing call graphs by using points-to analysis (or data flow analysis), for memory reasons the use of this kind of analysis was unachievable. In this research we used the CHA call graph, constructed by SOOT.

4 The pattern's performance tests

To answer part of our second research question we have to show that the pattern has performance improvement potential. We have executed some artificial examples to show the improvement potential of the pattern. Each of these artificial examples contains an instance of the pattern. We have executed the example and measured the number of queries sent to the database. Afterwards, we reimplemented the example (with the same functionality) with code that does not match the pattern. We executed the example again and measured the number of queries that were now sent to the database.

In appendix A we show that it is possible to modify the code of four general implementations of the pattern where a query is sent in a loop to improve speed (and reduce the number of queries sent). The four implementations of the pattern are:

	Load	Query	Set	Delete
With pattern	50000 queries	50000 queries	50000 queries	50000 queries
Without pattern	200 queries	200 queries	500 queries	200 queries
With pattern	26,5 sec	362,5 sec	16,7 sec	19,6 sec
Without pattern	10,3 sec	9,6 sec	3,6 sec	9,7 sec

Table 1 – Result performance tests

- Loading an object from the database in a loop.
- Querying for data in a loop.
- Setting a field of a persistent class in a loop.
- Deleting a persistent object in a loop.

In Table 1 we present the results of our artificial performance tests on the four occurrences of the pattern. In all four cases the number of queries sent after reimplementing the code is at least reduced by a factor of 10. To retrieve the same amount of data these queries are in general bigger (take more time to execute) than the queries sent when the code was still containing the pattern. But, as we can see in the last two lines, the implementation without the pattern takes also less time to execute.

These four artificial performance tests show that the pattern seems to have performance improvement potential.

5 Pattern detection

After we have established that the pattern has some performance improvement potential we focus on the algorithm/method we use to detect this pattern statically. In section 5.1 we describe how we have implemented the algorithm.

We have two main input streams from SOOT. The first one is the data model SOOT scene³. This data model contains the following information:

- All classes and all their methods, fields, names, super classes and interfaces.

All methods and their method body (Jimple, including line numbers), method signature and if the method is from an application class or library class.

The second one is the CHA call graph³. We used a CHA call graph to avoid a memory problem. The CHA call graph simply assumes that every variable might point to ever other variable which is conservatively sound but not terribly accurate [6].

We analyze the transitive CHA call graph and create a set of methods that can call at least one of the earlier named methods (or one of their implementations) from the session and query interface. We refer to this set as: “db-hit methods”.

We detect the persistent classes by analyzing the XML mapping files⁴ that are required when using Hibernate. We detect the setters of these persistent classes by analyzing the Jimple body and we collect all the methods that can call those setters in the transitive call graph and add these methods to the set of db-hit methods.

By analyzing the “goto’s” in the Jimple body we can detect where a loop starts and where it ends. We determine the content of every loop. By processing that content we fill the set of methods that are called in a loop (direct and indirect

³ See chapter 3.

⁴ This is the only information we do read from the XML mapping files.

by using the transitive call graph). We refer to this set as: “in-loop methods”.

By taking the intersection of those two collections (the db-hit methods and the in-loop methods) we can determine the places in the software where suspicious methods are called in the software.

Because the in-loop method set also contains information about the location of the methods we can generate a proper warning if a match is found.

By taking the intersection of those two collections (the db-hit methods and the in-loop methods) we are sure we find all the locations in the code where we could find a performance issue that is relating to this pattern. But on the other hand, in this way we produce a certain number of false positives.

We found that the number of false positives in the warnings is unacceptable. To make the warning report more useful we have separated all the warnings programmatically in three categories. The algorithm we used is explained in chapter 6.

5.1 Implementation

To test our hypothesis we have implemented this pattern detection algorithm. The main functionality of this tool is: pointing out which code matches the pattern. The tool generates a warning document that contains all the warning messages.

5.1.1 Architecture

The tool has the following three components:

- Soot component: This part of the tool executes Soot and delivers the extracted data to the Data component.
- Data component: This part of the tool manages the data that is extracted from the application under analysis.
- Detection component: This part of the software analyses the data

in the Data component and detects loops and method calls.

The implementation was written in Java and has 2671 Lines of code, 10 classes and 169 methods.

6 Research

To test our hypotheses we have set up a research environment which we discuss in the section 6.2.

First we analyzed two projects (MKB project and SMB project) with our tool. Second we analyzed all the generated warning messages by hand. Both of the projects are web-based and they have unacceptable database performance.

6.1 Projects and DAO-design

The interaction with the database in both projects is done with the use of the open source ORM-implementation Hibernate [1]. In the MKB project the Database Accessing Objects (DAO) is the code that is responsible for the database interaction (the code that calls Hibernate). These objects are the real layer between the business software and the persistent layer. This design is called DAO-design [10]. To create and send the query the DAO-class calls different Hibernate functions. DAO-classes are the only (application) classes that call Hibernate functions in a properly implemented DAO-architecture.

6.2 Research environment

To test our hypothesis we had to check if our pattern detection method finds performance issues in existing real life software projects. We had to check what the generated warning messages were and how many were false positives, recalls and actually bugs or performance issues.

We implemented the pattern detection algorithm as described earlier and

applied the algorithm to two projects. The MKB and the SMB project.

6.3 Research results

Our pattern detection algorithm generates a warning message for every occurrence of the pattern it found in the code. We have analyzed those warning messages and divided them in different groups and categories.

6.3.1 The categories

As said before: we found that the number of false positives in the warnings is unacceptably high. To make the warning report more useful we programmatically separated all the warnings in three categories.

