On the Detection of High-Impact Refactoring Opportunities in Programs

Jens Dietrich, Catherine McCartin, Ewan Tempero and Syed M. Ali Shah

Abstract—We present a novel approach to detect refactoring opportunities by measuring the participation of references between types in instances of patterns representing design flaws. This technique is validated using an experiment where we analyse a set of 95 open-source Java programs for instances of four patterns representing modularisation problems. It turns out that our algorithm can detect high impact refactorings opportunities — a small number of references such that the removal of those references removes the majority of patterns from the program.

1 INTRODUCTION

Software systems are subject to change. However, change 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 90’s, 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 80-20 rule, applies: a few targeted actions can have an overproportional impact. In our work, we set out to provide some statistical evidence for this.

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 patterns that represent design problems:

1) A circular dependency between the packages 1,2 and 3, caused by the path $A \rightarrow_{\text{extends}} B \rightarrow_{\text{uses}} C \rightarrow_{\text{uses}} A$

2) A circular dependency between the packages 2 and 3, caused by the path $B \rightarrow_{\text{uses}} C \rightarrow_{\text{uses}} D$

3) A subtype knowledge pattern where a type references its own subtype, caused by the edges $A \rightarrow_{\text{extends}} B$ and $B \rightarrow_{\text{uses}} C \rightarrow_{\text{uses}} A$

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

One particular class of large-scale refactorings we are interested in is modularisation, and in particular the refactoring of monolithic Java programs into dynamic component models such as OSGi 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 [21], are currently being modularised. In our previous work [10], we have assessed the scope of the problem by investigating a set of patterns 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 patterns to compute refactoring opportunities. These opportunities are then applied to a model representing the system, and the pattern 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 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 patterns, and the algorithmic tools used to detect these patterns. In Section 4 we motivate our choice of a particular set of patterns that hinder modularisation and discuss the algorithm used to detect refactoring opportunities. We then describe the setup of an experiment used to validate our approach, where we analyse a large corpus of open-source Java programs [31] for instances of four patterns 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 Pattern and Motif Detection

In our work, we propose to use sets of patterns as starting points for large-scale, architectural refactorings. These patterns can be viewed as the equivalent of smells [12] that are used as starting points for code-level refactorings. Closely related to code smells are antipatterns [22]. While early work on smells and patterns has focused on the analysis of source code, many of these concepts can also be applied to software architecture [23]. Research into code-level antipatterns and smell detection has resulted in a set of robust tools that are widely used in the software engineering community, including PMD [9] based on source code analysis, and FindBugs [20] based on byte code analysis. A closely related area is the detection of design patterns [15]. Several solutions have been proposed to formalise design patterns in a platform-independent manner, a good overview is given in [40].

Garcia et al. describe a set of architectural smells [16] using a format similar to the original gang of four pattern language [15]. These smells are somewhat different from our patterns, the definitions given by the authors do not seem to be precise enough for tool-supported detection.

Our approach is based on the detection of patterns 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., [21], [5]).

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. [27]). 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 [11] the potential of the Girvan-Newman clustering algorithm [18] to detect refactoring opportunities in dependency graphs. Here, we use a variant of this algorithm to prepare the graphs for analysis.

2.2 Tools

JDepend [1] is a popular tool that extracts dependencies between classes and packages from Java byte code and calculates a set of metrics [24] that can be used to quantify the quality of the system design. It also detects circular dependencies, but no other patterns we are interested in.

The existing tool most closely aligned to the purposes of our analysis is CrocoPat [5]. CrocoPat is an effective system to query and manipulate relations that can be extracted from programs. All graph motifs used in our analysis can be described by queries expressible in CrocoPat. In particular, Beyer at al. give definitions for queries describing two of the motifs that we are interested in, DEGINH and STK [5]. CrocoPat is based on Binary Decision Diagrams (BDDs), and known for its excellent performance. CrocoPat’s language (RML) is a full programming language with syntax elements such as conditionals and loops, based on first order logic. CrocoPat queries return facts for a certain predicate symbol. Paths cannot be directly represented in CrocoPat, however, reasoning about paths in CrocoPat is supported through the higher-order transitive closure predicate TC. In our analysis, we require the ability to arbitrarily constrain the lengths of paths when matching instances of our motifs. For instance, this feature is needed when querying for circular dependencies between packages, and when we are interested in the actual dependency paths, not just in the packages that are the start and end points of these paths.

