On the Existence of High-Impact Refactoring Opportunities in Programs

Jens Dietrich[1] **Catherine McCartin**[1] **Ewan Tempero**[2] **Syed M. Ali Shah**[1]

[1] School of Engineering and Advanced Technology
Massey University, Palmerston North, New Zealand
Email: {j.b.dietrich,c.m.mccartin,m.a.shah}@massey.ac.nz

[2] Department of Computer Science
University of Auckland, Auckland, New Zealand
Email: e.tempero@cs.auckland.ac.nz

Abstract

The refactoring of large systems is difficult, with the possibility of many refactorings having to be done before any useful benefit is attained. We present a novel approach to detect starting points for the architectural refactoring of large and complex systems based on the analysis and manipulation of the type dependency graph extracted from programs. The proposed algorithm is based on the simultaneous analysis of multiple architectural antipatterns, and outputs dependencies between artefacts that participate in large numbers of instances of these antipatterns. If these dependencies can be removed, they represent high-impact refactoring opportunities: a small number of changes that have a major impact on the overall quality of the system, measured by counting architectural antipattern instances. The proposed algorithm is validated using an experiment where we analyse a set of 95 open-source Java programs for instances of four architectural patterns representing modularisation problems. We discuss some examples demonstrating how the computed dependencies can be removed from programs. This research is motivated by the emergence of technologies such as dependency injection frameworks and dynamic component models. These technologies try to improve the maintainability of systems by removing dependencies between system parts from program source code and managing them explicitly in configuration files.

1 Introduction

Software systems are subject to change. However, changing software is risky and expensive. The development of methodologies and tools to deal with change, and to minimise risks and expenses associated with change is one of the great challenges in software engineering. Refactoring is a successful technique that has been developed in order to facilitate changes in the code base of programs. First developed in the late 90s, code refactoring tools have become commodities for many programmers, and refactoring is one of the main supportive technologies for agile process models such as Scrum and extreme programming. The first generation of refactoring tools has focused on the manipulation of source code, using the structure of the source code (in particular the abstract syntax tree (AST)) as the data structure that is being manipulated. In recent years, refactoring has been studied in different contexts, in particular the refactoring of models representing other aspects of software systems such as design, architecture and deployment.

The need to refactor systems on a larger scale arises from changing business requirements. Examples include moves from monolithic products to product lines, system integration, or the need to improve some of the "ilities" of systems such as maintainability, security or scalability. While the refactoring of systems at the large scale is difficult, it is a common belief amongst software engineers that the pareto principle, also known as the 80-20 rule, applies: a few targeted actions can have an over-proportional impact.

The main question we would like to answer is, can the pareto principle apply at all? If the answer to this question is no, even with very generous assumptions, then this would be a very important result with significant consequences for when refactoring can be profitably used. If the answer is yes, then, due to our assumptions, that would not necessarily mean efficient refactoring of large scale systems would always be possible, but it would at least provide support for pursuing that goal. Our approach is to create a mathematical model of the systems we would like to refactor, and examine whether small changes to the model will have large impacts on the overall quality of the design.

As a motivating example, consider the program depicted in figure 1. The design of this program can be considered as a graph, the so-called dependency graph (DG). The vertices in this graph are types, and the edges are relationships between these types. This particular program consists of four classes `A`,`B`,`C` and `D` and three name spaces `package1`, `package2` and `package3`. It contains several antipatterns [6] that represent design problems:

1. A circular dependency between the packages 1,2 and 3, caused by the path $A \rightarrow_{extends} B \rightarrow_{uses} C \rightarrow_{uses} A$
2. A circular dependency between the packages 2 and 3, caused by the path $B \rightarrow_{uses} C \rightarrow_{uses} D$
3. A subtype knowledge pattern where a type references its own subtype, caused by the edges $A \rightarrow_{extends} B$ and $B \rightarrow_{uses} C \rightarrow_{uses} A$

All three antipattern instances can be removed from the graph with the removal of the single edge $B \rightarrow_{uses} C$. An algorithm for finding this edge is simple: for each edge, record the number of occurrences in *all* instances of *each* antipattern, then remove one of the the edges with the *highest* score. This method would assign the highest score of three only to the edge $B \rightarrow_{uses} C$. All other edges participate in only one or two pattern instances.

The edge $B \rightarrow_{uses} C$ indicates a good starting point for *architectural refactoring*: changing the structure of the system without changing its external behaviour. A refactoring that gives rise to a new system that is modelled by the dependency graph $DG \setminus (B \rightarrow_{uses} C)$ would be a *high-impact refactoring* in the following sense: this particular

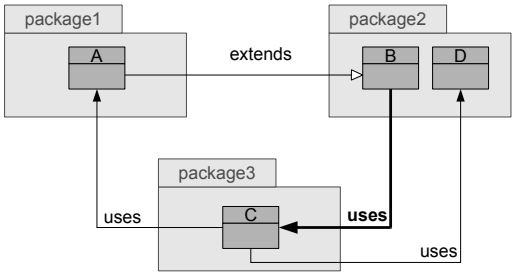

Figure 1: Simple program DG

refactoring removes significantly more (in this case: all) antipattern instances from the model, and can therefore be considered more valuable than alternative refactorings. Such a refactoring would have the highest impact on the overall quality of the system, measured by the number of architectural antipattern instances.

In general, there is no guarantee that such a refactoring can be achieved. However, the fact that a small change in the model has a large impact on the overall quality of the design, in this case, suggests that the pareto principle could apply here. Our aim is to discover whether or not the pareto principle applies in general.

One particular class of architectural refactorings we are interested in is modularisation, and in particular the refactoring of monolithic Java programs into dynamic component models such as OSGi [1] and its clones and extensions. This is a very timely issue, as some of the most complex systems including the Java Development Kit itself [2], and the leading commercial application servers WebLogic and WebSphere [17], are currently being modularised. In our previous work [8], we assessed the scope of the problem by investigating a set of antipatterns that hinder modularisation using an OSGi-style framework. The work presented here has grown out of this approach. We aim to generalise from our previous work to develop a generic approach for using sets of antipatterns to compute refactoring opportunities. The opportunities identified correspond to operations which are applied to a model representing the system. After application, the antipattern analysis is repeated in order to assess whether or not certain characteristics of the system have improved. It turns out that, using this approach, we can compute candidates for high impact refactorings.