- Category *one* contains all the warnings that are most likely to cause bad performance. It contains all the cases where the source calls (indirectly) one of the db-hit methods and the shortest path doesn't contain a node/method that is in a library class.
- Category *two* contains all the warnings that are likely to cause bad performance. It contains all the cases where the source calls (indirectly) one of the setters of a persistent class and the shortest path doesn't contain a node /method that is in a library class.
- Category *three* contains all the warnings that are most likely to be false positives. It contains all the cases that do not match one of the above criteria.

The categories should be used as follows: checking the warnings in category one, if no improvement is made the software reviewer continues by checking the warnings in category two. Only if the performances are still unacceptable he or she can check the warnings in category three.

6.3.2 The groups

To analyze our results better we manually separated the warnings in 5 groups. We describe the groups below.

6.3.2.1 False Positive Readability

Reimplementing the code that matches the pattern so that the query is sent outside the loop will have an unacceptable amount of impact on the readability of the source code.

6.3.2.2 False Positive Call Graph

We use the earlier mentioned CHA call graph. This call graph assumes that all reference variables can point to any object of the correct type [6]. This means that it can appear that our software concludes that a certain (suspicious) method is called, when this is in fact is not the case.

6.3.2.3 False Positive Assumption

We made some assumptions⁵ when developing the algorithm, such as: a persistent object is always in the persistent state. It can happen that these assumptions are wrong and the found warning should not be a warning because -for example- the persistent object is NOT in persistent state at that moment in the code.

6.3.2.4 False Positive Dead Code

The code that matches the pattern is never executed.

6.3.2.5 Recall

Two different warnings can point to the same performance problem in the code. If a loop is executed in another loop (in a different method) and in that loop a suspicious method is called.

⁵ See section 1.1.

6.3.2.6 Right

The last group contains the warnings that point to a pattern in the code that can be removed and causes performance improvements and does not fit in one of the other groups. These are the correctly found performance issues in the application.

6.3.3 MKB results

Found: 394 warnings (see Table 2).

6.3.4 SMB results

Found: 142 warnings (see Table 3).

6.3.5 Discussing the research results

6.3.5.1 Usefulness

According to the analyses carried out on these two projects we can conclude that the pattern exists in real life software projects. We have already concluded (in chapter 4) that the

pattern has performance potential. Now we can conclude that **the pattern seems useful**.

6.3.5.2 The categories

Figure 1 and Figure 2 show the total group distribution of both projects.

In appendix B we give more pie charts, they show the warning distribution by category and the group distribution by category. We see in all these pie charts that the distribution in both the MKB project and the SMB project is roughly similar.

Interesting to see in Table 2 and Table 3 is that the percentage of right warnings per category in both projects is decreasing. In the MKB project in category one, 50,0% of the warnings is right. In category two 23,8 % of the warnings is right. In category three only 1,5% of the warnings is right.

	Category 1	Category 2	Category 3	All Categories
F.P. Readability	7	8	16	31
F.P. Call graph	0	0	282	282
F.P. Assumption	0	19	9	28
F.P. Dead code	0	4	12	16
Recall	0	1	14	15
Right	7	10	5	22
Total	14	42	338	394

Table 2 - MKB results

	Category 1	Category 2	Category 3	All Categories
F.P. Readability	4	14	3	21
F.P. Call graph	0	0	91	91
F.P. Assumption	0	9	0	9
F.P. Dead code	0	0	6	6
Recall	0	2	0	2
Right	3	6	4	13
Total	7	31	104	142

Table 3 - SMB results

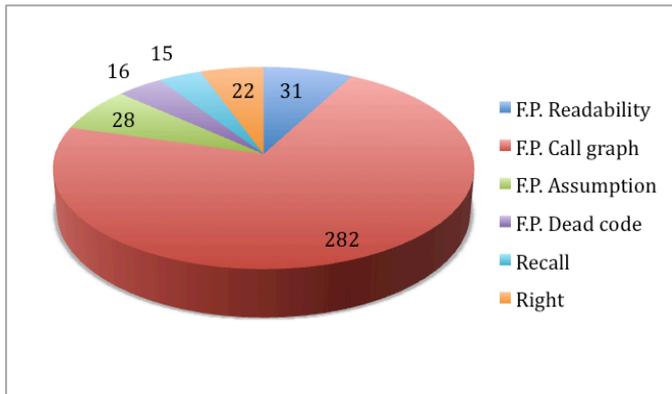


Figure 1 – Group distribution MKB project

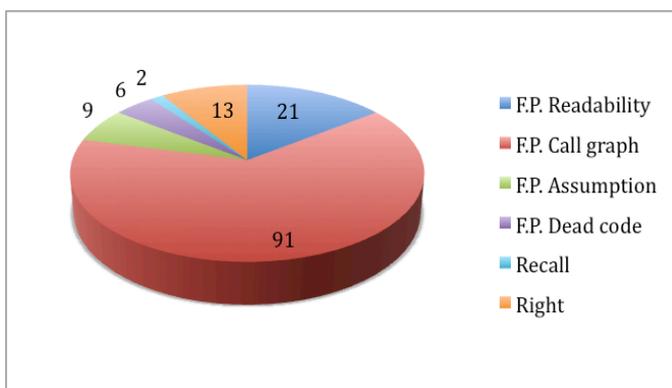


Figure 2 – Group distribution SMB project

By just analyzing the warning messages from category one and two you find 77,3% (MKB) and 69,2% (SMB) of the warnings by analyzing only 14,2% (MKB) and 26,7% (SMB) of the warnings. So we can conclude that in these two projects **categorizing the warnings improves the usefulness of the warning document.**

In these two projects the software is expected to have better performance after removing the patterns pointed out by the warnings in category one and two.

