Detection of the Abstract Factory Pattern: an experimental study

Koen Hanselman

Master thesis at the University of Amsterdam
Supervised by dr. Jurgen Vinju, CWI
January 2013

TABLE OF CONTENTS

PREFACE

I would like to thank thesis supervisor Jurgen Vinju for the extensive support and the opportunity to perform my thesis research at CWI. I would also like to thank Dennis van Leeuwen, Luuk Stevens, and Jouke Stoel for (unknowingly) offering their insights and experiences on the subject of identifier naming. Last, I would like to thank my family and friends for their support and understanding during the period I worked on my thesis.

1 INTRODUCTION

1.1 Software maintenance

After extensive past research on the subject of software costs, it has become publicly known by now that software maintenance has a major share in the total lifecycle costs of a software product. Numbers taken from research indicate that software maintenance costs represent 60-80% of the total costs in a software product lifecycle. [1] In their article, Canfora and Cimitile write:

> "Since 1972, software maintenance was characterized as an 'iceberg' to highlight the enormous mass of potential problems and costs that lie under the surface. … these surveys also report that maintenance costs are largely due to enhancements (often 75-80%), rather than corrections."
>
> *G. Canfora, A. Cimitile* [1]

There are, in addition, plenty of other articles about the huge costs involved in software maintenance. [2–4]

One of the major reasons for this large amount of time and effort spent on software maintenance, is a lack of software understanding and comprehension. [1], [2] Since maintainers of a product are often not the same person(s) as the initial developer of the system, documentation is needed to gain proper understanding of the system being modified. In practice this proper documentation is unfortunately however often not available, as (among others) research by G. Canfora and A. Cimitile shows [1].

The lack of proper documentation naturally affects an individual's understanding and comprehension in a negative way even further. It is therefore highly interesting to see if a solution can be found to solve the problem of missing documentation and complex software architectures and structures in software products.

1.2 Design patterns

To deal with the problem of software architectures getting more and more complex as a product grows, design patterns are often introduced. Design patterns are common and proven solutions to often occurring architectural "problems". As design patterns have become better known (mostly by the work of the "Gang of Four" [5]), they now also function as a common vocabulary among developers: they bring recognition, familiarization, and understanding to at first seemingly

complex architectures. As understandability of software increases, maintenance effort and time logically decreases.

Research however also shows that the use of design patterns in software is only useful for a better understanding of the software product, if the maintainer is explicitly aware of where and which design patterns are used. If proper documentation of the used design patterns is lacking, design patterns actually make the software (look) more complex and therefore increase the maintenance effort and time, instead of decreasing it. [6], [7]

Section 1.1 points out that documentation is unfortunately regularly missing. To solve this problem of a lack of documentation about the used design patterns, and therefore the unusable and hidden advantages of the use of design patterns, we aim to develop a tool that can detect design patterns in software. Because we aim to detect incorrect naming of identifiers, we will not use identifier information for the detection. We will focus on the detection of the Abstract Factory Pattern. Later this tool can be extended to support the detection of other design patterns. With such a tool a developer can still gain a certain level of understanding about a software architecture, even when documentation is missing.

By answering several research questions, we hope to be able to provide a good view on the current performance and possibilities of design pattern detection, and we also hope to be able to identify the aspects that should be focused on in the search for improvement of design pattern detection.

1.3 Research questions

In order to bring more clarity to the problems and difficulties stated in the previous sections, this thesis will aim to find an answer to the following main research question:

> *"What is the best way to perform design pattern detection on the code of a software product without using identifiers information and how well does this work?"*

To find an answer to this main research question, we will first try to answer some smaller research questions:

- What are the results of currently known design pattern detection methods?
- Considering the current design pattern detection methods and their results/performance, how much room for improvement can be found and what concepts can be best used to develop a detection tool that meets our requirements?

1.4 Previous master theses

Apart from the general use of the outcome of this thesis, the results will also provide an important addition to the master theses done by fellow students, which have done research after the naming of components. Design patterns often form exceptions to standard naming, so in this way the research in this thesis could contribute to former research about naming.

Luuk Stevens has done research on the automatic analysis of the consistency and preciseness of class names: do class names represent their nature in a proper way, and is there consistency in the naming of classes? [8] Stevens' motivation for doing his thesis on this subject shows some comparison to the motivation behind this thesis: consistent and precise class naming are important for program comprehension.

Dennis van Leeuwen has done his thesis on "comprehensible method names: focusing on the nouns". [9] Van Leeuwen was also motivated by making software more comprehensible (and decreasing maintenance efforts).

The thesis of Jouke Stoel focuses on exploring the detection of method naming anomalies. [10] He researched if it is possible to automatically determine if a method name describes well enough what the method actually does.

Because the use of design patterns causes exceptions in class-, method-, and other identifier naming, it is very useful to be able to detect these exceptions, for the use of correct naming.

1.5 Scope

For the research of this thesis, a tool will be developed using the domain specific language Rascal [11] (see the section "Background" for more info). Furthermore, the focus will lie in the detection of design patterns in Java code. Java is in this case most suitable because the theses of Stevens, van Leeuwen, and Stoel also scoped on Java projects, so it would be easy to match and combine outcomes. Also, Java provides good support for object oriented programming and the use of design patterns, and Java is also used in previous research in the subject of design pattern detection which we might want to reproduce.

We will exclude any naming information from our research on the detection of design patterns, for several reasons. First, our research will be more useful to the other theses because the other authors have already focused on different aspects of naming, and the problem is actually that these design patterns are exceptions to their research results.

Furthermore, the problem of incomprehensibility of code due to the use of design patterns is actually the worst when documentation (or correct naming) is lacking [7]. Therefore it is actually the most interesting to be able to retrieve design pattern information when proper documentation or naming is not in place.

Our tool will focus on the detection of the Abstract Factory Pattern. We focus on this design pattern because it is an often used design pattern. This will not only make the results more useful but will also enable us to test our solution properly on existing projects. The Abstract Factory Pattern is also a design pattern which we have recent practical experience with, which aids the quality of our research.

1.6 Structure of this thesis

The remainder of this thesis is structured as follows:

Section 2 will provide further background information on the subjects that will be covered in this thesis. This mainly includes information about design patterns and the meta programming language Rascal [11].

Section 3 covers the subject of design pattern detection, including previous research conducted on this subject, but also other aspects that are important for doing proper research on the subject of design pattern detection.

Section 4 will describe the development of our design pattern detection tool.

Section 5 will ultimately take a look at the results that were gathered in section 4, and will analyze and discuss these results.

Section 6 will highlight what can be concluded from this research and describe subjects that we think are interesting for doing further research on.

2 BACKGROUND

2.1 Design pattern awareness and program maintenance

Several experiments were conducted by Lutz Prechelt et al. to find out the importance of proper documentation of implemented design patterns in program code ([6], [7]). More concretely they researched if explicit documentation of implemented design patterns (using source code comments) helps a software maintainer, opposed to general source code comments that do not describe any implemented design patterns explicitly. (Assumed is that the maintainer knows what design patterns are and how they are used.)

Prechelt et al. were motivated by theories in the literature about software comprehension. These theories describe that a programmer will switch to a top-down approach of understanding, which allows for generating and validating hypotheses much more quickly, if hints (documentation) to familiar kinds of structures within the code are provided.

The experiment by Lutz Prechelt et al. was conducted on two groups of (randomly divided) people, in which both groups had to provide appropriate changes for change requests that they were presented with. One group received explicit documentation about design patterns implemented in the program that was used in the experiment, while the other group did not receive this documentation. The documentation about the design patterns was given as Pattern Comment Lines (PCL). The difference in performance of the two groups was investigated by measuring completion time, grading answers, and counting correct solutions. The experiments were conducted on a total of 96 subjects.

Results showed that the implementation of design patterns actually makes the software more complex. The experiment subjects that were presented with the program including PCL, completed the requested changes zero to 43 percent faster than the experiment subjects that worked on the program without PCL, and also provided better solutions in general.

This indicates that the use of design patterns can provide better and faster understandability of program code, but only if the maintainer is explicitly aware of the design patterns used in the code (which is not always the case). The subjects that were not aware of the used design patterns, did not seem to have any profit from the implemented design patterns. In fact, in that case, design patterns can even contribute to extra lines of code and a structure that seems more complex than “straightforward” code without the use of design patterns. This research thus shows that information about any design patterns that are used, is very important in software maintenance.

Dennis van Leeuwen, Luuk Stevens, and Jouke Stoel also point out in their thesis research that improper naming accounts for a substantial part of maintenance: Lawrie et al. showed the impact of identifier names on program comprehension [12], and Lawrie et al. and Deissenboeck and Pizka showed that identifier names should be meaningful and consistent ([13], [14]). Design pattern naming of course forms an important part of consistent naming, since consistent naming eases the recognition of design pattern instances (and thereby increases understandability), but also because it forms an often occurring and important exception to regular naming.

2.2 Rascal

For the development of the detection tool we will use Rascal [11]. Rascal is a domain specific language for source code analysis and manipulation, also known as meta-programming. It is currently being developed and tested at CWI.

The development of our tool takes place in Eclipse, for which a Rascal plugin exists. As is written on the Rascal website, Rascal is focused on code analysis and manipulation, and therefore serves as a good base for our research.

3 DETECTING DESIGN PATTERNS

3.1 Current research and methods

A lot of research has been done already on the detection of design patterns. This has led to several methods to perform design pattern detection, which of course differ in methodology, generality, speed, and accuracy. In this section we will highlight four methods:

- Manual blueprint coding
- Software metrics
- Information filtering
- Similarity scoring

We highlight these methods because they have promising results. Additionally, they all use different techniques/concepts to perform design pattern detection which all have their advantages and disadvantages. This will broaden our own vision and ideas on design pattern detection.

3.1.1 Manual blueprint coding

Heuzeroth et al. [15] present a rather straightforward way to do design pattern detection. Their analysis basically consists of 2 phases: static analysis and dynamic analysis. Both of these phases focus on structure and do not use any naming information available (method and class names may be compared to each other, but not to external constants).