The rest of the paper is organised as follows. In Section 2 we review related work. We continue in Section 3 with a short introduction to the framework we have developed for describing antipatterns, and the algorithmic tools used to detect these antipatterns. In Section 4 we motivate our choice of a particular set of antipatterns that hinder modularisation and discuss the algorithm used to detect potential refactoring opportunities. We then describe the organisation of an experiment used to validate our approach, where we analyse a large corpus of open-source Java programs [25] for instances of four antipatterns representing modularisation problems. An analysis of the results of our experiment is presented in Section 5. A discussion of open questions related to our work concludes this contribution.

2 Related Work

2.1 Antipattern and Motif Detection

In our work, we propose to use sets of antipatterns as starting points for architectural refactoring. These patterns can be viewed as the equivalent of smells [10] that are used as starting points for code-level refactorings. While early work on smells and antipatterns has focused on the analysis of source code, many of these concepts can also be applied to software architecture [18]. Research into code-level antipattern and smell detection has resulted in a set of robust tools that are widely used in the software engineering community, including PMD [7] based on source code analysis, and FindBugs [16] based on byte code analysis. A closely related area is the detection of design patterns [13]. Several solutions have been proposed to formalise design patterns in a platform-independent manner. A good overview is given in [33].

Garcia et al. describe a set of architectural smells [14] using a format similar to the original Gang of Four pattern language [13]. These smells are somewhat different from our patterns. The definitions given by the authors in [14] do not seem to be precise enough for tool-supported detection.

Our approach is based on the detection of antipatterns in the dependency graph extracted from a program. The use of dependency graphs as a basis for program analysis has been investigated by several authors (e.g. [19, 4]).

Patterns in graphs can be formalised as *motifs*. Detection of graph motifs has been widely studied in bioinformatics, and there is a large body of recent work in this area. The concept has also been proposed in the context of complex networks (e.g., Milo et al. [21]). The motifs used in both of these areas are simpler than those that we propose, in that we do not only consider local sets of vertices directly connected by edges, but also sets of vertices indirectly connected by paths.

We have investigated in previous work [9] the potential of the Girvan-Newman clustering algorithm [15] to detect refactoring opportunities in dependency graphs.

2.2 Refactoring

Architectural refactorings were first discussed by Beck and Fowler (“big refactorings”) [10], and then discussed by Roock and Lippert (“large refactorings”) [18]. Their work defines the framework for our contribution: starting with the detection of architectural smells by means of antipatterns (for example, cycles) or metrics in architectural models, systems are modified to improve their characteristics while maintaining their behaviour. Large scale refactorings can be broken down (decomposed into smaller refactorings). Our approach fits well into this framework; we compute a sequence of base refactorings that can be performed step by step, using the dependency graph as the architectural model.

Our work is related to the use of graph transformations and graph grammars [27], an area that has been applied in many areas of software engineering such as model transformations. The manipulations of the graphs we are interested in are simple: we only remove single edges. This does not justify the use of the full formalism of a graph grammar calculus. In work by Mens et al. graph transformations are directly used to detect refactorings [20]. There, the focus is on code-level refactoring and the detection and management of dependencies between those refactorings.

Simon et al. try to formalise the notion of smells [30]. The authors use metrics for this purpose, while we use patterns. Tsanatalis and Chatzigeorgiou have identified opportunities to apply the “move method” refactoring [36]. Their proposed algorithm is based on the Jaccard metric between feature sets and preconditions for the respective refactoring. Their aim is to remove only one particular smell (feature envy) from programs. Seng at al. use a genetic algorithm to detect code-level refactorings [29]. Their work is also restricted to the “move method” refactoring. O’Keeffe and O’Cinneide represent object-oriented design as a search problem in the space of alternative designs [22]. They use several search algorithms to

search this space for designs which improve the existing design with respect to a set of given metrics. The refactorings used to traverse the search space are all inheritance-related (extract and collapse hierarchies, move feature up and down the hierarchy) and, therefore, not very expressive.

Bourqun and Keller present a high-impact refactoring case study [5]. They first define the layered architecture of the program to be refactored, and then use Sotograph to detect violations of this architecture. They focus refactoring activities on packages associated with those violations, and validate their approach using violation counts and code metrics. They present their approach as a case study using an enterprise Java application developed by a Swiss telecommunication company. The approach discussed here can be seen as a generalisation of Bourqun and Keller's work [5]: the architecture violations can be expressed using patterns, and the algorithm to compute the artefacts to be refactored from the violations can be recast in our edge-scoring idiom. While our general approach supports and encourages the use of project-specific patterns derived from system architecture, we use a set of general, project-independent patterns for the empirical study.

3 Methodology

3.1 Motifs

As stated earlier, our approach to detect high-impact refactoring opportunities is based on the detection of antipatterns in the *program dependency graph* (DG). In the following paragraphs, we formally define this graph and related concepts.

A dependency graph $DG = (V,E)$ consists of a set of vertices, V, representing types (classes, interfaces and other types used in the programming language), and a set of directed edges, E, representing relationships between those types. Both vertices and edges are labelled to provide further information. Vertices have labels providing the name, the name space, the container (library), the abstractness (true or false) and the kind (interface, class, enumeration or annotation) of the respective type. Edges have a type label indicating whether the relationship is an extends, implements or uses relationship.

We formalise architectural antipatterns as network *motifs* in the dependency graph. Given a dependency graph, a motif can be defined as follows:

A motif $m = (VR, PR, C_V, C_P)$ consists of four finite sets: vertex roles (VR) and path roles (PR), vertex constraints C_V and path constraints C_P. If n is the cardinality of VR, a vertex constraint $c_V \in C_V$ is defined as an n-ary relation between vertices, $c_V \subseteq \times_{i=1..n} V$. If n is the cardinality of PR, a path constraint is $c_P \in C_P$ is defined as an n-ary relation between sequences of edges ($SEQ(E)$), $c_P \subseteq \times_{i=1..n} SEQ(E)$. Intuitively, constraints restrict the sets of possible vertex and path assignments. While vertex constraints are always defined with respect to vertex labels, there are three different types of path constraints:

1. Source and target constraints restricting, respectively, the start and end vertices of a path.
2. Cardinality constraints restricting the length of a path, usually defined using restrictions on the minimum and the maximum length of a path.
3. Constraints defined with respect to edge labels. These constraints have to be satisfied for all edges within a path.