6.3.5.3 False positives

In the MKB project we found that 90,6% of the warnings was false positive and in the SMB project we found that 89,4% of the warnings was false positive. Both these percentages are more than the 75% mentioned in Hypothesis 3. However **if we forget**

about category three and only look at category one and two we see that in the MKB project 67,9% is false positive and in the SMB project this is 71,1%.

6.3.5.4 Call Graph

In both projects the biggest amount of warnings is a false positive because of the use of the CHA Call Graph.

If somewhere in the code a method of the interface collection is called the call graph construction software assumes that any implementation of this interface method can be called. The graph construction software also assumes this for the implementations made by Hibernate, and some of these implementations cause a database hit. So every time a method of the collection interface is called the pattern detection software marks that as a warning. And

in some cases it is a performance issue but in most cases it is not.

6.3.5.5 Readability

We noticed that **increasing performance (by moving the send query action outside the loop) often decreases the readability** of a method unacceptably and with that it decreases the readability of the whole software system (unacceptably). This is an interesting trade off. It is hard to filter these warning programmatically because this needs the opinion of a real person.

6.3.6 Discussing the right warnings

In this section, we take a look at some warnings from the group “right warnings”. We give concrete examples of how one can change the source code to improve the performance.

6.3.6.1 Example: Loading in loop

A warning that we encountered a few times is: loading an object from the database inside a loop. Loading an object from the database inside a loop results in n queries to the database, with n being the times you run through the

loop. In Listing 3 we show a method from the MKB project. We encountered this code of loading an object from the database in a loop also in code that triggered other warning messages. In Listing 4 we show how you can reduce the number of queries to the database and still have the same functionality. In Listing 4 you see code that sends just one query and has the same functionality.

In Listing 4 we first save all the id's that are needed to be retrieved from the database, second we send one big query that retrieves all the information and third we assign the right value to the right object.

The code has become a bit more complex in Listing 4 but that complexity is acceptable, it is still good readable. But imagine that the original code was already complex or that the retrieval of the id's and the loading action is not happening in the same method. You can always implement this way of retrieving collections but there is a trade off between readability of the code and the performance

```
private void loadEmployees(CreateInstanceCommand createInstanceCommand) {
    for (TaskInstanceCommand command : createInstanceCommand.getDetailsAsSet()) {
        LOG.info("Id of user to load: " + command.getUserId());
        command.setUser(this.employeeService.load(command.getUserId()));
    }
}
```

Listing 3 – Bad performance

```
private void loadEmployeesNew(CreateInstanceCommand createInstanceCommand) {
    ArrayList<Long> ids = new ArrayList<Long>();
    //save id's
    for (TaskInstanceCommand command : createInstanceCommand.getDetailsAsSet()) {
        LOG.info("Id of user to load: " + command.getUserId());
        ids.add(command.getUserId());
    }
    //sent one big query
    List<Employee> emps = this.employeeService.findAll(ids);
    //set right user
    for (TaskInstanceCommand command : createInstanceCommand.getDetailsAsSet()) {
        for (Employee emp : emps) {
            if (emp.getId().equals(command.getUserId())) {
                command.setUser(emp);
            }
        }
    }
}
```

Listing 4 – Better performance

6.3.6.2 Example: Querying in loop

Another warning that we encounter often is: executing a database query in a loop. In the MKB project the method `findByUserName(String)` (see Listing 5) is called in a loop, while looping over a collection of usernames. For every time that this method is called a query is sent. To improve performance we send one big query instead of multiple small ones. With subselection in SQL/HQL you can retrieve all employees that have at least one of the names from the given set. We implement this solution in Listing 6.

In Listing 6 we retrieve all the Employees in one query.

6.3.6.3 Example: Field access in loop

A database hit occurs when you set a field of a persistent object, and this object is in persistent state. Looping over a collection of persistent objects and setting a field while the objects are in persistent state results in multiple queries. There are two things one can do

to increase performance on setting a field of a persistent object in a loop.

First, if the collection is an attribute of a persistent class: Get the collection out of persistent state first. But if the collections have a persistent object as a parent, you can save the whole collection in one save.

Second, if the collection is an attribute of a non-persistent class there is a problem getting the collection in persistent state again. Because you have to call save for all the objects to get them in persistent state again. For this situation there is an option to generate a query that updates all the fields that need updating in one time.

The latter one has a trade off between the performance and the readability of the code.

The optimization you can always make is persisting the object after setting its field(s) as shown in Listing 8.

```
public Employee findByUserName(String userName) {
    return getQueryBuilder().addRestriction("userName",
        userName).returnSingle();
}
```

Listing 5 – Bad performance

```
public List<Employee> findAllByUserNames(HashSet<String> userNames) {
    return getQueryBuilder().addRestriction("userNames",
        userNames).list();
}
```

Listing 6 – Better performance

```
if (supplier == null) {
    supplier = new Supplier();
    supplier.setSupplierId(supplierSchema.id);
    hibernate.persist(supplier);
    countInserted++;
}
supplier.setIpAddressRange(supplierSchema.iprange);
supplier.setName(supplierSchema.name);
supplier.setHarvestDirectory(supplierSchema.harvestDirectory);
supplier.setHarvestTransformer(supplierSchema.harvestTransformer);
supplier.setCleanBeforeHarvest(supplierSchema.cleanBeforeHarvest);
```