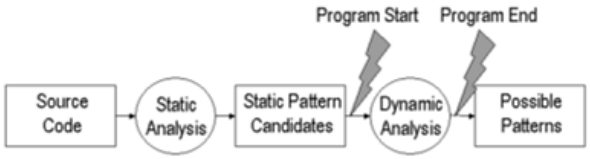


Figure 1 Static and dynamic analysis, taken from [15]

The static analysis reads out the program and constructs an AST. From there, it computes a set of tuples (the authors call them "candidates") with the appropriate AST structure. This computation is done by iterating over all program classes and their methods, and mapping a supposed design pattern structure, which is programmed by the authors as a kind of structural blueprint, to the AST nodes. The list of candidates that results from the static analysis, is then further refined by the dynamic analysis.

Where the static analysis performs an analysis in purely the code of the program, the dynamic analysis monitors the program during execution, and checks whether the interaction among them satisfies the rules of the design pattern "blueprint".

Evaluation of the results shows that this method has very little false positives (or zero in many cases), and in that way can be considered as good. Because this detection method is however dependent on execution of the program during the dynamic analysis phase, it can be hard to perform correct detection: it requires the execution of every line of code in a program. If complete execution of the program does not find place, the number of false negatives can be large [15]. Another downside of this methodology is that for every design pattern, a design pattern blueprint has to be coded. First, this can be a lot of work, and second, when done manually, it can be hard to validate. This method also can only find patterns that exactly match the given blueprint. Any patterns that have even the smallest modifications, will not be detected by this method.

3.1.2 Software metrics

Kontogiannis has researched a different, but somewhat comparable method for detecting design patterns, that is based on the idea of code clone detection, and makes use of software metrics [16]. According to Kontogiannis, this method is fast due to the use of signatures based on software metrics. Opposed to the method suggested by Heuzeroth et al. [15], this method also offers more flexibility: by calculating the Euclidean distance between the original signatures and any occurrences found in a program, this method can also find design pattern occurrences that have been modified in comparison to the definitions as they are officially stated by the Gang of Four [5] (although Kontogiannis in this case only researches matches with a Euclidean distance of zero). The drawback of this method is that the amount of false positives increases when the algorithm is adjusted to decrease the amount of false negatives.

Kontogiannis generates an attributed AST from a program, in which the attributes of each node are a tuple of 5 software metrics (that serve as a signature) computed for that node.

3.1.3 Information filtering

Pettersson and Löwe researched the detection of design patterns by graphs [17]. The concept of their method is to filter out any information that is not needed, after which the relations between classes and methods can be presented as a graph. From the original design patterns also a graph is constructed/generated. This graph is then matched to any graphs deducted from the program. According to the authors, in case of planar graphs, "this isomorphism problem (the matching) is always solvable in time linear in the number of graph nodes and found pattern instances". These planar

graphs were however only generated in 16 cases (from a total of 34 programs).

The authors use Crocopat [18] to filter out any unneeded information.

The explanation the authors give in their paper about their method in general and about their filtering methods, we however found hard to comprehend. This therefore makes it hard to reproduce any experiments done in this specific research. The paper however does provide useful insights about filtering and a new concept/idea on doing design pattern detection: performing detection by first removing all data that is not useful, after which the detection algorithm can perform much faster and room for false positives/negatives is already reduced up front.

3.1.4 Similarity scoring

Tsantalis et al. researched design pattern detection based on similarity scoring [19]. Their method is similar to the method by Pettersson and Löwe [17], in the sense that it also works with graphs.

By generating several graphs for the same piece of code (based on different code characteristics) and then combining these graphs into one graph, the authors create a signature for a piece of program code.

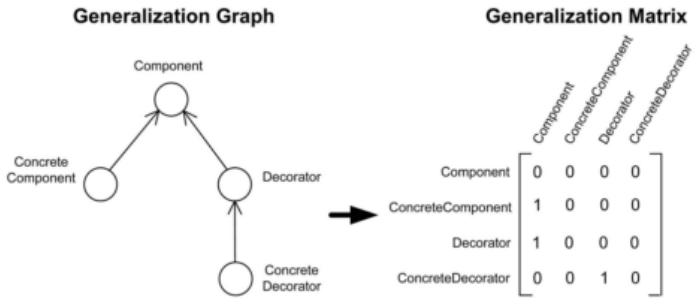


Figure 2 Graph generation from program architecture, taken from [18]

The signatures presented by these graphs are then compared to the signatures for the design pattern definitions, and the similarity algorithm calculates a similarity scores between the vertices of the graphs.

The authors present a similarity algorithm that combines generalization and association relation data into a similarity score. This enables better comparison between signatures than just comparing graphs in a more straightforward way. Their method is further described in a separate paper [20].

The results show that, because of the similarity scoring algorithm and the representation in graphs, this detection method is able to also detect pattern instances that are modified from the official structures of the patterns as described by the Gang of Four [5]. The results table that the authors present shows that the detection method is accurate. The results were measured by comparing the detected design pattern instances to the official documentation of the open source systems that they used for the experiments.

The authors have also put up a website which describes their method and shows the research results about the found pattern instances in more detail. Moreover, the code of their detection tool is downloadable from the website, and they refer to another paper that the authors wrote in which they further explain the application of graph theory to object oriented software engineering [20].

3.2 Comparison of detection methods

Section 3.1 describes four methods for design pattern detection. The first method by Heuzeroth et al. requires manual coding of "blueprints" for the design patterns that have to be detected. It can be hard to correctly code these blueprints. Moreover, this method can only detect exact matches with the blueprint. Modified versions (even slightly modified) of design patterns will not be detected by this method. Still the paper presents some good results, and the coding of blueprints allows for easy extension and perfection of the detection.

The second method described in section 3.1 is by Kontogiannis. His method uses software metrics to create a unique signature for parts of a program, which is then compared to the signatures of design pattern definitions. This method is able to detect modified versions of design patterns (due to the use of Euclidean distance), but the amount of false positives increases when the algorithm is adjusted to decrease the amount of false negatives. According to the paper, their detection is fast, although actual numbers are not provided.

The third method, by Pettersson and Löwe, uses information filtering to filter out all information that is not needed for the design pattern detection. We found their paper hard to comprehend at some points, but the results from this method show to be average. For approximately half of the cases in their experiment the design pattern detection could be performed in linear time, but for more complex design patterns this is not the case. Speed performance can therefore also be considered average.

Of the methods described in the previous section, the results of the last method, by Tsantalis et al., seems the most promising. Opposed to the other methods described in section 3.1, it seems to offer the most accurate and reliable detection of design patterns, including modified versions of design patterns. The speed of this method also seems to be doable, although it is hard to make a comparison

between the speed of all methods covered in this section. This research method is however also the most complicated one, and is therefore hard to extend and perfect.

Overall the method by Heuzeroth et al. seems to offer the most flexibility due to its simple nature, although this method does not perform well at the detection of modified design patterns. The method by Tsantalis et al. seems to provide the best detection results (although also here it is hard to make a straight comparison between the different methods), but its implementation is complicated and thus hard to extend/modify. The other two methods by Kontoginannis and Pettersson/Löwe perform average on most aspects and therefore do not excel at any point.

Depending on the goals and interests, either the method by Heuzeroth et al. or the method by Tsantalis et al. will probably be the best choice.

The table below shows a comparison of the different methods' strong and weak characteristics:

	Blueprint coding	**Software metrics**	**Information filtering**	**Similarity scoring**
Precision	+/-	+/-	+/-	++
Recall	-	+/-	+/-	+
Flexibility	++	+	+	--
Ease of use / maintenance / development	++	+/-	+	--
Conciseness / completeness	-	+/-	+/-	++

All in all, none of the methods described here meet our requirements. We are looking for a method with good results (a high recall, accuracy is less important) which is easy to maintain and extend. The blueprint coding method by Heuzeroth et al. offers flexibility and is easy to maintain and extend, but it cannot detect modified instances of design patterns and therefore has a low recall. The similarity scoring method by Tsantalis et al. is the opposite: a high recall with good results, but very hard to maintain and extend because of its algorithm and conciseness. We have therefore chosen to develop our own method, which we will describe in section 4.

4 DEVELOPMENT OF THE DETECTION TOOL

4.1 Goals, requirements and decisions

Our goal in the creation of our design pattern detection tool was to create a tool that performed adequately, but was also easy to extend and perfect and did not use identifier naming information for its detection (although naming information could be used to compare names).

We put together the following list of requirements:

1. The tool should be easy to extend, so that it was easy to also make our tool detect other design patterns, if we wanted to in a later stadium.
2. The detection algorithms for the detection of each design pattern should also be easy to change and perfect over time.
3. Since design patterns are hardly ever used in their strict form according to the official definitions by the GoF, our tool should also be able to detect slightly modified instances of design patterns.
4. The detection tool results should have little false negatives (of course also depending on the level of modification of the concerning instance of the design pattern). If that comes at the expense of an increase in false positives, so be it, but of course the amount of false positives should be kept to usable limits, in order for our detection results to still be usable. We aim to have zero false negatives, and allow up to 50% false positives in order to be able to satisfy our zero-false negatives goal.

Looking back at the previous research we covered in section 3.1, the concept of using blueprints for the design pattern detection seems very suitable for the requirements we have: we can easily add new blueprints in case we want to add the detection of other patterns, and also the blueprints can be edited and modified separately in case we want to change or perfect our detection for a certain design pattern.

Opposed to the complicated similarity algorithm as used by for example Tsantalis et al., blueprints show a direct relation to the design patterns to be detected, which also adds up to the ease of modification.

The blueprint method as described by Heuzeroth et al. does however not support the detection of modified occurrences of design patterns. For this the Euclidean distance methodology idea that Kontogiannis uses in his research can bring a solution. The blueprints that we will use for our patterns detection can in fact also be seen as a list of characteristics (in which these characteristics might be related to each other). This enables us to compute “satisfaction scores” for certain

characteristics, representing the extent to which they match the originally required characteristics. By using adjustable thresholds for these satisfaction scores, we can then adjust to which extent we want to detect modified instances of the design pattern.