A *binding* is a pair of functions $\langle inst_V, inst_P \rangle$, where $inst_V : VR \to V$ and $inst_P : PR \to SEQ(E)$. A binding associates vertex roles with vertices and path roles with sequences of edges. A *motif instance* is a binding such that the constraints are fulfilled, i.e. the following two conditions must be true:

- $(inst_V(vr_1), .., inst_V(vr_n))) \in c_V$ for all vertex constraints $c_V \in C_V$
- $(inst_P(pr_1), .., inst_P(pr_n))) \in c_P$ for all path constraints $c_P \in C_P$

3.2 Motif Definition and Detection

The detection of motif instances in non-trivial dependency graphs is complex. The worst-case time complexity for the type of motif search that we do is $O(n^k)$, where n and k are the number of vertices in the dependency graph and the number of roles in the motif, respectively. This worst-case time complexity is a consequence of the NP-hardness of the subgraph isomorphism problem, which is essentially the problem that we must solve each time we successfully find an instance of a query motif in a dependency graph. Note that the algorithm that we use to detect motif instances returns all possible bindings of vertex roles, but for each such binding only one selected binding for path roles. Formally, we consider only classes of instances $(inst_v, inst_p)$ modulo $(instance_p)$. This means that two instances are considered as being equal if and only if they have the same vertex bindings.

To detect motifs in the dependency graph we use the GUERY[1] tool. The tool represents dependency graphs in memory, and employs an effective solver to instantiate motifs. The solver used takes full advantage of multi-core processors, and uses various optimisation techniques. It is scalable enough to find motifs in large programs with vertex counts of up to 50000 and edge counts of up to 200000. This kind of scalability is required to analyse real world programs, such as the runtime library of the Java Development Kit, consisting of 17253 vertices and 173911 edges.

Listing 1 shows a motif definition. This motif has two vertex roles $VR = \{$ `type`, `supertype`$\}$ and two path roles $PR = \{$`inherits`, `uses`$\}$. The paths roles have source and target constraints defined by the `from` and `to` attributes in the `connectedby` elements, and edge constraints defined in the expressions in line 4. The edge constraints state that all edges in paths instantiating the `inherits` role must be `extends` or `inherits` relationships, and that all edges in paths instantiating the `uses` role must be `uses` relationships. The length of the paths is not constrained in this example, but the language would support this through the `minLength` and `maxLength` attributes defined for the `connectedBy` element. The default values are 1 for `minLength` and -1, representing unbound, for `maxLength`.

3.3 Edge Scoring

Motif detection in dependency graphs can be used to assess the quality of architecture and design of systems. The classical example is the detection of circular dependencies between packages, modules and types that has been widely discussed [32, 24]. In general terms, we aim to use motifs to formalise antipatterns and smells and thus facilitate detection of design problems. For a single motif, the number of separate motif instances in a dependency graph can be very large. However, edges can simultaneously participate in many instances.

This raises the question of whether, for a given set of motifs representing antipatterns, there are some edges

[1] `http://code.google.com/p/queryframework/`

```
1 motif stk
2 select type,supertype
3 connected by inherits(type>supertype) and uses(supertype>type)
4 where "uses.type=='uses'" and "inherits.type=='extends' || inherits.type=='implements'"
```

Listing 1: The Subtype knowledge motif (STK)

which participate in large numbers of overlapping motif instances. It also raises the question of whether such "high-scoring" edges participate in instances arising from more than one of the motifs in the set. If so, refactorings of the system resulting in the removal of those edges from the dependency graph would be an effective way to improve the overall quality of the architecture and design of the underlying system.

In general, given a dependency graph $DG = (V,E)$, a motif $m = (VR, PR, C_V, C_P)$ and a set of motif instances $I(m) = \{(inst_V^i, inst_P^i)\}$ of m in DG, we define a *scoring function* as a function associating edge-instance pairs with natural numbers, $score : E \times I(m) \to \mathbb{N}$. We also require that a positive score is only assigned if the edge actually occurs in one of the paths instantiating a path role in the motif:

$$\forall i : score(e, (inst_V^i, inst_P^i)) > 0 \Rightarrow \exists pr \in PR : e \in inst_P^i(pr).$$

For a given set of motifs $M = \{m_j\}$ with sets of instances $\{I(m_j)\}$ of M in DG, the overall score of an edge with respect to M is defined as the sum of all scores for each instance of each motif:

$$score_M(e) := \sum_{m \in M} \sum_{inst \in I(m)} score(e, inst).$$

The simplest scoring function is the function that just scores each occurrence of an edge in a motif instance as 1. We call this the *default scoring function*. Given a dependency graph, a set of motifs representing antipatterns and a scoring function, we can define the following generic algorithm to detect edges in the dependency graph that may be associated with high impact refactorings of the underlying system:

1. Compute all instances for all motifs.
2. Compute the scores for all edges.
3. Sort the edges according to their scores.
4. Remove some edges with the highest scores from the graph.
5. Recompute all instances for all motifs and compare this with the initial number to validate the effect of edge removals.

Note that this algorithm has several variation points that affect its outcome:

1. The set of motifs used.
2. The scoring function used.
3. If only one edge is to be removed, the selection function that selects this edge from the set of edges with the highest scores.

Depending on the decisions made for these variation points, the effects of edge removal will be different. However, the existence of these variation points supports the customisation of this algorithm in order to adapt it to project-specific settings. For instance, domain-specific antipatterns and scoring functions can be used to represent weighted constraints penalising dependencies between certain classes or packages.

The selection of the edge from the set of edges with high scores can also take into account the difficulty of performing the actual refactoring on the underlying system that would result in the removal of this edge from the dependency graph .

4 Case Study: Detecting High-Impact Refactorings to Improve System Modularity

To demonstrate the use of our generic algorithm, we present a case study that is based on a particular set of antipatterns representing barriers to modularisation. The presence of instances of these antipatterns in dependency graphs implies that packages (name spaces) are difficult to separate (poor *name space separability*), in particular due to the existence of circular dependencies, and that implementation types are difficult to separate from specification types (poor *interface separability*). Both forms of separability are needed in modern dynamic component models such as OSGi, and in this sense the presence of these motifs represents barriers to modularity. For more details, the reader is referred to Dietrich et al. [8].

4.1 Motif Set

4.1.1 Overview

We use the following four antipatterns that represent design problems in general, and barriers to the modularisation in particular:

1. Abstraction Without Decoupling (AWD)
2. Subtype knowledge (STK)
3. Degenerated inheritance (DEGINH)
4. Cycles between name spaces (CD)

These antipatterns can easily be formalised into graph motifs. We discuss each of these motifs in the following subsection. For a more detailed discussion, the reader is referred to Dietrich et al. [8]. We use a simple visual syntax to represent antipatterns. Vertex roles are represented as boxes. Path roles are represented by arrows connecting boxes. These connections are labelled with either *uses* (uses relationships) or *inherits* (extends or implements relationships). They are also labelled with a number range describing the minimum and maximum length of paths, with "𝕄" representing unbound ("many"). If vertex roles have property constraints, these constraints are written within the box in guillemets.