Listing 7 – Bad performance

```

if (supplier == null) {
    supplier = new Supplier();
    supplier.setSupplierId(supplierSchema.id);
}
supplier.setIpAddressRange(supplierSchema.iprange);
supplier.setName(supplierSchema.name);
supplier.setHarvestDirectory(supplierSchema.harvestDirectory);
supplier.setHarvestTransformer(supplierSchema.harvestTransformer);
supplier.setCleanBeforeHarvest(supplierSchema.cleanBeforeHarvest);
if (supplier == null) {
    hibernate.persist(supplier);
    countInserted++;
}

```

Listing 8 – Better performance

Consider the code sample in Listing 7 in a loop. That results in two queries for every supplier (one for persisting and one for achieve of changes in the database). If the code is changed to the code in Listing 8 it sends only one query for every supplier.

7 Concluding remarks

In this study we reported on an experiment about the possibility of finding performance issues with the use of Hibernate by means of static code analysis/SOOT. In the Hibernate documentation [1,4] it is shown that using Hibernate the wrong way can cause serious performance issues. When studying SOOT we found that valuable information can be elucidated from a Hibernate project. For this study we formulated three hypotheses.

To test them we analyzed two real life applications with our pattern detection software. We analyzed and studied the results and we noticed the following things:

- The pattern seems useful.
- In both projects most of the false positive warnings are caused by the use of the CHA Call Graph.
- Categorizing the warnings improved the usefulness of the warning document.
- Replacing source code that matches the pattern by source code that does not match the

pattern often decreases the readability.

- In category one and two the percentage of false positives is less than 75%.

We conclude that:

- It is possible to define a useful pattern.
- It is possible to statically detect a useful pattern.
- The percentage of false positives is acceptable when analyzing category one and two. But the usability of the generated warning document will improve if we reduce the percentage of false positives in category three.

8 Further research

In this thesis we have defined one pattern that can cause bad performance. This pattern is useful but the pattern and its pattern detection implementation need to be more specific to reduce the number of false positives.

The pattern can be more specific by:

- Removing the assumptions made (see section 1.1 in the introduction of this thesis) and include those variables in the pattern.
- Not only detect if a query is sent in a loop but also detect if a query is sent in a method that is called recursively.

The pattern detection implementation can be made more specific by:

- Using points-to analysis when constructing the call graph [6]. This will reduce the amount of false positives a lot. Also using a dynamic Call Graph will improve the quality of the Graph [13].
- Read the Hibernate XML-mapping files.

This pattern is not the only pattern that causes bad Hibernate performance. There are more patterns out there that can be defined and a matching algorithm can be implemented. One pattern that might be useful implementing is: A SELECT query without a WHERE clause.

9 References

- [1] Christian Bauer and Gavin King, *Java persistence with Hibernate*, Manning, 2006
- [2] Jeroen Bach, *Theory and experimental evaluation of object-relational mapping optimization techniques*, Master Thesis Software Engineering, University of Amsterdam, 2010.
- [3] Baron Schwartz, Peter Zaitsev, Vadim Tkachenko, Jeremy D. Zawodny, Arjen Lentz and Derek J. Balling, *High Performance MySQL 2nd edition*, O'Reilly, 2008
- [4] G. King, C. Bauer, M.R. Andersen, E. Bernard and S. Ebersole, *Hibernate Reference Documentation*, Manning Publications, 2009
- [5] SOOT:
<http://www.sable.mcgill.ca/soot>
- [6] Arni Einarsson and Janus Dam Nielsen, *A survivor's Guide to Java Program Analysis with Soot*, University of Aarhus (Denmark), 2008
- [7] <http://www.javana.com/jarvani/ew/org/hibernate/hibernate/3.2.2.ga/hibernate-3.2.2.ga-Javadoc.jar!/index.html>
- [8] SOOT Scene API:
<http://www.sable.mcgill.ca/soot/doc/soot/Scene.html>
- [9] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon and Phong Co, *SOOT - A Java Bytecode Optimization Framework*, September 1999
- [10] DAO-design: <http://java.sun.com/blueprints/patterns/DAO.html>
- [11] Shilong Stanley Yao, Rafael Hiriart, Irene barg, Phillip Warner and Dave Gasson, *A Case Study of Applying Object-Relational Persistence in Astronomy Data Archiving*, 2005
- [12] Pieter van Zyl, Derrick G. Kourie and Andrew Boake, *Comparing the Performance of Object Databases and ORM Tools*, University of Pretoria, 2006
- [13] Ondrej Lhoták, *Comparing Call Graphs*, University of Waterloo, June 2007
- [14] Pieter van Zyl, Derrick G. Kourie, Louis Coetzee and Andrew Boake, *The influence of optimizations on the performance of an object relational mapping tool*, University of Pretoria, 2009

10 Appendix A

In this appendix we describe the artificial performance tests and their results. We show that the pattern has performance improvement potential using these artificial performance tests.

10.1 The environment

To test the performance improvement potential of the pattern we have developed an artificial environment. The two persistent classes are shown in Listing 9 and Listing 10.

The relationship between the two classes is: Students study at one university and a university can have multiple students.

```
public class University {
    private Long id;
    private String name;
    private String city;
    private String startDate;
    private List<Student> students= new ArrayList<Student>();

    public University(){}

    public University(String name, String city, String dob){
        this.name=name;
        this.city=city;
        this.startDate=dob;
    }

    public void addStudent(Student student){
        student.setUniversity(this);
        this.students.add(student);
    }

    //...getters and setters...
}
```

Listing 9 - University.java

```
public class Student {
    private Long id;
    private String name;
    private String dob;
    private University university;

    public Student(){}

    public Student(String name, String dob){
        this.name = name;
        this.dob = dob;
    }

    //...getter and setters...
}
```

Listing 10 - Student.java

Hibernate requires a mapping file for every persistent class. The mapping files of both persistent classes are shown in Listing 11 and Listing 12.