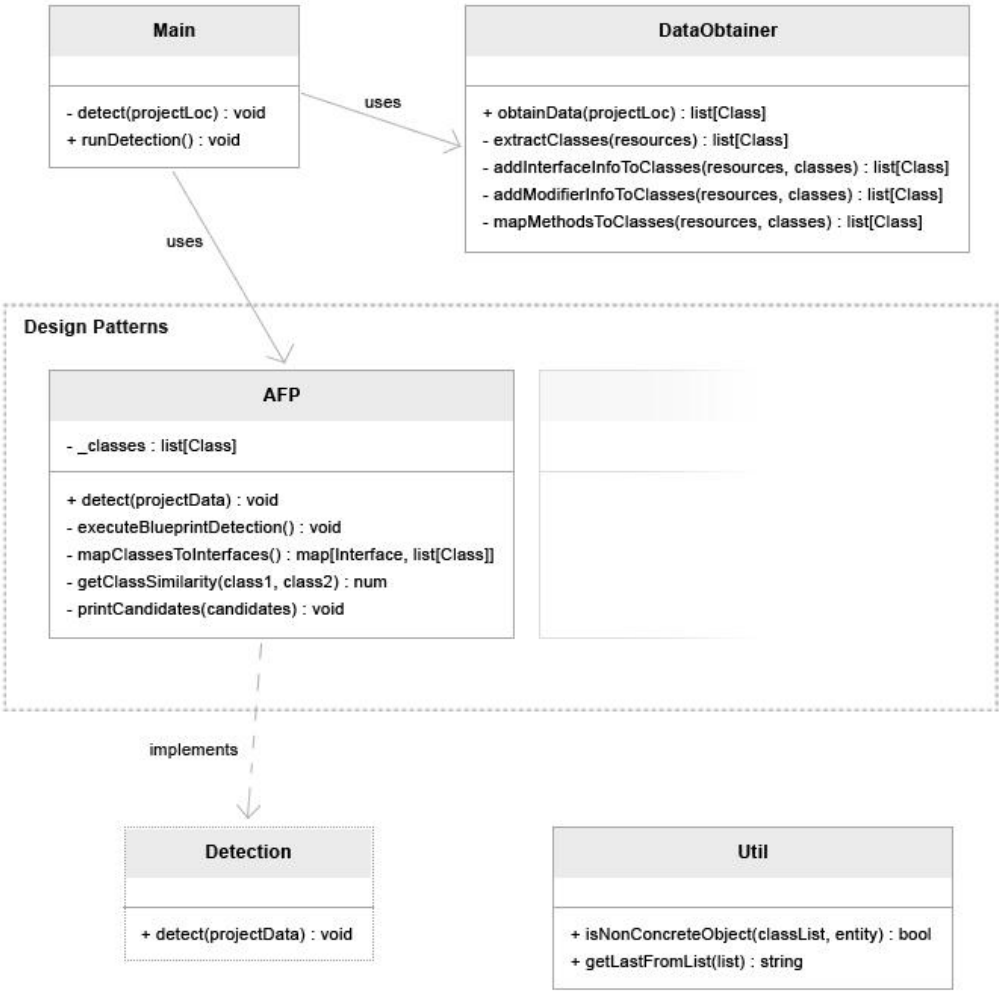
Through this mechanism we can also satisfy our fourth requirement. By setting our threshold values to 1 (values between 0 and 1 are generated) we can set the algorithm to strictly match our blueprint structure, and by lowering several threshold values of choice, we can allow more modified occurrences of design patterns to show up in our results.

To summarize our decisions:

- We use the blueprint coding method to detect design patterns because it is straightforward and easy to maintain and extend.
- We use threshold values to match specific characteristics of the design pattern up to a degree of choice, so that we can choose to allow for less or more false positives/negatives in our results. This enables us to tune our results to our needs in different cases.

4.2 Architecture

As a result of the requirements and decisions we described in section 4.1, we have designed an architecture for our tool, which can be seen below:



Model 1 The architecture for our detection tool

The Main module calls/uses the DataObtainer module to obtain all necessary project data, which is returned as a list of Class objects. The Main module then calls de detect() methods for each of the design patterns to be detected. Each design pattern has its own module which accepts the projectdata as a parameter (this is the list of Classobjects which was obtained), and eventually prints out its detected design pattern candidates. Each design pattern detection module implements its own result printing function, because the desired output format may vary. The Util module contains several general functions which can be used by all other modules.

Model 1 shows the structure our intention of the architecture, however the practical implementation does not have the “Detection” interface as shown in the model, because Rascal does not support interfaces, but modules instead. The responsibility of adhering the Detection interface in the creation of the design pattern detection modules will therefore shift to the developers.

4.3 Intermediate format

For more ease of use and better understandability of our algorithm we created an intermediate format to represent our data. We set up three datatypes to represent our class, interface, and method objects.

As explained in section 4.2, the tool uses a DataObtainer module to obtain all necessary project data. It uses Rascal's JDT functions to extract information about classes, interfaces, methods, and more, and maps this information to custom Rascal datatypes (objects). The objects that are created in this step by the dataobtainer, are then used to do the analysis necessary to detect any design patterns.

The datatypes that are used, are defined as follows:

```
1  data Class = Class (Entity entityObject, str name,
   list[Interface] interfaces, list[str] package, list[str]
   classes, list[Method] methods, list[str] modifiers, loc
   location);

2  data Method = Method (Entity entityObject, str name, str
   className, list[str] classes, Entity returnType, list[Id]
   returnTypeIds, loc location);

3  data Interface = Interface (Entity entityObject, str name,
   list[Id] ids);
```

There are three datatypes currently used: Class, Method, and Interface. Since we have only focused on the detection of the Abstract Factory Pattern, we do only obtain the data that is necessary for our detection. Additional data might be needed to perform detection of other design patterns, but this information can easily be added to the DataObtainer later on.

The extraction of data from the test projects takes place in basically five steps:

1. First the project is loaded into the tool. The project location is pointed out and the project files and directories are loaded.

Then there are four methods in place to extract information and create objects from this information:

2. First all class information is extracted and the Class objects are created.
3. Second, modifier info is added to the Class objects in order to be able to detect abstract classes.
4. Third, all interface information is extracted, Interface objects are created, and these objects get matched with their corresponding implementing Class objects.

5. Fourth, all method information is extracted (and Method objects are created) and mapped to their corresponding Class objects.

The extraction of information from the source projects is done by a visitor which uses pattern matching to detect the necessary elements to create our custom objects. All the code which we used to perform this data extraction, is attached in Appendix A.

4.4 Abstract Factory Pattern detection algorithm

After all data is obtained and the necessary objects are created, we call the detect() method of our Abstract Factory Pattern (AFP) detection module, to perform the detection of instances of the AFP.

To create a method that can successfully detect instances of the AFP, we first analyzed the structure of the AFP. For this we used the Gang of Four book [5] as a reference, since this is generally considered the standard in design patterns.

According to the Gang of Four book the definition of the AFP is as follows:

> Abstract Factory provides an interface for creating generic product objects. It removes dependencies on concrete product classes from clients that create product objects.
>
> *Definition of the Abstract Factory Pattern according to the GoF book* [5].

The following diagram shows the structure of an instance of the AFP:

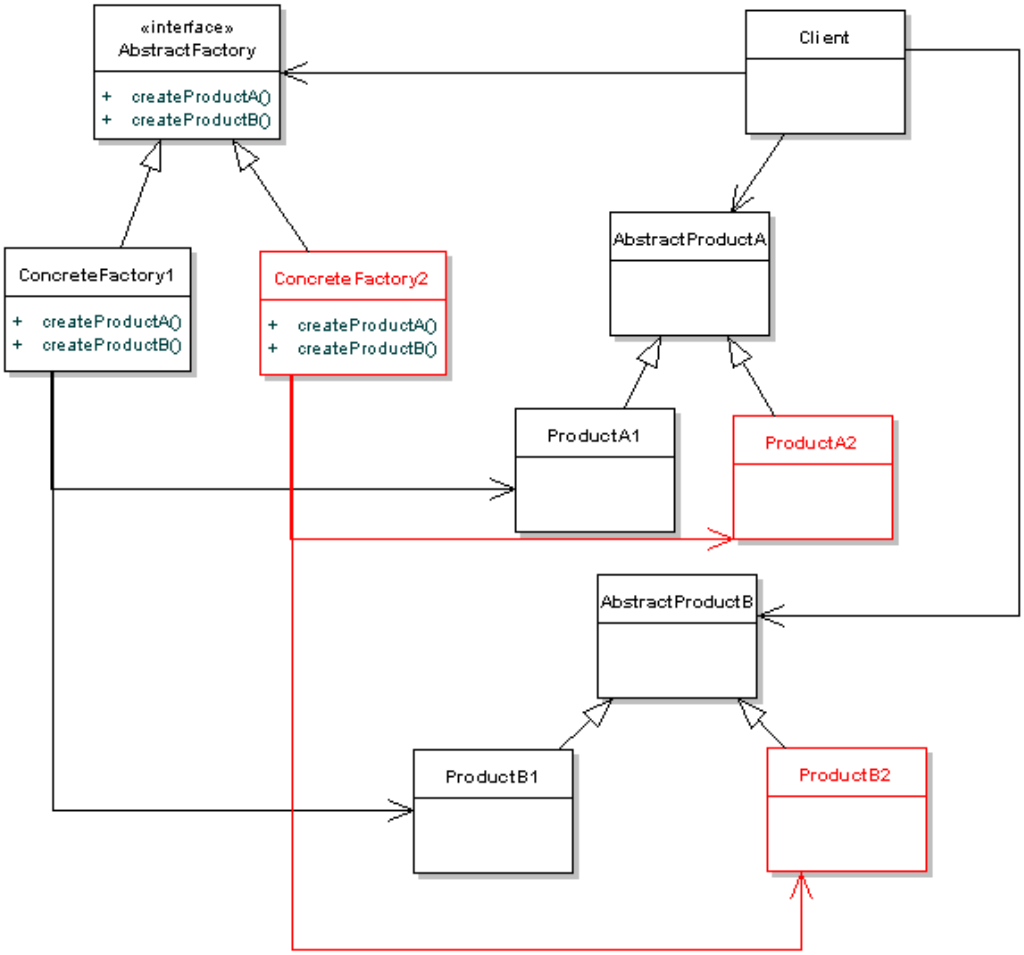


Figure 3 The architecture of an instance of the Abstract Factory Pattern

We identified several characteristics to start with:

- At the "top" of the structure there is an interface (the abstract factory)
- There are preferably two or more classes that implement this interface (the concrete factories).
- Each of these classes have methods with corresponding names and corresponding return types.
- The return types of all methods in the concrete factories must be of a non-concrete type (interface or abstract class). The return type is therefore not allowed to be void (since returntype void means that nothing is returned).