4.1.2 Abstraction Without Decoupling (AWD)

The Abstraction Without Decoupling (AWD) pattern describes a situation where a client uses a service represented as an abstract type, and also a concrete implementation of this service, represented as a non-abstract type extending or implementing the abstract type. This makes it hard to replace the service implementation and to dynamically reconfigure or upgrade systems. To do this, the client code must be updated. The client couples service description and service implementation together.

Techniques such as dependency injection [12] could be used to break instances of this pattern. Fowler discusses

how patterns can be used to avoid AWD [11]. This includes the use of the Separated Interface and the Plugin design patterns. The visual representation of this pattern is shown in figure 2.

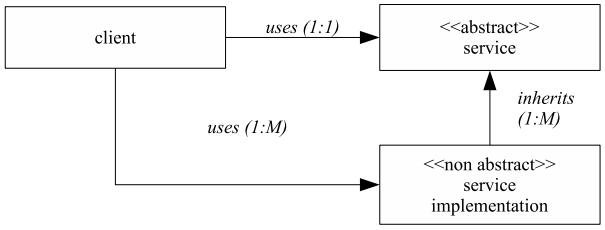

Figure 2: AWD

4.1.3 Subtype Knowledge (STK)

In this antipattern [26], types have `uses` relationships to their subtypes. The presence of STK instances compromises separability of sub- and supertypes. In particular, it implies that there are circular dependencies between the name spaces containing sub- and supertype. Instability in the (generally less abstract) subtype will cause instability in the supertype, and the supertype cannot be used and understood without its subtype. The visual representation of this pattern is shown in figure 3, the definition is given in listing 1.

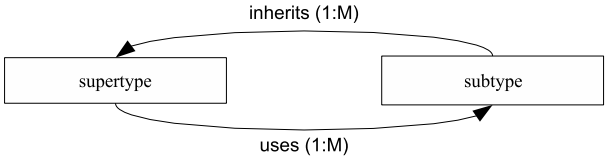

Figure 3: STK

4.1.4 Degenerated Inheritance (DEGINH)

Degenerated inheritance [28, 31], also known as diamond, repeated or fork-join inheritance, means that there are multiple inheritance paths connecting subtypes with supertypes. For languages with single inheritance between classes such as Java, this is caused by multiple interface inheritance. The presence of instances of DEGINH makes it particularly difficult to separate sub- and superclasses.

The visual representation of this pattern is shown in figure 4.

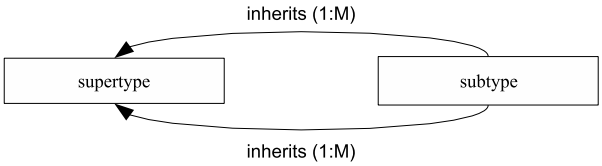

Figure 4: DEGINH

4.1.5 Cycles between Name Spaces (CD)

Dependency cycles between name spaces (CD) is a special instance of cycles between modules [32]. This antipattern implies that the participating name spaces cannot be deployed and maintained separately. In particular, if these name spaces were deployed in several runtime modules (jars), this would create a circular dependency between

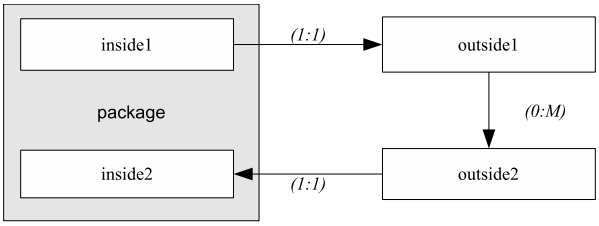

Figure 5: CD

those jars. This antipattern is stronger than the usual circular dependency between name spaces A and B which requires that there be two paths, one connecting A to B and the other connecting B to A. CD requires the existence of one path from A, through B, back into A. The weaker form of circular dependency can sometimes be removed by simply splitting the names spaces involved. CD is more difficult to remove as the path must be broken through refactoring.

The visual representation of this pattern is shown in figure 5. Note that the cardinality constraint for the path connecting `outside1` and `outside2` is `[0,M]`. This means that the path can have a length of 0. In this case, the antipattern instance has a triangular shape and the two outside roles are instantiated by the same vertex.

4.2 Scoring Functions

In this experiment, we have used the default scoring function that increases the score by one for each edge encountered in any path instantiating any path role in each instance for each of the four motifs.

4.3 Data Set

For the validation of our approach we have used the Qualitas Corpus, version 20090202 [34]. For many programs, the corpus contains multiple versions of the same program, sometimes with only minor differences between those versions. We have therefore decided to keep only one version of each program in the data set. We decided to use the latest version available. There are two programs in this set that do not have instances for any antipattern in the set used: `exoportal-v1.0.2.jar` and `jmeter-2.3.jar`. We have removed those two programs from the data set. We have also removed `eclipse_SDK-3.3.2-win32` and `jext-5.0` — these programs already use a plugin-based modularisation model (e.g., through the Eclipse extension registry and the Equinox OSGi container) and therefore many of the antipatterns we are interested in will not be present. Finally, we have removed the Java Runtime Environment (JRE, `jre- 1.5.0_14-linux-i586`) — it turns out that our tools are not yet scalable enough to do a full analysis due to the size to the JRE. However, we have done a partial analysis of the JRE, the results are discussed below. This has given us the final set of the 95 programs.

The dependency graphs extracted from the programs in the corpus differ widely in size. The largest graph, extracted from `azureus-3.1.1.0`, has 6444 vertices and 35392 edges. The smallest graph, extracted from `ivatagroupware-0.11.3`, has 17 vertices and 22 edges. The average number of vertices in graphs extracted from corpus programs is 660, the average number of edges 3409.

4.4 Graph Preparation

The dependency graphs can be extracted from different sources, such as byte code and source code of programs

written in different programming languages. We have used the dependency finder library [35] to extract dependency graphs from Java byte code. Dependency graphs built from byte code are slightly different from graphs built from source code. For instance, relationships defined by the use of generic types are missing due to erasure by the Java compiler[2]. We do not see this as a problem as the focus of our investigation is to find refactoring opportunities to improve the runtime characteristics of deployed systems.

Graphs are represented as instances of the JUNG [23] type `edu.uci.ics.jung.graph.DirectedGraph`. This has caused some issues related to the repeatability of results. The GUERY solver we have used to detect motif instances returns all possible bindings of vertex roles, but for each such binding, only one selected binding for path roles. It is possible to override this behaviour and compute all possible path role assignments as well. However, we have found that this is only feasible for very small motifs or graphs and that the combinatorial explosion in the number of possible paths makes a scalable implementation impossible for graphs of a realistic size. Formally, we consider only classes of instances $(inst_v, inst_p)$ modulo $(instance_p)$, i.e., two instances are considered equivalent if they have the same vertex bindings.