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="domain">
    <class name="University" table="UNIVERSITY"
        schema="ArtificialperformanceTest">
        <id name="id" column="UNIVERSITY_ID">
            <generator class="increment">
                <param name="schema">ArtificialperformanceTest
            </param>
            </generator>
        </id>
        <property name="name" column="UNIVERSITY_NAME"/>
        <property name="city" column="UNIVERSITY_CITY"/>
        <property name="startDate"
            column="UNIVERSITY_STARTDATE"/>
        <list name="students" cascade="save-update">
            <key column="UNIVERSITY_ID"/>
            <index column="POSITION"/>
            <one-to-many class="Student"/>
        </list>
    </class>
</hibernate-mapping>

```

Listing 11 - University.hbm.xml

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="domain">
    <class name="Student" table="STUDENT"
        schema="ArtificialperformanceTest">
        <id name="id" column="STUDENT_ID">
            <generator class="increment">
                <param name="schema">ArtificialperformanceTest
            </param>
            </generator>
        </id>
        <property name="name" column="STUDENT_NAME"/>
        <property name="dob" column="STUDENT_DOB"/>
        <many-to-one name="university"
            column="UNIVERSITY_ID" class="University"
            not-null="true"/>
    </class>
</hibernate-mapping>

```

Listing 12 - Student.hbm.xml

For every artificial performance test we implement the functionality two times. The first implementation is the implementation with the expected bad performance and the second implementation is the implementation with the expected better performance.

10.2 Loading multiple objects from the database

Loading a (sub collection of a) collection of objects from the database is shown in Listing 13.

```

for(int i=1; i<50001; i++){
    Student st = (Student) s.get(Student.class, new Long(i));
    System.out.println(st.getName());
}

```

Listing 13 - Bad performance

This results in 50.000 queries, a query for every object we want to load from the database. The query that is sent by Hibernate (50.000 times) look like the query shown in Listing 14.

```

select
    student0_.STUDENT_ID as STUDENT1_0_0_,
    student0_.STUDENT_NAME as STUDENT2_0_0_,
    student0_.STUDENT_DOB as STUDENT3_0_0_,
    student0_.UNIVERSITY_ID as UNIVERSITY4_0_0_
from
    ArtificialperformanceTest.STUDENT student0_
where
    student0_.STUDENT_ID=?

```

Listing 14 - SELECT query

Listing 15 is a different implementation of the code in Listing 13. The query is still sent in a loop but the loop is executed only 200 times (instead of 50.000 times).

```

ArrayList<String> strs = new ArrayList<String>();
for(int i=1; i<50001; i+=batchsize){
    String str = "";
    for(int j=i; j<(i+batchsize)&&j<50001; j++){
        str+=j+", ";
    }
    str=str.substring(0, str.length()-1);
    strs.add(str);
}
for(String str : strs){
    List<Student> students = s.createQuery("from Student s where
s.id in (" +str+")").list();
    for(Student student : students){
        System.out.println(student.getName());
    }
}

```

Listing 15 - Better performance

Batchsize is in this case 250, but this can be different in every situation. We found that 250 is an optimal number of objects in this situation. If the objects have connecting id's it is much faster to query for "WHERE STUDENT_ID=? AND STUDENT_ID>?", but keep the size of your local memory in mind.

In Listing 16 we show the query that is sent 200 times in the code in Listing 15.

```

select
    student0_.STUDENT_ID as STUDENT1_0_,
    student0_.STUDENT_NAME as STUDENT2_0_,
    student0_.STUDENT_DOB as STUDENT3_0_,
    student0_.UNIVERSITY_ID as UNIVERSITY4_0_
from
    ArtificialperformanceTest.STUDENT student0_
where
    student0_.STUDENT_ID in (1 , 2 , 3 , 4 , ... , 250)

```

Listing 16 - SELECT query

10.2.1 Performance

Action: Loading 50.000 students.

In the first case (Listing 13) it takes 26,5 seconds (and 50.000 queries) to load all the students and print all their names.

In the second case (Listing 15) it takes 10,3 seconds (and 200 queries) to load all the students and print all their names.

10.3 Querying multiple objects from the database

Querying for multiple students from the database is shown in Listing 17.

```
for(int i=1; i<50001; i++){
    Student student = (Student) s.createQuery("from Student s where
s.id="+i).uniqueResult();
    System.out.println(student.getId());
}
```

Listing 17 - Bad performance

The implementation in Listing 17 results in 50.000 queries, one query for every object we retrieve. The query sent by hibernate (50.000 times) looks like the query shown in Listing 18.

```
select
    student0_.STUDENT_ID as STUDENT1_0_,
    student0_.STUDENT_NAME as STUDENT2_0_,
    student0_.STUDENT_DOB as STUDENT3_0_,
    student0_.UNIVERSITY_ID as UNIVERSITY4_0_
from
    ArtificialperformanceTest.STUDENT student0_
where
    student0_.STUDENT_ID=4
```

Listing 18 - SELECT query

Listing 19 is a different implementation of the code in Listing 17. The query is still sent in a loop but the loop is executed only 200 times (instead of 50.000 times).

```
ArrayList<String> strs = new ArrayList<String>();
for(int i=1; i<50001; i+=batchsize){
    String str = "";
    for(int j=i; j<(i+batchsize)&& j<50001; j++){
        str+=j+", ";
    }
    str=str.substring(0, str.length()-1);
    strs.add(str);
}
for(String str : strs){
    List<Student> students = s.createQuery("from Student s where
s.id in (" +str+")").list();
    for(Student student : students){
        System.out.println(student.getName());
    }
}
```

Listing 19 - Better performance

Batchsize is in this case 250, but this can be different in every situation. We found that 250 is an optimal number of objects in this situation. If the objects have connecting id's

it is much faster to query for “WHERE STUDENT_ID<? AND STUDENT_ID>?”, but keep the size of your local memory in mind.