For the methods we ruled out any methods that have a primitive type as return type (like int/string). The AFP is usually used for creating more complex types (having a factory to create integers would be of very little use) and this criterion should reduce the percentage of false positives heavily, while the impact on false negatives very small.

The function we created loops through a list of objects and searches for matches to the set of characteristics we identified for the AFP. For some design pattern characteristics we use a threshold value to be able to tune our results to our needs.

```
1  private void executeBlueprintDetection()
2  {
3   interfaceMaps = mapClassesToInterfaces();
4   map[Interface interface, list[Class] classes] candidates =
   ();
5
6   for(interface <- interfaceMaps)
7   {
8     bool creationalMethodsOnly = true;
9     bool sufficientSimilarity = false;
10    bool satisfiesReturnConditions = false;
11
12    list[Class] processedClasses = [];
13    int numberOfNonConcreteReturnTypes = 0;
14    num returnTypeScore = 0.0;
15
16    if(size(interfaceMaps[interface]) > 1)
17    {
18      for(class <- interfaceMaps[interface])
19      {
20        for(method <- class.methods)
21        {
22          if(entity([primitive(_)]) := method.returnType)
23          {
24            creationalMethodsOnly = false;
25          }
26          elseif(isNonConcreteObject(_classes,
   method.returnType))
27          {
28            numberOfNonConcreteReturnTypes += 1;
29          }
30        }
31
32        if(size(class.methods) > 0)
33        {
34          returnTypeScore = numberOfNonConcreteReturnTypes
   / size(class.methods);
35        }
36        if(returnTypeScore >= 0.8)
37          satisfiesReturnConditions = true;
38
39        for(processedClass <- processedClasses)
40        {
41          if((getClassSimilarity(class, processedClass)) >=
   0.8)
42            sufficientSimilarity = true;
43        }
44
45        processedClasses += [class];
46      }
47
48      if(creationalMethodsOnly && sufficientSimilarity &&
   satisfiesReturnConditions)
```

```
49        {
50           candidates += (interface:interfaceMaps[interface]);
51        }
52     }
53   }
54
55   printCandidates(candidates);
56 }
```

The main steps that are performed in the algorithm are:

- Take each interface that has two or more implementing classes.
- Go through its implementing classes and compute a score for their methodname-returntype similarity. We used a scoring mechanism to allow for small deviations.
- Go through all methods in each class and compute a score (percentage) of the methods returning a non-concrete type object (interface or abstract class). Again we used a scoring mechanism to allow for small deviations.
- Compare the computed scores to a threshold and possibly add the instance to the list of candidates (if likely to be an instance of the corresponding design pattern).

Eventually the method returns a list of possible candidates, which is printed by a custom print method printCandidates():

```
1  ------------------------------
2  Project: |project://DPTestProject/src|
3  Extracting project...
4  Obtaining class data...
5  Obtaining interface data...
6  Obtaining method data...
7  19 Classes obtained.
8
9  Detecting occurrences of the Abstract Factory Pattern...
10 2 candidates detected:
11 AddressFactory : USAddressFactory FrenchAddressFactory
12 AnimalFactory : LandFactory SeaFactory
```

The result is printed from line number 12 on: “AddressFactory” is the name of the abstract factory and all strings after the colon, separated by a space, represent the concrete factories for the concerning abstract factory. Each detected instance of the AFP is printed on a new line.

We think this algorithm will match our requirements. The concept of using blueprints for the detection of design patterns allows for high flexibility and

customization, and the implementation of thresholds and similarity scoring functions allows for detection of modified instances of design patterns. This should lead to high ease of use and at the same time good results. In section 5 we evaluate our results to see whether our method satisfies our requirements.

4.5 Tried methods, techniques, and trade-offs

During the development process of the tool, we considered several techniques to make the detection more flexible and to allow for better detection of modified instances of the design pattern.

4.5.1 Intention mapping

One idea we had, was something we named “intention mapping”. In all cases where design patterns are implemented in a modified form in order to adjust to certain specific needs, the intention of using the design pattern is still the same as with the official description of the design pattern (as described in section 5.1), regardless of the somewhat modified implementation.

We thought of a technique in which a design pattern is built up out of several sub-systems, each with their own intention. For example we ran into the problem that we could not indisputably determine whether a method is a creational method. In jHotDraw we encountered a claimed instance of the AFP (it was documented) in which the methods of the concrete factories did not return anything. The methods did however create objects, so the intention/the “idea” behind these concrete factory objects still was in some way the same as in the official description of the AFP. If we could only say something like “the methods must have a creational intention”, where the creational intention could be represented by a family of highly related coding idioms with the same intention and the same non-functional quality attributes, we could possibly have better detection results. An example of such a family of highly related coding idioms, that represent a creational intention, would be:

- A method returns an object.
- An object is created in the method by calling a constructor (but does not necessarily return the object).
- An object is created by calling another method (which creates an object and returns it to the first method). In this case there would not be a direct constructor call in the first method.

We found however that the application of this idea was first of all very hard to realize, since it is hard to determine the “intentions” of a sub-system (there are a lot of possible cases and also a lot of exceptions to those cases), and secondly although it seemed very usable in this instance,

it was not as practical to use for other design patterns because in some other design patterns the “intentions” of the concerning design pattern are much more complicated.

Although this method has some heavy practical (and technical) problems, we still think this could prove to be a good detection mechanism if properly implemented. Validation could be preserved because this method still holds on to the original specifications (in the form of intentions) of the design patterns, while at the same time allowing for flexibility.

4.5.2 The use of graphs

Because two out of four papers in our literature study use graphs for their detection method, we also researched the possibilities of using graphs. We however found that the use of graphs is not that different from the use of blueprints. Blueprints can still detect relations among objects, but in addition also offer more possibilities the add other characteristics to improve the detection of design patterns. The use of graphs is a more abstract, uniform concept than our current solution, but that also means that a solution which uses graphs is harder to specialize to perform specific actions. Because we are looking for a solution that is easy to maintain and extend, we chose for a solution that is easy to specialize, without the use of graphs.

4.5.3 Scoring via a list of characteristics

Another method to possibly make our detection algorithm more flexible was to base our detection on a list of characteristics which were extracted from the original description of the design pattern. We could then execute the detection and see how these specific characteristics were present in the detected structure, and compute a score from it. By using a threshold on this score, we could then eliminate the candidates with a badly matching structure.

While trying to work out this idea in more detail, we however found that it is not that simple to just create a list of characteristics to match, because many characteristics are related to each other. Some characteristics are just too complicated and detailed.

We tried to see if we could implement this idea in some form, and this is how we eventually came up with the idea of integrating scoring mechanisms for certain suitable parts in our blueprint detection.

5 EVALUATION

5.1 Evaluation questions

To evaluate our research outcome and the level in which the tool eventually meets our goals, we have composed several evaluation questions in order to come to a structured evaluation and analysis of our research:

1. How well do the implementation, architecture, and detection results meet our goals and requirements?
2. Which decisions and tradeoffs were made and how have these decisions worked out for the results of our tool?
3. What notable remarks or details have come forth during the research and the development of the tool, and how have they affected the research?
4. How has the use of Rascal aided (or not) in our research on the subject of design pattern detection and source code analysis?

We will use our detection results and experiences during the development of the tool to answer these evaluation questions in the following sections.

5.2 Evaluation method

To measure and the working of our tool, we used several projects to test our tool on. First we used jHotDraw 7.6 because it is provided with documentation about the design patterns that are used. We also used the *org.eclipse.imp.pdb.values* library since it was created by thesis supervisor J. Vinju, so the location(s) of any implemented design patterns were also known in this case. Our third test case project is a project that we created ourselves and which we filled with various code examples (implementations of the Abstract Factory Pattern) from online articles about the AFP.

We executed our tool on these projects. To evaluate our trade-offs, we disabled and enabled different parts of our tool and changed our threshold values to measure the differences in results.

We analyzed our tool results by comparing the detection results to the documentation provided with the test projects, but also by evaluating detected and non-detected instances ourselves. This is important because of the large amount of modified pattern implementations (compared to the official design pattern definitions). It is not that clear what our tool should and should not detect, and that makes it hard to state specific search criteria. Proper evaluation of the results is therefore especially important, to see where adjustments in the tool and thresholds should be made.

5.3 Results

Below you will find a summary of our detection results. These results are based on the documentation of AFP occurrences, although several occurrences do in our opinion not qualify as occurrences of the AFP.

Detected instances	jHotDraw 7.6	org.eclipse.imp.pdb.values	Online articles
Abstract Factory Pattern	**0** out of 3	**2** out of 1	**2** out of 3

Table 3 Detection results summary

Our tool was unable to detect any of the three documented AFP occurrences in jHotDraw, mostly due to deviating implementations of the design pattern.

In the org.eclipse.imp.pdb.values library the single known occurrence of the AFP was successfully detected, but additionally our tool also came up with a possible second occurrence of the AFP. We could not indisputably assess whether the second occurrence could be counted as an instance of the AFP. We discuss this issue in section 5.5.

In our third test case, two out of three occurrences of the AFP were detected. The third instance was not detected due to the fact that our tool only checks for interface abstract factories, and not for abstract classes as abstract factory. This was a limitation known to us up front.