The problem arising from this is that, when the computation is repeated, in some cases different path role bindings are computed for the same binding of vertex roles, since the query engine traverses outgoing/incoming paths in a different order. This is caused by the internal indexing of incoming/outgoing edges in the JUNG API that uses hashing. For this reason, we have modified the JUNG API and to represent outgoing and incoming edges as lists with predictable order. We have also added a method to set a comparator to sort incoming/outgoing edges for all vertices in the graph. In the experiment presented here we have used a comparator that sorts edges according to their betweenness score [15]. If the betweenness value is the same for two edges, they are sorted by the fully qualified names of source and target vertices. The objective of using this particular comparator function is to make it more likely that edges that are more active in the overall topology of the graph will be bound to path roles and thereby gain an increase in score. Thus, this idea should promote the identification of edges with high global impact.

5 Results

5.1 Impact of Edge Removal

Figure 6 shows the decline of numbers of antipattern instances after removing the edges with the highest score. Data were obtained using the simple scoring function $score_1$.

The number of instances is scaled to 100%. Initially, all programs have 100% of their antipattern instances. The values on the x-axis represent the number of edge removal iterations performed. In each iteration, one edge with the highest score is removed, and then the antipattern counts and the edge scores are recomputed. If there is more than one edge with the same highest score, these edges are sorted according to the fully qualified names of source and target vertices, and the first edge is removed. The main reason for using this selection function is to make the experiment repeatable, and to remove only one edge at a time in order to observe the effects of single edge removals corresponding to *atomic* architectural refactorings.

The chart is a boxplot. The dots in the middle represent the medians in the distribution, and the bold bars around the median represent areas containing 50% of the population.

It is remarkable that the median falls below 50% after only 8 iterations. This means that for half of the programs from the data set, only 8 or fewer edge removals are necessary to remove half of the pattern instances from the model. This suggests that applying refactorings corresponding to these edge removals to the actual programs would have a similarly dramatic effect. The argument is purely statistical: this method works well for most, but not all, programs — the chart shows several outliers.

[2]Generic type information is only stored using the signature attribute in Java byte code, this information can be used for reflection, but is not used by the Java runtime when loading, linking and initialising classes and objects.

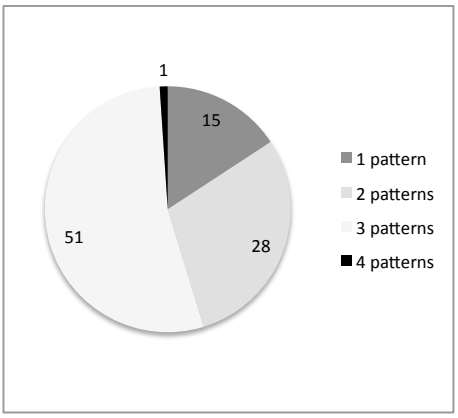

Figure 7: Number of antipatterns instantiated by edge with highest score

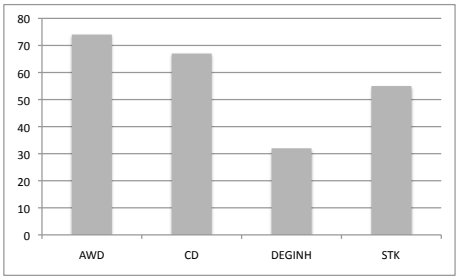

Figure 8: Number of programs with highest scored edge instantiating a given antipattern

5.2 Pattern Distribution

The question arises whether high scores are caused by single antipatterns, or whether there is an "overlay effect" — edges have high scores because they participate in instances of more than one antipattern. Analysis shows that the latter is the case. For the 95 programs analysed, there are only 15 programs for which the edge with the highest score only participates in instances of a single antipattern. For the majority of programs (51/95), this edge participates in instances of three different antipatterns (figure 7).

Figure 8 shows participation by pattern. For all four patterns we find a significant number of programs where the highest scored edge participates in instances of the respective pattern. This is an indication that we picked a favourable set of patterns in the sense that the combination of these patterns yields synergy effects when detecting edges corresponding to possible high impact refactorings.

The next question we have investigated is whether the simultaneous analysis of multiple patterns yields better results than using one pattern at a time. To answer this question we have created a scoring function for each single pattern. This scoring function increases the score of an edge by one whenever the edge participates in an instance of the respective pattern, and by zero otherwise. That means that