In Listing 20 we show the query that is sent 200 times in the code in Listing 19.

```
select
    student0_.STUDENT_ID as STUDENT1_0_,
    student0_.STUDENT_NAME as STUDENT2_0_,
    student0_.STUDENT_DOB as STUDENT3_0_,
    student0_.UNIVERSITY_ID as UNIVERSITY4_0_
from
    ArtificialperformanceTest.STUDENT student0_
where
    student0_.STUDENT_ID in (1 , 2 , 3 , 4 , ... , 250)
```

Listing 20 – SELECT query

10.3.1 Performance

Action: Querying 50.000 students.

In the first case (Listing 17) it takes 362,5 seconds (and 50.000 queries) to query all the students and print their id's.

In the second case (Listing 19) it takes 9,6 seconds (and 200 queries) to query all the students and print their id's.

10.4 Setting multiple persistent fields

Setting all the student names from students who attend at universities with id between 1 and 500 to “Jasper Alblas” is shown in Listing 21.

```
for(int i=1; i<500; i++){
    University u = (University) s.get(University.class, new Long(i));
    if(u==null) continue;
    List<Student> students = u.getStudents();
    for(Student student : students){
        student.setName("Jasper Alblas");
    }
}
```

Listing 21 – Bad performance

The implementation in Listing 21 results in 50.500 queries, one query for every University and one for every Student we retrieve. The queries sent by hibernate for each Student looks like the query shown in Listing 22.

```
update
    ArtificialperformanceTest.STUDENT
set
    STUDENT_NAME=?,
    STUDENT_DOB=?,
    UNIVERSITY_ID=?
where
    STUDENT_ID=?
```

Listing 22 – UPDATE query

Listing 23 is a different implementation of the code in Listing 21. A query is still sent in a loop but the loop is executed only 500 times (instead of 50.500 times).

```

for(int i=1; i<500; i++){
    Query q = s.createQuery("update Student s set s.name='Jasper
    Alblas' where s.university="+i);
    q.executeUpdate();
}

```

Listing 23 - Better performance

10.4.1 Performance

Action: Setting 50000 names of students

In the first case (Listing 21) it takes 16,7 seconds (and 50.500 queries) to set all the 50.000 names.

In the second case (Listing 23) it takes 3,6 seconds (and 500 queries) to set all the 50000 names.

10.5 Deleting persistent objects from a collection

Deleting all the students from a particular university is shown in Listing 24.

```

University p2 = (University) s.load(University.class, new Long(4));
List<Student> students = p2.getStudents();
int size = students.size(); //initialize collection
s.evict(p2);

for(int i=0; i<students.size();i++){
    Student d = students.get(i);
    if(d!=null) s.delete(d);
}

```

Listing 24 - Bad Performance

The implementation in Listing 24 results in 50.000 queries, one query for every student that is deleted. The query sent by hibernate looks like the query shown in Listing 25.

```

delete
from      ArtificialperformanceTest.STUDENT
where     STUDENT_ID=?

```

Listing 25 - DELETE query

Listing 26 is a different implementation of the code in Listing 24. A query is still sent in a loop but the loop is executed only 200 times (instead of 50.000 times).

```

University p2 = (University) s.load(University.class, new Long(4));
List<Student> students = p2.getStudents();
int size = students.size(); //initialize collection
s.evict(p2);

ArrayList<String> str = new ArrayList<String>();
for(int i=1; i< students.size(); i+=batchsize){
    String str = "";
    for(int j=i; j<(i+batchsize)&& j<students.size(); j++){
        str+=j+", ";
    }
    str=str.substring(0, str.length()-1);
    str.add(str);
}
for(String str : str){
    Query q = s.createQuery("delete Student s where s.id in
    (" +str+ ")");
    q.executeUpdate();
}

```

Listing 26 - Better performance

In Listing 27 we show the query that is sent 200 times in the code in Listing 26.

```

delete
from
    ArtificialperformanceTest.STUDENT
where
    STUDENT_ID in (
        49501 , 49502 , ... , 49750
    )

```

Listing 27 - DELETE query

10.5.1 Performance

Action: Deleting 50.000 persistent objects from the database.

In the first case (Listing 24) it takes 19,6 seconds (and 50.000 queries) to delete 50.000 students from the database.

In the second case (Listing 26) it takes 9,7 seconds (and 200 queries) to delete 50.000 students from the database.