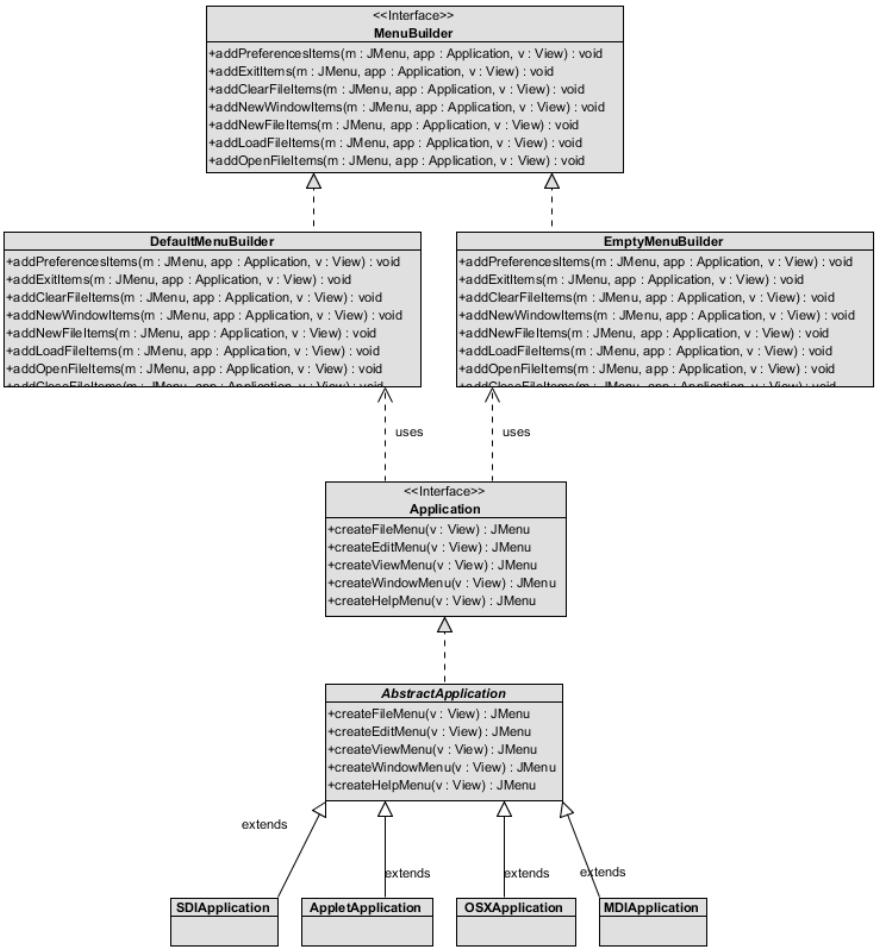
5.3.1 jHotDraw 7.6

In the jHotDraw 7.6 project 0 instances of the AFP were detected, out of the 3 instances that are documented.

A quick look into the code however made clear why our tool was unable to detect these occurrences of the AFP:

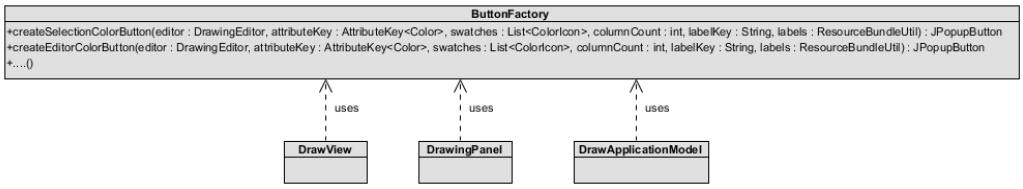
- The first documented instance of the AFP is a highly modified implementation of the AFP. The intentions behind the use of the AFP are still visible, but structure-wise there is little resemblance left with the structure of the AFP. There are no concrete factories implementing an abstract factory. Instead there are “builders” implementing a general builder interface (these builders are documented as the AFP). These builders do however only modify objects. They do not create new objects and also do not return any objects. They are helper methods for a third class, called “Application”, which actually creates the menu objects and lets the builder classes “set-up” and modify these objects to match specific needs.

This structure can be seen in the following diagram:

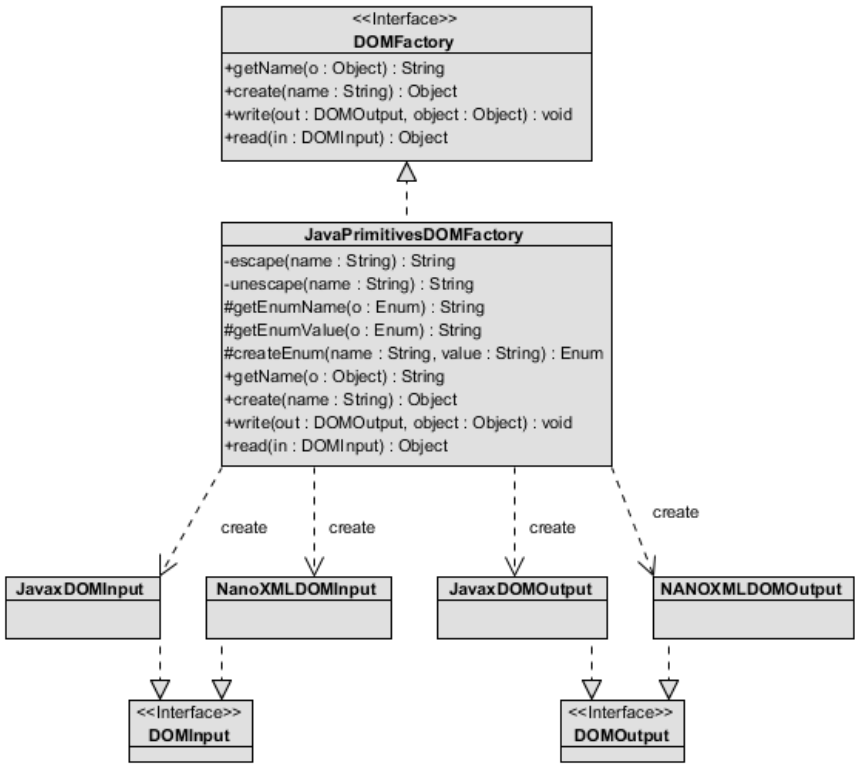


- The second implementation of the AFP was more peculiar: it barely had any of the characteristics of the AFP. The implementation that was documented as AFP here, only consisted of a single class with some creational methods, without implementing or extending any interfaces or abstract classes. This implementation clearly did not qualify as an AFP instance. Perhaps a documentation error occurred here?

The following model show the structure of this second documented instance. The real ButtonFactory class contains many more similar methods, but they are not shown in the diagram below in order to preserve clarity:



- The reason the third instance of the AFP was not detected by our tool, was because the methods of the factories in this instance all returned primitive types, like integers or strings. In our tool we have excluded all instances that have methods which only return primitive types, in order to narrow down our list of possible AFP instances. We reasoned that the AFP is usually used to create "complex" objects, so we think the influence of false positives here is much bigger and more important, than the increase of false negatives that this measure brings along. In addition, the implemented structure is also here not according to the design pattern definition; the concrete factory does not return objects of a custom abstract type (but instead objects of the abstract type are create via the out-parameter).



A result of 0 out of 3 detected occurrences may at first seem disappointing, but further analysis shows that our tool performance was okay. All

occurrences that were documented as AFP were heavily modified or had no resemblance with the AFP at all. Although the instances were named and documented as occurrences of the AFP, there were no actual occurrences of the AFP in jHotDraw.

5.3.2 org.eclipse.imp.pdb.values

The detection results for the org.eclipse.imp.pdb.values library were more promising. The library does not come with documentation, but since it was implemented by J. Vinju (this thesis' supervisor) the implemented instances of the AFP were known: the library should contain one instance of the AFP that was knowingly implemented.

Our tool identified two instances of the AFP, of which one was the instance that was knowingly implemented.

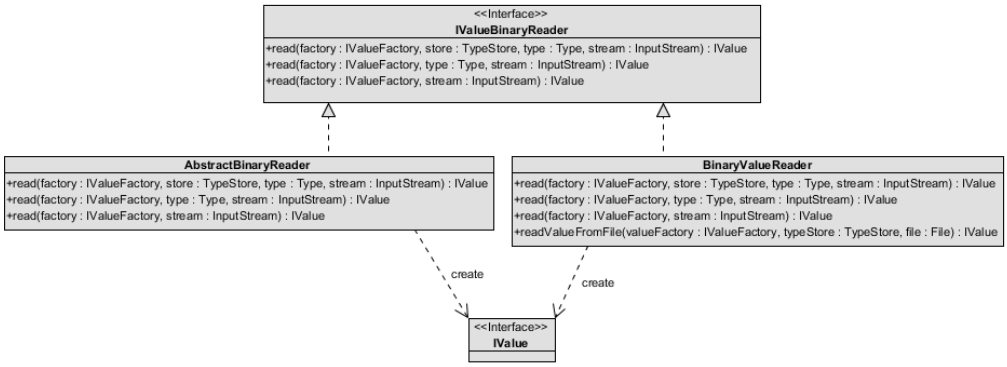
This result raised the question whether the second result was also an instance of the AFP, which was "accidentally" implemented (after all, the goal of this tool was to detect instances of design patterns that were implemented unknowingly and therefore did not have the "appropriate" naming). We analyzed and judged the structure that was identified as a possible instance of the AFP by our tool, but we could not come up with an indisputable answer to this question. According to the characteristics that we extracted from the AFP, this instance would qualify as an occurrence of the AFP, but still in our eyes this was not the case. We will further discuss this issue in section 5.5.

Detected instances	Abstract factory	Concrete factories (clients)
#1	org.eclipse.imp.pdb.facts.io.*IValueBinaryReader*	org.eclipse.imp.pdb.facts.io.*BinaryValueReader* org.eclipse.imp.pdb.facts.io.*AbstractBinaryReader*
#2 **true positive**	org.eclipse.imp.pdb.facts.*IValueFactory*	org.eclipse.imp.pdb.facts.impl.*BaseValueFactory* org.eclipse.imp.pdb.facts.impl.shared.*SharedValueFactory*

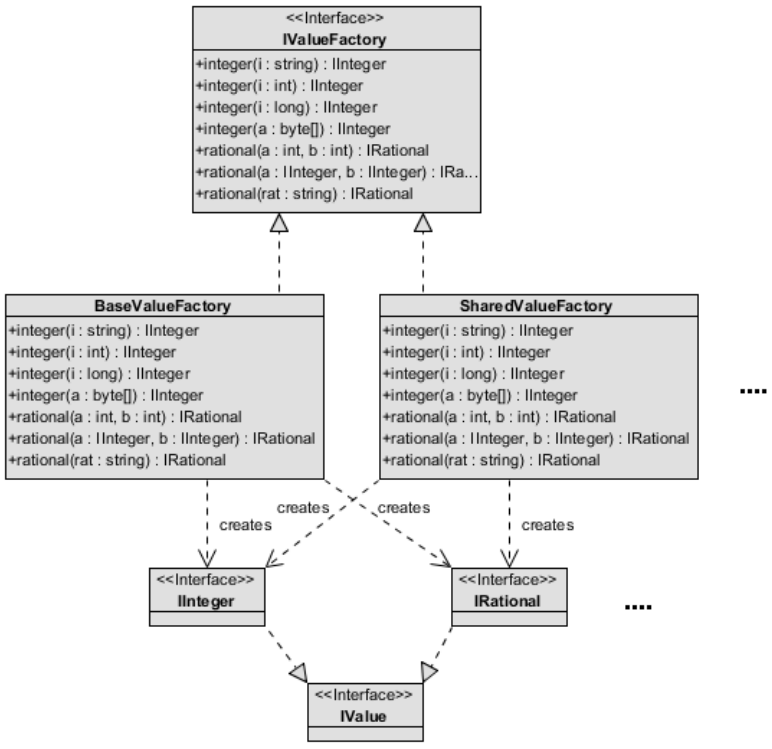
Table 1 Detection results for org.eclipse.imp.pdb.facts library

Detected instance #2 is a true positive. The detected structure consists of an abstract factory object with two implementing factories, which have a high similarity score and have mainly creational methods.