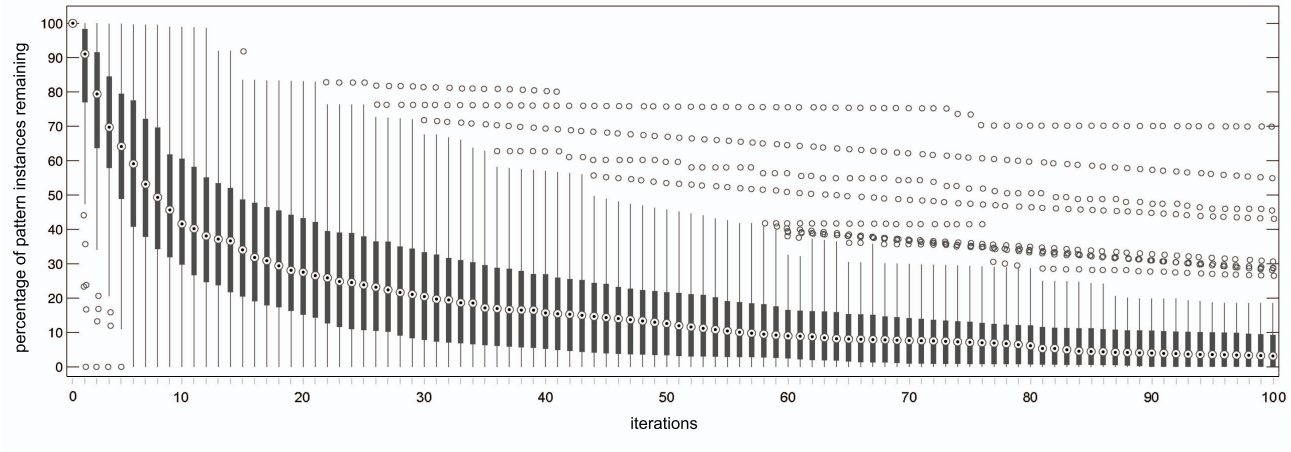

Figure 6: Number of antipattern instances by number of refactorings performed

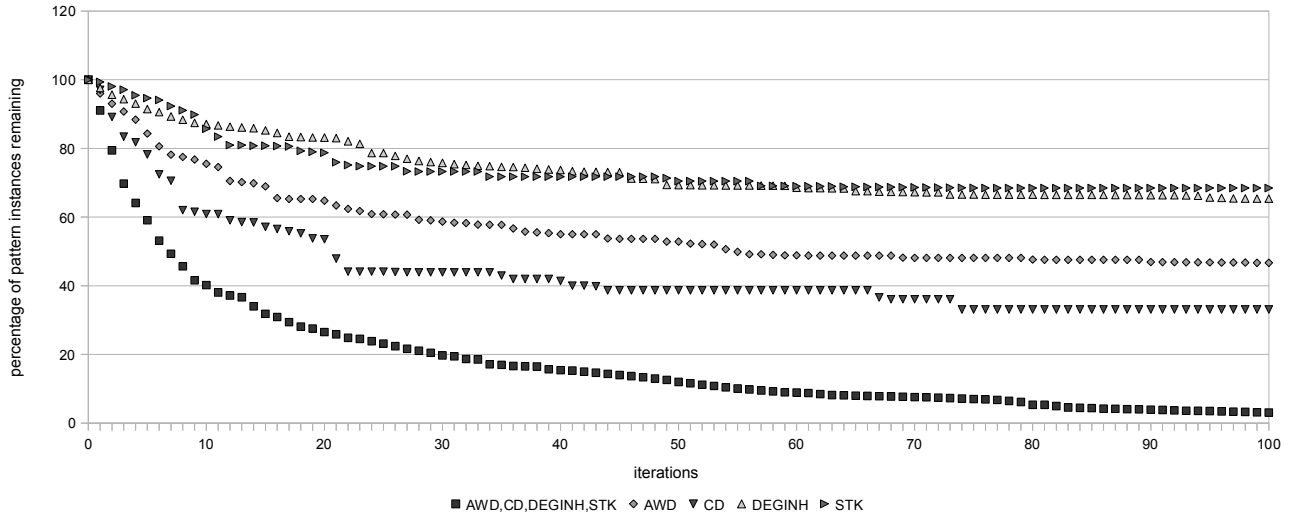

Figure 9: Comparison of antipattern instance removal using analysis based on single and combined antipatterns

only this one pattern is used to compute the edge to be removed. We have then measured how the total number of pattern instances found for all patterns drops. Figure 9 shows the results for the first 50 iterations — the values are the means of pattern instances remaining after the respective number of edge removals. This figure shows that by using the combined strategy (the data series with the label "AWD,CD,DEGINH,STK") better results can be obtained. The curves representing the single pattern scoring strategies flatten out — indicating that all instances of the respective patterns are eventually removed, but that a significant number of instances of other patterns remain.

5.3 Dependency on Program Size

The question arises of whether the trend depends on program size. To address this issue, we have divided the set of programs into two new sets, consisting of relatively small and relatively large programs. The difference between the larger and smaller halves of the programs is relatively small (to remove 50% of the initial number of antipatterns, the mean of edges to be removed is 8 for larger programs and 6 for smaller programs). That indicates that our approach may be particularly useful to guide the refactoring of larger programs: the effort (to apply refactorings corresponding to the removal of edges in the dependency graph) only increases slowly with program size.

This is surprising, since the number of antipattern instances increases significantly with program size. The average number of instances in the smaller (larger) half is 335 (15423) for AWD, 2748 (140254) for CD, 153 (1605) for DEGINH and 27 (356) for STK, respectively. The large numbers for some antipatterns are caused by the combinatorial explosion of the number of paths defined by references (edges in the dependency graph). It turns out that many of these antipattern instances can be considered as variations of a much smaller number of "significant" instances [8].

Figure 10 shows the number of iterations that are necessary to remove 50% of antipattern instances, depending on program size measured by the number of vertices in the dependency graph. This chart shows that, for most programs, only few edge removals are necessary to achieve the goal. However, there are a few programs that require a very large number of edge removals.

One of these programs is the spring framework, a well known dependency injection container. Here, the fact that there are no high-impact refactorings can be seen as an indicator of good design — those refactoring opportunities have already been detected and the respective refactorings have been performed by moving dependencies into configuration files. Those dependencies are not part of the dependency graph.

5.4 Scalability

We have found that for average size programs our implementation of the algorithm scales very well. Anal-

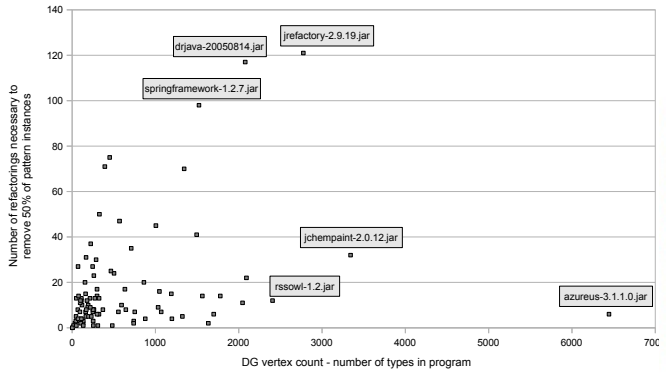

Figure 10: Number of refactorings necessary to remove 50% of antipattern instances

program	vertices	edges	iter. 1	iter. 10	iter. 100
azureus-3.1.1.0	6444	35392	717718	275599	73288
jruby-1.0.1.jar	2093	11016	90713	89101	31226
derby-10.1.1.0	1198	11174	27882	8468	3898
xerces-2.8.0.jar	878	4782	3448	1533	738
ant-1.70	752	3326	6553	1838	659
lucene-1.4.3.jar	231	930	479	456	430
junit-4.5.jar	188	648	439	432	426

Table 1: Performance Data (times in ms)

ysis typically finishes within a few seconds or minutes. We have used a MacBook Pro with a 2.8 GHz Intel Core 2 Duo with 4GHz of memory. We have used the Java(TM) SE Runtime Environment (JRE build 1.6.0_17-b04-248-10M3025) with the Java HotSpot(TM) 64-Bit Server VM, and a multithreaded solver running on two threads for analysis. For the largest programs in the data set, `azureus-3.1.1.0`, the time needed to finish the initial iteration was about 12min (717718ms). Table 1 shows performance data for some selected, widely-used programs. The time to run an iteration decreases significantly as more edges are removed, in particular for larger programs. As more and more edges are removed, the dependency graph becomes more and more disconnected and the solver has to iterate over fewer sets of paths.