11 Appendix B

Category distribution:

MKB:

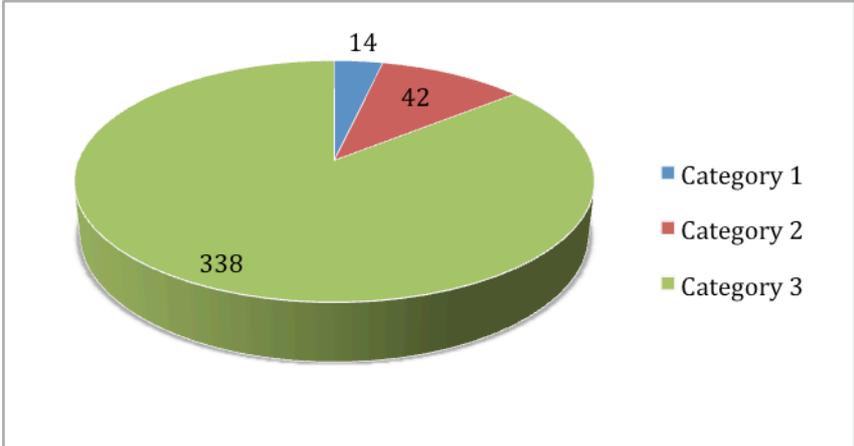


Figure 3 - MKB category distribution

SMB:

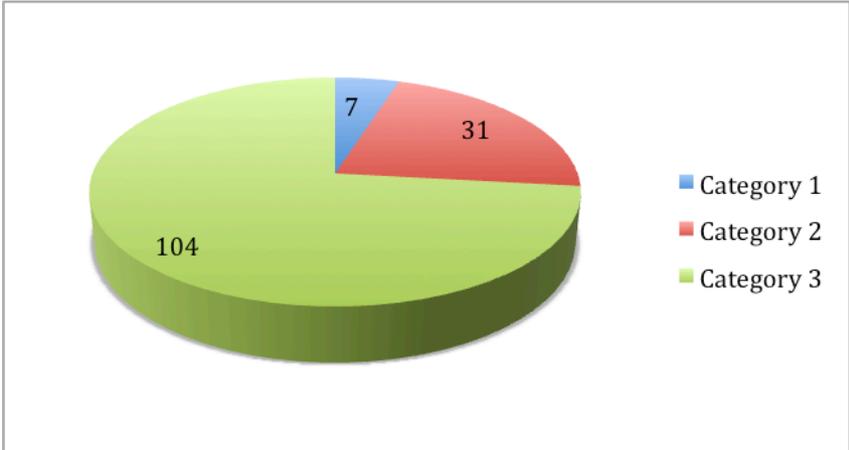


Figure 4 - SMB category distribution

Category one group distribution:

MKB:

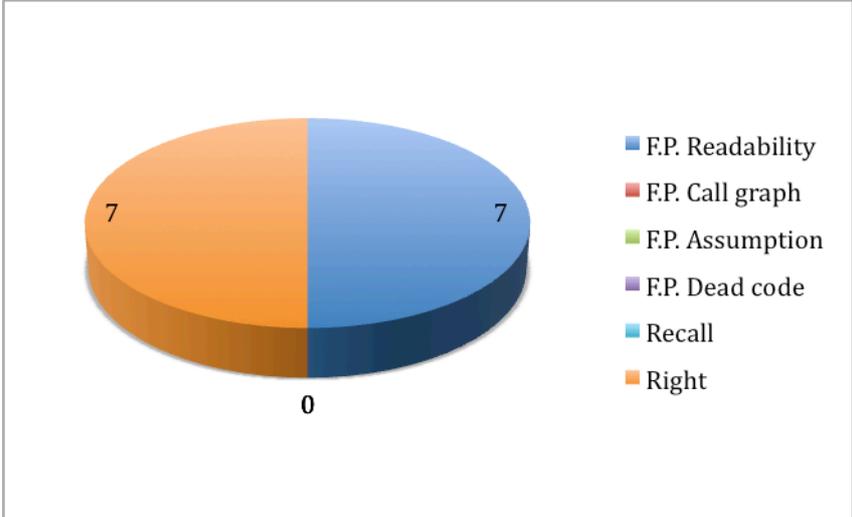


Figure 5 - MKB category one group distribution

SMB:

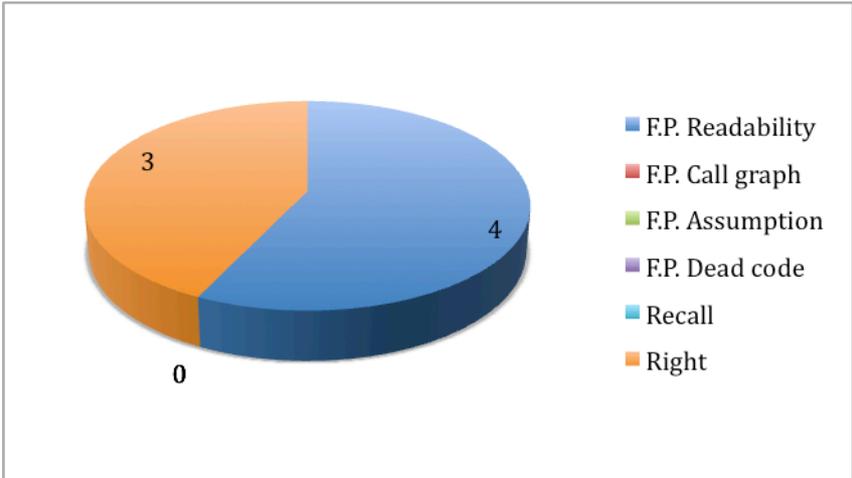


Figure 6 - SMB category one group distribution

Category two group distribution:

MKB:

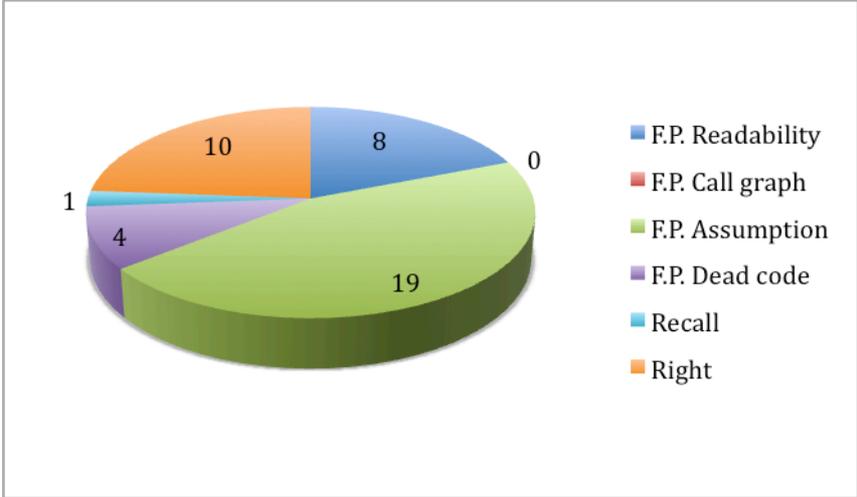


Figure 7 - MKB category two group distribution

SMB:

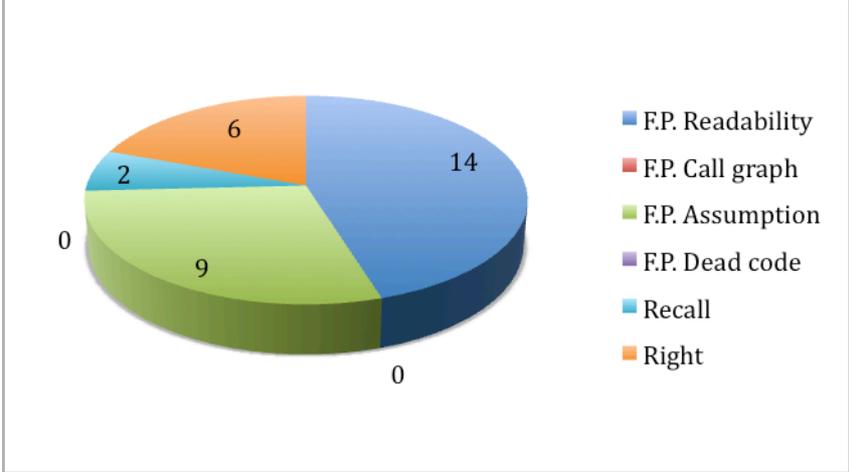


Figure 8 - SMB category two group distribution

Category three group distribution:

MKB:

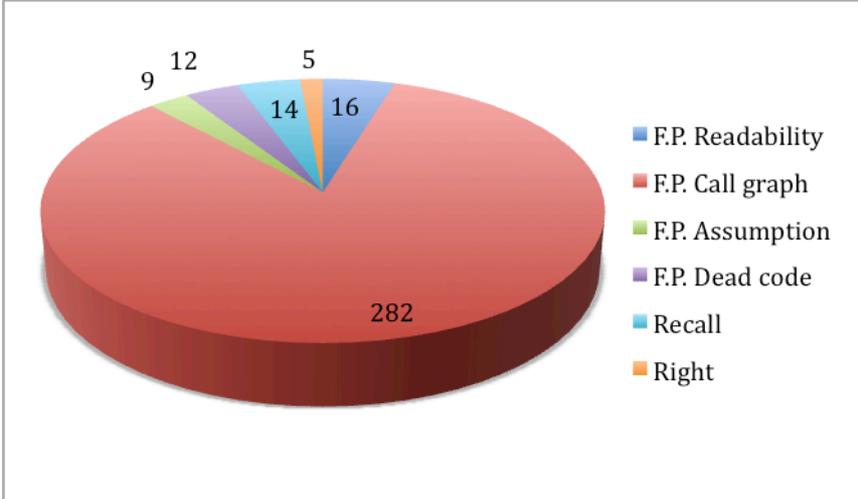


Figure 9 - MKB category three group distribution

SMB:

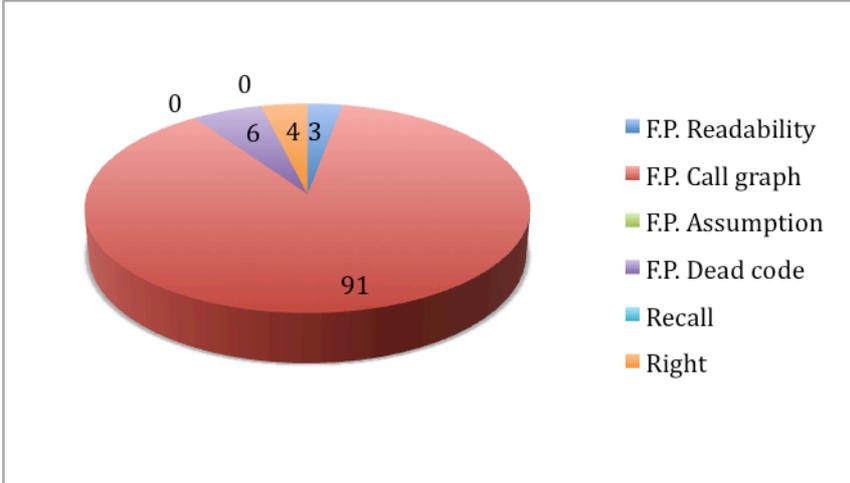


Figure 10 - SMB category three group distribution