There are a number of commercial architectural analysis tools available, such as the Software Tomograph
(Sotograph) [6], Structure101 [19] and Lattix [35]. The main limitations of these tools are that they are user-interface centric and therefore not suitable for scripting. Scripting is, however, necessary to automate the analysis of a large program corpus. Also, these tools do not support the definition of custom queries.

None of the existing analysis tools fulfilled all of our requirements. In particular, we required a tool that made available a solid implementation open enough to allow integration of domain-specific heuristics, supported scriptability to analyse large sets of programs in batch mode, and had support for path constraints and aggregation conditions.

The static analysis tool that we have developed to fulfill our specific requirements is GQL4JUNG (“graph query language for Jung ”). This tool, and its method of deployment, and experiments evaluating it, are described in detail by Dietrich et al [10]. One experiment suggests that GQL4JUNG is faster than CrocoPat for computing instances of motifs, and requires significantly less memory.

2.3 Corpus Analysis

In recent years corpora-based empirical studies have become more prevalent, largely due to the fairly widespread availability of open-source software. These studies have had a variety of goals. For example, open-source systems have been used for metric validation (e.g. [8]), for studying the appearance of power law distributions [4], and for studying how certain features of a language are used in practice, such as use of multiple dispatch [28], inheritance [32] and unused design decisions [41].

There have also been a number of studies attempting to characterise common idioms in software, such as micro-patterns [17] and more recently nano-patterns [37]. Our work differs in that we target idioms that have a negative impact on software quality. Other studies with similar goals to ours include [25] which attempts to characterise the cyclic structures in dependency graphs.

2.4 Refactoring

Large scale refactorings were first discussed by Beck and Fowler [12] (“big refactorings”), the term architectural refactoring was introduced by Roock and Lippert [23]. Their work defines the framework for our contribution: starting with the detection of architectural smells by means of patterns (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 [33], 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 [26]. There, the focus is on code-level refactoring and the detection and management of dependencies between those refactorings.

Work by Simon et al. try formalise the notion of smells [35]. The authors use metrics for this purpose, while we use patterns.

Bourquin and Keller present a high-impact refactoring case study [7]. 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 Bourquin and Keller’s work [7]: 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 the detection of patterns in the program dependency graph (DG). In the following paragraphs, we define this graph and related concepts in abstract terms.

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 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 patterns as graph motifs. Given a DG, a motif can be defined as follows:

A motif \( m = (VR, PR, CV, CP) \) consists of four finite sets: vertex- (VR) and path roles (PR), vertex constraints \( CV \) and path constraints \( CP \). If \( n \) is the cardinality of VR, a vertex constraint \( CV \subseteq x_{i=1}^{n} V \). If \( n \) is the cardinality of PR, a path constraint \( CP \subseteq x_{i=1}^{n} P \) is...
defined as a $n$-ary predicate between tuples of sequences of $E$, $c_P \subseteq \times_{i=1...n} SEQ(E)$. 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 the start and the end vertex of a path.
2) Cardinality constraints restricting the length of the 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 $(\text{inst}_V, \text{inst}_P)$, where $\text{inst}_V : V \rightarrow V$ and $\text{inst}_P : PR \rightarrow SEQ(E)$. I.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:

- $(\text{inst}_V(v_1), ..., \text{inst}_V(v_n)) \in c_V$ for all vertex constraints $c_V \in C_V$
- $(\text{inst}_P(p_1), ..., \text{inst}_P(p_n)) \in c_P$ for all path constraints $c_P \in C_P$

### 3.2 Motif Definition and Detection

The detection of motif instances in non-trivial DGs 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 $(\text{inst}_V, \text{inst}_P)$ modulo $(\text{instance}_P)$, i.e., two instances are considered similar iff they have the same vertex bindings.

To detect motifs in the DG we use the GQL4JUNG ("graph query language for Jung") tool. The tool represents DGs using the JUNG API and employs an effective solver to instantiate motifs. The motifs are represented in a combination of XML and the MVEL2 expression language. The XML part is used to define the topology of the motif, while the MVEL2 expressions are used to express constraints on labels. MVEL2 is particularly suitable for this as it supports an easy, Java style syntax and features runtime compilation into Java byte code. The solver uses 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 type and supertype defined in lines 2 and 3, respectively. The two path roles inherits and uses define paths connecting the vertices instantiating the vertex roles. 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 lines 5 and 8. 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.