The diagram of the first detected instance is as follows. The IValue interface is implemented by numerous classes and interfaces. These interfaces and classes are not shown in this diagram:



The model of the second detected instance is as follows. Also here objects of numerous other types are created, the diagram is incomplete. This is to preserve clarity in the diagram. No objects of structural importance are left out:



5.3.3 Various online article implementations

The third project we used to test our detection tool on, is a project which we created ourselves, and which we filled with implementations of the AFP from online articles explaining the AFP:

1. Javapapers.com
   http://javapapers.com/design-patterns/abstract-factory-pattern/

2. Sourcemaking.com
   http://sourcemaking.com/design_patterns/abstract_factory/java/2#

3. Sun Microsystems
   http://java2s.com/Code/Java/Design-Pattern/AbstractFactoryPatterninJava2.htm

The project contains three (claimed) implementations of the AFP, of which our tool identified two. Further investigation turned out that the implementation from the second article (from Sourcemaking.com) does not use an interface for its abstract factory but an abstract class instead. Strictly speaking this could therefore also be considered a "false" implementation of the AFP, but again this is arguable, since this could be considered as just a "technical" difference: an empty abstract class has practically the same semantic characteristics as an interface, and therefore both an abstract class or an interface can be used in this case.

Detected instances	Abstract factory	Concrete factories (clients)
#1 **true positive**	sunmicrosystems.*AddressFactory*	sunmicrosystems.*USAddressFactory* sunmicrosystems.*FrenchAddressFactory*
#2 **true positive**	javapapers.*AnimalFactory*	javapapers.*SeaFactory* javapapers.*LandFactory*

Table 2 Detection results for our custom "online article implementations" project

Detected instances #1 and #2 are both true positives. The detected structures consist of an abstract factory object with two implementing factories (interface or abstract class), which have a high similarity score and have mainly creational methods.

5.4 Analysis and interpretation of results

Overall we are satisfied with the detection results. Our tool did not detect some of the instances that were documented as AFP, but in our opinion these occurrences were not actual instances of the AFP due to incorrect or heavily modified implementations. Our modularity and flexibility requirements are also satisfied the designed architecture, which separates the logic for each design pattern and thereby makes it easy to change, remove, and add detection algorithms for any design pattern of choice.

Part of the research results however also raised a difficult question. For the org.eclipse.imp.pdb.values library our tool also detected a second possible occurrence of the AFP, aside from the officially documented occurrence. This is interesting, since the main goal of our tool is actually to detect instances of design patterns that are unknowingly implemented and thus have "wrong" or unclear naming. Code inspection showed us that the second instance is a binary reader: an interface IValueBinaryReader which is implemented by two classes BinaryValueReader and AbstractBinaryReader. Both implementing classes have methods that actually create objects and return them, so by looking purely at the characteristics of the detected structure, we could indeed qualify the found instance as an instance of the AFP. Could it be that readers or parsers with a similar structure are in fact also factories? If we look back at the definition by the GoF in section 4.4, we conclude that this might indeed be the case.

We however still identified a problem here: the definition and qualification for the design pattern is not clear enough. It mostly describes an intention, instead of a structure. We could not give an indisputable answer to whether we had found an additional instance of the AFP. We already encountered some assessment problems during the development of the tool, in which it was hard to decide in some cases whether a certain structure would qualify as an occurrence of the AFP, which made it hard for us to assess in some cases what structures should be detected and what shouldn't (which made it also difficult to decide whether we should change our detection algorithm or not).

5.5 The problem of disputable definitions and practical use

As addressed in section 5.4, the problem of disputable definitions and the practical use of those definitions forms quite a big problem in the analysis and validation of our tool and results.

The definitions and descriptions of the design patterns (e.g. by the Gang of Four [5]) seem very clear and unambiguously, but during our development process we found out that these definitions might not be as clear as they seem to be. We detected structures that, according to their architecture, would qualify as instances of the AFP, but in our opinion actually were not indisputably instances of the AFP (in the org.eclipse.imp.pdb.values project). We analyzed that there is one main aspect that brings the clarity that is needed in this case: intentions. In deciding and assessing on what name should be put to a certain structure, it is very important to know what the intentions were of the developer that created the related piece of code.

This also brings us to the second part of the problem, practical use. During our research we also noted that the amount of edited and customized instances of the design pattern were even much higher than we had anticipated. We found very little instances of the AFP that were implemented according to the exact descriptions and definitions by the GoF [5]. This led to a second problem we encountered: if we would have made the detection algorithm so that it would only

detect instances of the AFP that strictly match the official definition, the tool would not be of much use in practice and have a lot of false negatives. We somehow had to find a way to create an algorithm that was able to detect customized instances of the AFP, without presenting us with a load of false positives.

We again analyzed the AFP pattern and tried to find out how we could minimize or change our set of characteristics needed for the detection, but found that whichever solution we came up with, we had a validation problem. We had some ideas of how we could possibly improve our detection, but with all methods we tried, questions raised which we found ourselves unable to answer:

- If we change the characteristics/blueprint for our detection algorithm in such a way that they do not match the official definitions anymore, how can we still justify our results and how do we validate our detection?
- If we are going to change our detection algorithm in such a way that it can also detect occurrences of modified versions of a design pattern, where do we draw the line? What modifications do still qualify as an instance of the concerning design pattern and what modifications don't? Even in already detected instances we found it in some cases hard to judge whether an instance qualified as an instance of the AFP or not.

The bottom-line here is that because of the enormous usage of modified patterns, design pattern detection becomes very hard because it is not clear anymore what the tool should actually detect. This vagueness about what to detect, makes it hard to develop, and especially, to validate a tool on its correctness and performance.

5.6 Tradeoffs in the current algorithm

It's clear by now that our detection tool is the result of a lot of decisions and tradeoffs. The most important tradeoffs are described in this section:

- Our detection tool will only detect instances of the AFP that use two or more concrete factories. Strictly speaking it is also possible to create an instance of the AFP with only one concrete factory, but in this case results showed an increase in false positives. Secondly one could argue that the use of the AFP is only useful when having to or more concrete factories, and practice indeed shows that instances of the AFP with only one concrete factory barely occur.
- Our detection tool searches for concrete factories that have mostly methods with a return type other than the primitive types. There is not much use in using an AFP for creating primitive types (primitive types are not abstract), since the whole intention of the AFP is to have a structured way to create different objects from the same object family. Cases in which the use of the AFP in combination with the creation of primitive objects is truly justifiable are very rare, and also in this case the

inclusion of the returning of primitive objects would lead to a big increase in false positives.

- To further improve the results, we compare the classes which implement the interface (the possible concrete factories) to see if the classes are similar. The concrete factories should implement the methods from the interface, and no (or barely no) other methods. We generate a "similarity" score based on the similarity of method names and their return types among classes. On our test projects this measure did not make any difference in the results. We however still think this measure is useful in bigger projects where more possible candidates have to be filtered.

By applying these techniques/mechanisms, we were able to implement some flexibility without having a heavy increase in false positives.

The following table show the detection results when disabling or adjusting the measures and thresholds mentioned above:

Settings	jHotDraw	Org.eclipse..	Online articles
Concrete factories: >= 2 Returntype score: >= 0.8 Concr.fact. similarity score: >= 0.8	**0**	**2**	**2**
Concrete factories: >= 2 Returntype score: >= 0.8 Concr.fact. similarity score: disabled	0	2	2
Concrete factories: >= 1 Returntype score: >= 0.8 Concr.fact. similarity score: disabled	0	3 (+1)	2
Concrete factories: >= 2 Returntype score: >= 0.1 Concr.fact. similarity score: disabled	0	3 (+1)	2
Concrete factories: >= 2 Returntype score: disabled Concr.fact. similarity score: disabled	1 (+1)	5 (+3)	2
Concrete factories: >= 2 Returntype score: disabled Concr.fact. similarity score: disabled *Check for primitive return types disabled*	50 (+50)	30 (+28)	3 (+1)
Concrete factories: >= 2 Returntype score: disabled Concr.fact. similarity score: >= 0.8 *Check for primitive return types disabled*	36 (+36)	28 (+26)	3 (+1)

These results show that especially the check for primitive return types is of major importance. The returntype score (which checks if a return type is of an abstract type) further improves the results even with a very low threshold of 0.1. The check for abstract factories with 2 or more concrete factories eventually filters out one false positive.

The similarity scoring method, which checks similarity of the concrete factories and if the concrete factories do not have specific methods that are not prescribed by their interface, does not make a difference in our results. This is however because the other measures already filtered out a lot of false positives. If we run the class similarity measure while the other measures are disabled, we can see that this measure does also make a difference in the results.

5.7 Evaluation of the use of Rascal in this research

A secondary goal of our research was to evaluate the use of Rascal in source code analysis.

The JDT functions that are available in Rascal definitely have proven to be of huge help during the development of our tool. The JDT functions make it easy to extract classes, interfaces, methods, modifier info, and more objects and information, with just a single command. It was not necessary to create any form of project crawler to retrieve this information ourselves, so this did not only save us time in development, but it also provides reliability on this part of the code that we did not have to worry about.