We have also tried to analyse the JRE itself, consisting of the three libraries `rt.jar`, `jce.jar` and `jsse.jar`. The dependency graph extracted from these libraries is large, consisting of 17253 vertices and 173911 edges. The algorithm can still be applied, but computing the first iteration alone took approximately 4.5 hours. Note that the solver algorithm takes full advantage of multi-core processors and can be easily distributed on grids. We therefore think that it is still possible to use our approach for exceptionally large programs by utilising distributed computing environments such as Amazon's EC2.

5.5 Classifying Edge Removals

An edge in the dependency graph represents a dependency from a source type to a target type in the program. Dependencies arise in a number of ways from the source code. The edge removal we have performed corresponds to an actual refactoring that has to be applied to the original program. We expect that a template based approach can be used for this purpose, based on the kind of dependency. For this purpose, we have classified the edges according to the source code pattern detected that has caused this dependency.

We classify the edges into eight categories as follows:

1. **Variable Declaration (VD):** The target type is used to declare a field or a temporary variable.
2. **Constructor Invocation (CI):** A target type constructor is invoked with the keyword new.
3. **Static Member Invocation (SMI):** Invocation of a static member (method or field) of the target type.
4. **Method Return Type (MR):** The target type is used as a method return type.
5. **Method Parameter Type (MP):** The target type is used as a parameter type in the method signature.
6. **Method Exception Type (ME):** The target type is used as an exception type with throws keyword.
7. **Superclass (SC):** The target type is used as a supertype by using extends keyword.
8. **Interface (IN):** The target type is used as an interface by using the implements keyword.

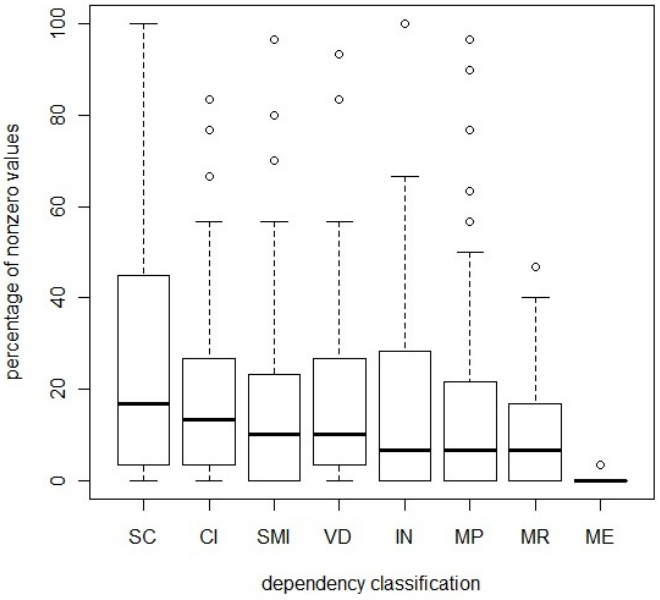

Figure 11: Dependency classification results

We have analyzed a high-scoring subset of the removed edges in order to classify them according to the dependencies giving rise to those edges. The edges in the dependency graph contain one of the three different labels i.e. `uses`, `extends` and `implements`. A `uses` edge can be involved in multiple dependency categories. This is because a source type can use the target type in a number of above-mentioned ways.

Figure 11 shows the distribution of the percentage of non-zero values in every dependency category. We analysed all 95 programs and in every program the first 30 removed edges, with a few exceptions where the total number of edges removed was less than 30. We scaled the non-zero values of every category to 100% with respect to the number of edges analysed. For example, if, in the top 30 relationships (edges) SMI is encountered 15 times, then, for the given program the usage of SMI would be 50%. We can see from figure 11 that most of the dependencies are caused by inheritance relationships, while the lowest number of dependencies comes from the method exception types.

In order to see how often we have multiple reference types, we calculated the participation of the first removed edge for every program in different dependency categories. We found that 41% of the programs have edges

that participate in multiple dependency categories. This suggests that refactoring of these programs will be more challenging.

5.6 Implementing Edge Removals

The algorithm presented here operates only on *models* (the dependency graphs), not *programs*. The refactorings recommended by the algorithm are operations to remove artefacts from the model rather than from the program itself. The question arises as to how these refactorings can be implemented so that the actual program can be transformed.

In general, implementing refactorings of the program corresponding to edge removals in the dependency graph is a difficult problem as it requires a very detailed understanding of the design of the respective program. There are situations when nobody has this understanding, for instance if projects evolve, many people are involved, participants change, and design is neither documented nor planned. However, there are some edge removals that can easily be interpreted. The first edge tagged for removal from the Java Runtime Environment (OpenJDK version 1.6.0 b_14 for Windows) is the reference from `java.lang.Object` to `java.lang.Class`. This is caused by the fact that all classes reference `Object` and `Class` has outgoing edges as well. It is probably very difficult and not necessarily desirable to refactor the JRE in order to get rid of this particular edge. However, the second and third targeted edges are references from AWT to Swing: `java.awt.Component` uses `javax.swing.JComponent` and `java.awt.Container` uses `javax.swing.JInternalFrame`. These references point to a real problem. While it is understandable that Swing references the older AWT toolkit, it is hard to see why AWT has to reference the newer Swing toolkit. This makes it impossible to deploy AWT applications without the more resource-demanding Swing. There are several use cases for this: AWT uses the more efficient platform widget toolkits, and AWT applets are at least partially compatible with Microsoft Internet Explorer.

It is interesting to see that those two references are not present in the alternative Apache Harmony [3] implementation of the Java development kit (version 6.0, r917296-snapshot). This implies that it is really possible to "break" the respective edges in the model without compromising the behavioural integrity of the respective system. In this case, a comprehensive set of test suites is used to ensure compatibility between Apache Harmony and the OpenJDK, which is the reference implementation of the Java Development kit.

Another interesting example is `azureus-3.1.1.0`, the largest program in the data set. It has a large initial number of pattern instances in the model (846147) that suddenly drops to 271166 (32.05% of the initial count) after only 5 edge removals. The first five edges removed are:

The first edge is a reference from the plugin manager interface `org.gudy.azureus2.plugins.PluginManager` that orchestrates the application to its concrete subclass `org.gudy.azureus2.pluginsimpl.local.PluginManagerImpl`. There are five references in the compilation unit, all sharing the same structure: static method calls are delegated to the implementation class. These dependencies can be easily removed through the use of a service registry: the plugin manager can obtain the name of the implementation class from the registry, load this class and invoke the respective method using reflection. The next four edges are similar, and can be removed using the same strategy.