Note that the pattern definitions presented here do not contain groupBy clauses used by Dietrich et al. In general terms, we refer to non-aggregated instances further modulo user-defined equivalent relations, the algorithm used here uses all instances.

### 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. 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 DG can be very large. However, edges can simultaneously participate in many instances. This raises the question of whether, for a given set of motifs, there are some edges 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 resulting in the removal of those edges would be an effective way to improve the overall quality of the architecture and design of the underlying problem.

In general, given a DG $(V,E)$, a motif $m = (VR, PR, CV, CP)$ and a set of motif instances $I(m)$ of $m$ in $(V,E)$, we define a scoring function as a function associating instances-edge pairs with natural numbers, $score : E \times I(m) \rightarrow \mathbb{N}$. We also require that a positive score is only assigned if the edge


2. In [10], we have referred to non-aggregated instances as variants.
for each instance of each motif:

\[
\forall i: \text{score}(e, (\text{inst}_V, \text{inst}_P)) > 0 \Rightarrow \exists pr \in PR : e \in \text{inst}_P(pr).
\]

For a given set of motifs \( \{m_j\} \) 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:

\[
\text{score}_M(e) := \sum_{m \in M} \sum_{\text{inst}(m) \in I(m)} \text{score}(e, \text{inst}).
\]

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 DG, a set of motifs and a scoring function, we can define the following generic algorithm to detect high impact refactorings of the DG:

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 patterns 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 removing this edge from the actual program.

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 motifs representing barriers to modularisation. The presence of instances of these motifs in DGs implies that packages (name spaces) are difficult to separate (name space separability), in particular due to the existence of circular dependencies, and that implementation types are difficult to separate from specification types (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 [10].

4.1 **Motif Set**

4.1.1 **Overview**

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

1) Abstraction Without Decoupling (AWD) — affects interface separability
2) Subtype knowledge (STK) — affects interface separability
3) Degenerated inheritance (DEGINH) — affects interface separability
4) Cycles between name spaces (CD) — affects name space separability

These patterns can easily be formalised into graph motifs. We discuss each of these motifs in the following subsection, for a more detailed discussion that reader is referred to [10]. We will use a simple visual syntax to represent motifs. 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 “\(*\)” representing unbound (“many”). If type roles have property constraints, these constraints are written within the box in guillemets.

Listing 1: The Subtype knowledge motif (STK)

1 <motif name="stk">  
2 <select role="type"/>  
3 <select role="supertype"/>  
4 <connectedBy role="inherits" from="type" to="supertype"/>  
5 <constraint>inherits.type==‘extends’ || inherits.type==‘implements’</constraint>  
6 </connectedBy>  
7 <connectedBy role="uses" from="supertype" to="type"/>  
8 <constraint>uses.type==‘uses’</constraint>  
9 </connectedBy>  
10 </motif>
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, but 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 \[14\] could be used to break instances of this pattern. Fowler discusses how patterns can be used to avoid AWD\[13\]. This includes the use of the Separated Interface and the Plugin patterns.

The visual representation of this pattern is shown in figure 2 and the definition is given in listing 2.

![Fig. 2: AWD](image)

4.1.3 Subtype Knowledge (STK)

In this pattern \[32\], 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 and the definition is given in listing 3.

![Fig. 3: STK](image)

4.1.4 Degenerated Inheritance (DEGINH)

Degenerated inheritance \[34\], \[38\], 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 like 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 and the definition is given in listing 3.

![Fig. 4: DEGINH](image)

4.1.5 Cycles between Name Spaces (CD)

Dependency cycles between name spaces is a special instance of cycles between modules \[39\]. These imply that these name spaces cannot be deployed and maintained separately. In particular, if those name spaces were deployed in several runtime modules (jars), this would create a circular dependency between those jars. This pattern is stronger than the usual dependency of circular dependencies between name spaces A and B that requires that there must be two paths, one connecting A and B and the other one connecting B and A. CD requires the existence of one path from A through B back into A. This is more difficult to remove as this path must be broken through refactoring. On the other hand, the weaker form of circular dependency can sometimes be removed by simple splitting the names spaces involved.

The visual representation of this pattern is shown in figure 5 and the definition is given in listing 4.