After seeing that Rascal made this part of the analysis so easy, it is somewhat a pity to see how the extracted information thereafter is presented. All information that is returned is presented in a relation, mostly containing Entity objects which hold the information we were looking for. The Entity objects simply hold a list of id's, but from which the information can basically only be extracted using a visitor and pattern matching. Although this allows for flexibility and it has very little overhead in retrieving and converting information you might not need, we experienced this to be a bit devious and in contrast with the ease of which we could at first extract the information and relations from the source code.

During our tool development and testing we however experienced a major drawback of using Rascal for our design pattern detection: Rascal can only analyze and operate on projects that are loaded into the Eclipse environment, and most important, do actually compile. In order to be able to make a comparison to other research results, some older source projects (some of which haven't had an update in 5-10 years) were very important and interesting for our research, but we simply could not get those projects to compile anymore, due to various reasons, and we could therefore not run our detection tool on those projects.

This really limited us in the validation of our tool and the ability to draw the wanted conclusions from our research.

5.8 Conclusion

The evaluation of our tool results shows us that our tool performed well on the detection of the AFP. We integrated several measures in our tool to improve the detection of modified instances of the AFP, and our results show that these measures indeed improved the results by a great deal.

The analysis however also showed us that the detection of modified instances of design patterns is and will still remain a difficult subject. The main reason for this is that definitions of design patterns are often vague and mainly describe intentions. A possible solution to this problem could be to develop a detection method that can detect intentions, for example by defining families of related code-idioms which represent certain intentions (e.g. a family of code idioms which can be used for a creational method). A problem with this concept is however that some design patterns have complicated relations between objects, which are hard to describe by intentions. The intention-mapping concept can be valuable in some cases, but there are other cases in which a different approach is needed.

The use of Rascal proved useful in our research. Rascal allows for easy and reliable source code analysis. A big limitation in Rascal was however that it can only analyze source code from projects that are able to compile. Projects that do not compile, because of a missing library or various other reasons, cannot by analyzed, which caused us to remove several test projects from our corpus.

6 CONCLUSION

In order to improve correct identifier naming and improve program comprehension when documentation is missing, we researched the detection of design patterns in program code. We discussed two sub research questions to get an answer to the following research question:

> *"What is the best way to perform design pattern detection on the code of a software product without using identifiers information and how well does this work?"*

We analyzed four existing methods for design pattern detection. None of these methods offered a suitable solution for our requirements. We therefore composed our own detection method, partly based on concepts from the four research detection methods. Our method is based on blueprint coding for the design patterns, which is easy to maintain and perfect. To be able to detect modified versions of design patterns, we implemented several scoring mechanisms to allow for deviation in our blueprint matching.

The results showed that our tool performed well if occurrences of the design pattern have enough resemblance to the original definition of the Abstract Factory Pattern (AFP) by the GoF [5]. Our results and analysis however also showed that in many cases structures are presented as instances of the AFP while the concerning structures show little resemblance with the structure of the official descriptions of the AFP. The intentions of the concerning structures often match up to the intentions and definitions by the GoF, but are heavily modified in order to fit into existing architectures. This makes it very hard to detect such structures, purely based on architectural characteristics.

We concluded that a good way to detect design patterns might instead be to detect intentions of structures in program code. This could be realized by using families of highly related code-idioms which represent (parts of) intentions. This concept, which we call "intention-mapping", however also has some difficulties: not all design patterns are easily describable by intentions, or become much too abstract when described by intentions. The best solution would therefore be to combine architectural detection with intentions-mapping.

6.1 Future work

For our tool we focused on the detection of the Abstract Factory Pattern. The detection of other design patterns is not yet implemented in our tool, and is therefore an interesting subject for future work. Each design pattern has its own

specific characteristics and might bring to light new difficulties in design pattern detection.

A second subject for future work is automatic suggestion of identifier names, based on the detection results of our tool. Our main goal in the creation of our tool was to detect improper naming. It would be interesting to see if the tool could be extended so that it could also automatically suggest better naming.

Another interesting subject is of course the research in intention-mapping, as described in our research. We believe that design pattern detection which is (partly) based on the detection of intentions is a very promising concept which should definitely be researched.

7 REFERENCES

[1] G. Canfora and A. Cimitile, “Software Maintenance.” Benevento, Italy, 2000.

[2] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, “Software complexity and maintenance costs,” *Communications of the ACM*, vol. 36, no. 11, pp. 81–94, 1993.

[3] H. Morgan, “Characteristics of Application Software Maintenance,” *Communications of the ACM*, vol. 21, no. 6, pp. 466–471, 1978.

[4] C. Jones, “The economics of software maintenance in the twenty first century,” no. Version 3. 2006.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design Patterns : Abstraction and Reuse of Object- Oriented Design,” 1993, pp. 406–431.

[6] L. Prechelt, B. Unger, and M. Philippsen, “Documenting Design Patterns in Code Eases Program Maintenance,” *Proc. ECSE Workshop on Process Modeling and Empirical Studies of Software Evolution*, pp. 72–76, 1997.

[7] L. Prechelt, I. C. Society, and B. Unger-lamprecht, “Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance,” *IEEE Transactions on software engineering*, vol. 28, no. 6, pp. 595–606, 2002.

[8] L. Stevens, “Automatically Analyzing the Consistency and Preciseness of Class Names,” University of Amsterdam, 2012.

[9] D. van Leeuwen, “Comprehensible Method Names: Focusing on the Nouns,” University of Amsterdam, 2012.

[10] J. Stoel, “Exploring the Detection of Method Naming Anomalies,” University of Amsterdam, 2012.

[11] P. Klint, T. van der Storm, and J. Vinju, “EASY Meta-programming with Rascal: Leveraging the extract-analyze-synthesize paradigm for meta-programming.,” *Proceedings of the 3rd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE’09), LNCS*, 2010.

[12] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, “What’s in a name? A study of identifiers,” in *14th IEEE International Conference on Program Comprehension, ICPC 2006.*, 2006, pp. 3–12.

[13] F. Deissenboeck and M. Pizka, “Consice and consistent naming,” *Software quality journal*, vol. 14, no. 3, pp. 261–282, 2006.

[14] D. Lawrie, H. Feild, and D. Binkley, “Syntactic Identifier Conciseness and Consistency,” *Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 139–148, 2006.

[15] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, “Automatic design pattern detection,” *MHS2003. Proceedings of 2003 International Symposium on Micromechatronics and Human Science (IEEE Cat. No.03TH8717)*, pp. 94–103, 2003.

[16] K. Kontogiannis, “Evaluation experiments on the detection of programming patterns using software metrics,” *Proceedings of the Fourth Working Conference on Reverse Engineering*. IEEE Comput. Soc, Waterloo, Canada, pp. 44–54, 1997.

[17] N. Pettersson and W. Lowe, “Efficient and Accurate Software Pattern Detection,” *2006 13th Asia Pacific Software Engineering Conference (APSEC’06)*, pp. 317–326, 2006.

[18] D. Beyer and C. Lewerentz, “CrocoPat: efficient pattern analysis in object-oriented programs,” *MHS2003. Proceedings of 2003 International Symposium on Micromechatronics and Human Science (IEEE Cat. No.03TH8717)*, pp. 294–295, 2003.

[19] N. Tsantalis, A. Chatzigeorgiou, I. C. Society, G. Stephanides, and S. T. Halkidis, “Design Pattern Detection Using Similarity Scoring,” *IEEE Transactions on software engineeringsoftware engineering*, vol. 32, no. 11, pp. 896–909, 2006.

[20] A. Chatzigeorgiou, N. Tsantalis, and G. Stephanides, “Application of graph theory to OO software engineering,” *Proceedings of the 2006 international workshop on Workshop on interdisciplinary software engineering research - WISER ’06*, p. 29, 2006.

8 APPENDIX A: RASCAL CODE

8.1 DataTypes.rsc

```
module detectiontool::Datatypes

import lang::java::jdt::Java;

// Data elements to describe several objects
data Class = Class (Entity entityObject, str name, list[Interface]
interfaces, list[str] package, list[str] classes, list[Method]
methods, list[str] modifiers, loc location);
data Method = Method (Entity entityObject, str name, str
className, list[str] classes, Entity returnType, list[Id]
returnTypeIds, loc location);
      data Interface = Interface (Entity entityObject, str name,
      list[Id] ids);
```

8.2 Main.rsc

```
module detectiontool::Main

// Project libraries
import detectiontool::Datatypes;
import detectiontool::DataObtainer;
import detectiontool::Util;

// Design patterns
import detectiontool::detection::AFP;

// System libraries
import IO;

import lang::java::jdt::JDT;
import lang::java::jdt::Java;
import lang::java::jdt::JavaADT;

private void detect(loc projectLocLocal)
{
  // Print progress
  println("------------------------------");
  println("Project: <projectLocLocal>");

  classes = obtainData(projectLocLocal);

  println("");

  //--- Call all the design pattern detection methods ----

  detectiontool::detection::AFP::detect(classes);

  //-----------------------------------------------------
```

```
  println("");
}

// Run detection
public void runDetection()
{
  detect(|project://JHotDraw7/src/main/java|);
  detect(|project://org.eclipse.imp.pdb.values/src|);
  detect(|project://DPTestProject/src|);
}
```