We believe that it may not be possible to automate, or even always implement, the refactorings recommended by the proposed algorithm. Actual refactoring is about manipulating program source code or models close to source code (such as abstract syntax trees), and is therefore programming language dependent. However, we can observe certain patterns causing dependencies between classes which occur in all mainstream programming languages. These are the categories discussed in section 5.5. For some of these categories, there are common refactoring techniques that can be applied. These include the use of design patterns and dynamic programming techniques that have been developed to avoid or reduce dependencies, such as factories, proxies, service registries and dependency injection containers. These techniques are particularly useful to remove dependencies between client classes and service implementation classes. Examples include general-purpose frameworks such as the Spring framework, Guice, the `java.util.ServiceLoader` class, OSGi and its extensions such as declarative services and the Eclipse extension registry, and specialist solutions such as the JDBC driver manager and the JAXP Document Builder Factory.

Often, referenced types can be replaced by their supertypes if those supertypes define the part of the interface of the type that is being referenced by the client. This is possible in all modern mainstream programming languages that use dynamic method lookup. For instance, if a (Java) method references a method with a parameter type `java.util.ArrayList`, the parameter type can usually safely be replaced by `java.util.List`.

There are limitations to this approach that make it unlikely that this can be completely automated. In particular, the use of dynamic programming techniques such as reflection makes it sometimes difficult to predict the behavioural changes caused by these transformations. This implies that firstly, refactoring activities must be safeguarded by verification techniques, such as post refactoring testing; and secondly, that it is an empirical question to find out to what extent these refactorings can be automated in real world systems.

A comprehensive study to determine to what extent refactorings corresponding to our edge removal operations can be automated is subject to further research.

6 Conclusion

We have presented an algorithm that can be used to detect potential high-impact refactorings based on the participation of references in sets of antipatterns that are seen as design flaws. We have validated our approach by using a set of four antipatterns that are known to compromise modularisation of programs, applied to a set of 95 programs. The main result presented in this paper is that, in most cases, the algorithm will be able to detect high-impact refactoring opportunities.

We have demonstrated that the respective refactorings can be applied without changing the program behaviour, for some examples, using the largest and most complex programs in our data set. We did not discuss an actual algorithm to automatically perform refactorings corresponding to edge removal. This question has to be explored in future investigations. We believe that the classification of dependency types in section 5.5 is a good starting point for such a study. We have realistic expectations here — while we expect that in many cases the refactorings are easy to describe and can be automated (for instance, by introducing dependency injection or replacing concrete type references by references to interfaces), this will not always be the case. The research challenge is to define refactorings that can be automated in restricted situations where certain prerequisites are fulfilled, and then to find the *weakest prerequisites*. The difficulty of performing dependency-breaking refactorings represents a cost that could be taken into account when defining the scoring functions used in our approach.

Investigating alternative combinations of antipattern sets and scoring functions is an interesting and promising field. There is no evidence that the combination we have used is optimal. Unfortunately, the validation for each set of parameters against the corpus is computationally expensive and takes several hours to complete, this makes a trial and error approach difficult.

There are several interesting theoretical aspects related to this work that can be explored further. For instance, how does the pattern density found in the dependency graphs of typical Java programs compare to that for randomised graphs? For the simpler notion of motifs used in bio-informatics, a study of this kind has been done by Milo et al. to detect the Z-score [21].

References

[1] OSGi™— the dynamic module system for Java. http://www.osgi.org/.

[2] Project jigsaw. http://openjdk.java.net/projects/jigsaw/.

[3] Apache Harmony, 2010. http://harmony.apache.org/.

[4] D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Transactions on Software Engineering*, 31(2):137–149, 2005.

[5] F. Bourqun and R. K. Keller. High-impact refactoring based on architecture violations. In *Proceedings CSMR '07*.

[6] W. J. Brown, R. C. Malveau, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, March 1998.

[7] T. Copeland. *PMD Applied*. Centennial Books, 2005.

[8] J. Dietrich, C. McCartin, E. Temero, and S. M. A. Shah. Barriers to Modularity — An empirical study to assess the potential for modularisation of Java programs. In *Proceedings QoSA'10*, 2010.

[9] J. Dietrich, V. Yakovlev, C. McCartin, G. Jenson, and M. Duchrow. Cluster analysis of Java dependency graphs. In *Proceedings SoftVis'08*, pages 91–94, 2008.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[11] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[12] M. Fowler. Inversion of control containers and the dependency injection pattern, 2004. http://martinfowler.com/articles/injection.html.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Boston, MA, USA, 1995.

[14] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *Proceedings CSMR'09*, pages 255–258, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[15] M. Girvan and M. E. Newman. Community structure in social and biological networks. *Proc Natl Acad Sci U S A*, 99(12):7821–7826, June 2002.

[16] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Proceedings OOPSLA '04*, pages 132–136, New York, NY, USA, 2004. ACM.

[17] C. Humble. IBM, BEA and JBoss adopting OSGi. http://www.infoq.com/news/2008/02/osgi_jee.

[18] M. Lippert and S. Roock. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, 2006.

[19] R. Martin. OO Design Quality Metrics: An Analysis of Dependencies. `http://www.objectmentor.com/resources/articles/oodmetrc.pdf`, May 1994.

[20] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 6(3):269–285, September 2007.

[21] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, October 2002.

[22] M. O'Keeffe and M. O'Cinneide. Search-based software maintenance. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 249–260, Washington, DC, USA, 2006. IEEE Computer Society.

[23] J. O'Madadhain, D. Fisher, S. White, and Y.-B. Boey. The JUNG (Java universal network/graph) framework. Technical Report UCI-ICS 03-17, University of California, Irvine, 2003.

[24] D. L. Parnas. Designing software for ease of extension and contraction. In *Proceedings ICSE '78*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press.

[25] Qualitas Research Group. Qualitas corpus version version 20090202, 2009.

[26] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[27] G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[28] M. Sakkinen. Disciplined inheritance. In *Proceedings ECOOP'89*, pages 39–56, 1989.

[29] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1909–1916, New York, NY, USA, 2006. ACM.

[30] F. Simon, F. Steinbrueckner, and C. Lewerentz. Metrics based refactoring. In *Proceedings CSMR'01*, page 30. IEEE Computer Society, 2001.

[31] G. B. Singh. Single versus multiple inheritance in object oriented programming. *SIGPLAN OOPS Mess.*, 5(1):34–43, 1994.

[32] W. Stevens, G. Myers, and L. Constantine. Structured design. pages 205–232, 1979.

[33] T. Taibi, editor. *Design Patterns Formalization Techniques*. Idea Group Inc., Hershey, USA, 2007.

[34] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010.

[35] J. Tessier. Dependency finder. http://depfind.sourceforge.net/.

[36] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 99(RapidPosts):347–367, 2009.