8.3 DataObtainer.rsc

```
module detectiontool::DataObtainer

// Project libraries
import detectiontool::Util;
import detectiontool::Datatypes;

// System libraries
import util::Resources;

import String;
import List;
import IO;
import Real;
import Set;

import lang::java::jdt::Java;
import lang::java::jdt::JDT;
import lang::java::jdt::JavaADT;

// Obtain all project, class and method data from the given
project
public list[Class] obtainData(loc project)
{
  println("Extracting project... ");
  resources = extractProject(project);

  println("Obtaining class data...");
  classes = extractClasses(resources);
  classes = addModifierInfoToClasses(resources, classes);

  println("Obtaining interface data...");
  classes = addInterfaceInfoToClasses(resources, classes);

  println("Obtaining method data...");
  classes = mapMethodsToClasses(resources, classes);

  println("<toString(size(classes))> Classes obtained.");

  return classes;
}

// Obtain class information
private list[Class] extractClasses(resources)
```

```
{
  list[Class] classList = [];

  visit (resources@classes)
  {
    case <loca, class>:
    {
      list[str] package = [];
      list[str] classes = [];

      top-down-break visit(class)
      {
        case package(p): package += p;
        case class(c): classes += c;
        case class(c,_): classes += c;
      }

      classList += Class(class, last(classes), [], package,
classes, [], [], loca);
    }
  }

  return classList;
}

// Add interface information to classes
private list[Class] addInterfaceInfoToClasses(resources,
givenClasses)
{
  list[Class] classList = givenClasses;

  top-down-break visit(resources@implements)
  {
    case <implementer, implemented>:
    {
      list[str] package = [];
      list[str] classes = [];
      str interfaceName = "";
      list[Id] interfaceEntityIds = [];

      top-down-break visit(implementer)
      {
        case method(_,_,_): ;
        case package(p): package += p;
        case class(c): classes += c;
        case class(c,_): classes += c;
      }
      top-down-break visit(implemented)
      {
        case entity(eIds): interfaceEntityIds = eIds;
      }
      top-down-break visit(implemented)
      {
        case interface(i): interfaceName = i;
        case interface(i, _): interfaceName = i;
      }

      bool classFound = false;
```

```
        for(class <- classList) // Loop through existing classes to
find the right class
        {
          if(class.entityObject == implementer && !classFound) //
Class found, add method to its method list
          {
            newInterface = Interface(implemented, interfaceName,
interfaceEntityIds);
            classNew = Class(class.entityObject, class.name,
class.interfaces + newInterface, class.package, class.classes,
class.methods, class.modifiers, class.location);
            classList = delete(classList, indexOf(classList,
class)); // Delete old class
            classList = classNew + classList; // Add edited class
            classFound = true;
          }
        }
      }
    }

    return classList;
}

// Add modifier information to classes
private list[Class] addModifierInfoToClasses(resources,
givenClasses)
{
    list[Class] classList = givenClasses;

    top-down-break visit(resources@modifiers)
    {
      case <entity, modifier>:
      {
        list[str] package = [];
        list[str] classes = [];
        str modifierString = "";

        top-down-break visit(entity)
        {
          case package(p): package += p;
          case class(c): classes += c;
          case class(c,_): classes += c;
        }

        top-down-break visit(modifier)
        {
          case \public(): modifierString = "public";
          case protected(): modifierString = "protected";
          case \private(): modifierString = "private";
          case static(): modifierString = "static";
          case abstract(): modifierString = "abstract";
          case final(): modifierString = "final";
          case native(): modifierString = "native";
          case synchronized(): modifierString = "synchronized";
          case transient(): modifierString = "transient";
          case volatile(): modifierString = "volatile";
          case strictfp(): modifierString = "strictfp";
          case deprecated(): modifierString = "deprecated";
        }
```

```
        bool classFound = false;

        for(class <- classList) // Loop through existing classes to
find the right class
        {
          if(class.entityObject == entity && !classFound) // Class
found, add method to its method list
          {
            classNew = Class(class.entityObject, class.name,
class.interfaces, class.package, class.classes, class.methods,
class.modifiers + modifierString, class.location);
            classList = delete(classList, indexOf(classList,
class)); // Delete old class
            classList = classNew + classList; // Add edited class
            classFound = true;
          }
        }
      }
    }

    return classList;
}

// Map all methods to their class
private list[Class] mapMethodsToClasses(resources, givenClasses)
{
    list[Class] classList = givenClasses;

    visit(resources@methodDecls)
    {
      case <loca, method>:
      {
        list[str] package = [];
        list[str] classes = [];
        list[Method] methods = [];

        top-down-break visit(method)
        {
          case package(p): package += p;
          case class(c): classes += c;
          case class(c,_): classes += c;
          case method(name,_,entity(returnTypeIds)): methods +=
Method(method, name, getLastFromList(classes), classes,
entity(returnTypeIds), returnTypeIds, loca);
        }

        bool classFound = false;
        for(class <- classList) // Loop through existing classes to
find the right class
        {
          if(class.classes == classes && class.package == package
&& !classFound) // Class found, add method to its method list
          {
            methodsNew = class.methods + methods;
            classNew = Class(class.entityObject, class.name,
class.interfaces, class.package, class.classes, methodsNew,
class.modifiers, class.location);
```

```
            classList = delete(classList, indexOf(classList,
class)); // Delete old class
            classList = classNew + classList; // Add edited class
            classFound = true;
          }
        }
      }
    }

    return classList;
      }
```

8.4 Detection/AFP.rsc

```
module detectiontool::detection::AFP

// Project libraries
import detectiontool::Datatypes;
import detectiontool::Util;

// System libraries
import util::Math;

import IO;
import List;
import Map;
import String;
import Set;

import lang::java::jdt::JDT;
import lang::java::jdt::Java;
import lang::java::jdt::JavaADT;

// Variables
list[Class] _classes = [];

// Detect all occurrences of the Abstract Factory Pattern
public void detect(list[Class] projectData)
{
  println("Detecting occurrences of the Abstract Factory
Pattern...");
  _classes = projectData;
  executeBlueprintDetection();
}

// Start the pattern detection
private void executeBlueprintDetection()
{
  interfaceMaps = mapClassesToInterfaces();
  map[Interface interface, list[Class] classes] candidates = ();

  for(interface <- interfaceMaps) // Go through all interfaces
  {
    bool creationalMethodsOnly = true;
    bool sufficientSimilarity = false;
    bool satisfiesReturnConditions = false;

    list[Class] processedClasses = [];
```

```
    int numberOfNonConcreteReturnTypes = 0;
    num returnTypeScore = 0.0;

    if(size(interfaceMaps[interface]) > 1) // Find interfaces
that have at least 2 implementing classes
    {
      for(class <- interfaceMaps[interface]) // Go through each
class that implements the current interface
      {
        for(method <- class.methods) // Go through all the
methods per class and see if it has only creational methods (this
rule might be made optional to allow detection of edited versions
of the pattern)
        {
          if(entity([primitive(_)]) := method.returnType)
          {
            creationalMethodsOnly = false;
          }
          elseif(isNonConcreteObject(_classes,
method.returnType))
          {
            numberOfNonConcreteReturnTypes += 1;
          }
        }

        if(size(class.methods) > 0)
        {
          returnTypeScore = numberOfNonConcreteReturnTypes /
size(class.methods);
        }

        if(returnTypeScore >= 0.8)
          satisfiesReturnConditions = true;

        for(processedClass <- processedClasses)
        {
          if((getClassSimilarity(class, processedClass)) >= 0.8)
            sufficientSimilarity = true;
        }

        processedClasses += [class];
      }

      if(creationalMethodsOnly && sufficientSimilarity &&
satisfiesReturnConditions) // Pattern candidate found, IF the
found structure has only creational methods and the implementing
classes are similar enough
      {
        candidates += (interface:interfaceMaps[interface]);
      }
    }
  }

  printCandidates(candidates);
}

// Map all classes to their interfaces
private map[Interface interface, list[Class] classes]
mapClassesToInterfaces()
```

```
{
  map[Interface interface, list[Class] classes] interfaces = ();

  for(class <- _classes)
  {
    if(size(class.interfaces) > 0)
    {
      for(interface <- class.interfaces)
      {
        if(size(interfaces) > 0 && interface in interfaces)
        {
          interfaces[interface] = interfaces[interface] +
[class];
        }
        else
        {
          interfaces[interface] = [class];
        }
      }
    }
  }

  return interfaces;
}

// See how similar two classes are, return value between 0 and 1
private num getClassSimilarity(Class class1, Class class2)
{
  real similarityScore = 0.0;
  int matchingMethodSignatures = 0;
  int numberOfMethodsInClass2 = 0;

  rel[str methodName, Entity returnType] classMethods1 = {};
  rel[str methodName, Entity returnType] processedMethods = {};
// Prevent wrong similarity score due to overloaded methods

  for(method <- class1.methods) // Go through all the methods of
the first class
  {
    if(<method.name, method.returnType> notin classMethods1) //
Don't count overloaded methods more than once
      classMethods1 += <method.name, method.returnType>;
  }

  for(method <- class2.methods) // Go through all the methods of
the second class and find matching methods in the first class
  {
    if(<method.name, method.returnType> notin processedMethods)
    {
      if(<method.name, method.returnType> in classMethods1) //
Check for a method with the same name and return type
      {
        matchingMethodSignatures += 1;
      }

      numberOfMethodsInClass2 += 1;
      processedMethods += <method.name, method.returnType>;
    }
  }
```

```
  if(size(classMethods1) == 0 || numberOfMethodsInClass2 == 0)
  {
    similarityScore = 0.0;
  }
  else
  {
    similarityScore = matchingMethodSignatures /
toReal(min({size(classMethods1), numberOfMethodsInClass2}));
  }

  return similarityScore;
}

private void printCandidates(map[Interface interface, list[Class]
classes] candidates)
{
  print("<size(candidates)> candidates detected");

  if(size(candidates) > 0)
    println(":");
  else
    println(".");

  for(candidate <- candidates)
  {
    print("<candidate.name> :");
    for(class <- candidates[candidate])
    {
      print(" <class.name>");
    }
    println("");
  }
}
```

8.5 Util.rsc

```
module detectiontool::Util

import IO;
import String;
import List;
import Real;

import detectiontool::Datatypes;

import lang::java::jdt::Java;
import lang::java::jdt::JDT;
import util::Resources;

// Checks whether a given Entity object is a non concrete object
public bool isNonConcreteObject(list[Class] classList, Entity
entityObject)
{
  bool nonConcreteObject = false;
```

```
  list[str] package = [];
  list[str] classes = [];

  top-down-break visit(entityObject)
  {
    case interface(i): nonConcreteObject = true;
    case interface(i, _): nonConcreteObject = true;

    case package(p): package += p;
    case class(c): classes += c;
    case class(c,_): classes += c;
  }

  if(!nonConcreteObject)
  {
    for(class <- classList)
    {
      if(class.package == package && class.classes == classes &&
"abstract" in class.modifiers)
        nonConcreteObject = true;
    }
  }

  return nonConcreteObject;
}


public str getLastFromList(list[str] itemList)
{
  str lastItem = "";

  if(size(itemList) > 0)
    lastItem = last(itemList);

  return lastItem;
}
```