![Fig. 5: CD](image)

4.2 Scoring Functions

We have used two different scoring functions. The first scoring function \(\text{score}_1\) is the default function that increases the score by one for each edges encountered in any path instantiating any path role in each instance for each of the four motifs. The second scoring function used \(\text{score}_2\) assigned higher values to edges that are seen as more critical. This is done as follows:

1) For AWD, assign a higher score to edges participating in the paths connecting the client with the service implementation.
2) For STK, assign a higher score to edges participating in the uses relationship connecting the super types with their own subtypes.
3) For CD, assign a higher score to edges representing transitions between packages.

To define a scoring function, a Java class implementing the interface ScoringFunction has to be implemented. Listings 5 and 6 show the definitions of score1 and score2, respectively.

4.3 Data Set

For the validation of our approach we have used the Qualitas Corpus, version 20090202 [31]. 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 pattern in the pattern set used: exoportal-v1.0.2.jar and jmeter-2.3.jar. We
int getEdgeScore(Motif motif, String pathRole, Path path, Edge e) {
    String name = motif.getName();
    if ("awd".equals(name)) {
        if ("implementation_dependency".equals(pathRole)) return 2; else return 1;
    } else if ("stk".equals(name)) {
        if ("uses".equals(pathRole)) return 2; else return 1;
    } else if ("cd".equals(name)) {
        String namespace1 = e.getStart().getNamespace();
        String namespace2 = e.getEnd().getNamespace();
        if (namespace1.equals(namespace2)) return 1; else return 2;
    }
    return 1;
}

Listing 6: Advanced Scoring Function (score^2)

have removed those two programs from the data set. We 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 patterns we are interested in will not be present. Finally, we 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 did a partial analysis of the JRE, the results are discussed below. This gave us the final set of the 95 programs listed in table 1.

The DGs 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 DGs 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 [43] to extract DGs from Java byte code. DGs 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.

Graphs are represented as instances of the JUNG type edu.uci.ics.jung.graph.DirectedGraph. This has caused some issues with the repeatability of results. The GQL4JUNG 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

| ant-1.7.0  | antlr-2.7.7 |
| aoi-2.5.1  | argouml-0.24 |
| aspectj-1.0.6 | axiom-1.0-M2 |
| azureus-3.1.1.0 | c_jdbc-2.0.2 |
| checkstyle-4.3 | cobertura-1.9 |
| colt-1.2.0  | columba-1.0 |
| compiere-250d | derby-10.1.0 |
| displaytag-1.1 | drawsvg-1.2.9 |
| drjava-20050814 | emma-2.0.3312 |
| findbugs-1.0.0 | fitjava-1.1 |
| fitlibraryforfitnesse-20050923 | freecol-0.7.4 |
| freecs-1.2.20060130 | galileo-1.8.0 |
| ganttproject-1.11.1 | git-2.2.3rc3 |
| heritrix-1.8.0 | hibernate-3.3.1-ga |
| hsqldb-1.8.0.4 | htmlunit-1.8 |
| informa-0.6.5 | ireport-0.5.2 |
| itext-1.4.5 | ivalagroupware-0.11.3 |
| j-5.0.1 | james-2.2.0 |
| jasmin-0.10 | jasperreports-1.1.0 |
| javacc-3.2 | jchempaint-2.0.12 |
| jedit-4.3pre14 | jena-2.5.5 |
| jfin_date_math-1.0.0 | jfreechart-1.0.1 |
| graph-5.9.2.1 | graphpad-5.10.0.2 |
| graph-0.7.3 | groups-2.6.2 |
| jhotdraw-5.3.0 | jmoney-0.44 |
| jogglayer-1.1.4s | jparse-0.96 |
| jpl-1.0.2 | jrat-0.6 |
| jrefactory-2.9.19 | jruby-1.0.1 |
| jswikis-2.2.33 | jxce-0.4 beta |
| jtopen-4.9 | jung-1.7.6 |
| junit-4.5 | log4j-1.2.13 |
| lucene-1.4.3 | marauroa-2.5 |
| megamark-2005.10.11 | mvnforum-1.0-ga |
| myfaces_core-1.2.0 | nakedobjects-3.0.1 |
| neokhtmll-0.9.5 | openjms-0.7.7-alpha-3 |
| oscache-2.3-full | picococntainer-1.3 |
| pmd-3.3 | poi-2.5.1 |
| pooola-1.1-060227 | proguard-3.6 |
| quartz-1.5.2 | quickserver-1.4.7 |
| quotl-0.6-a-5 | roller-2.1.1-incubating |
| rssowl-1.2 | sablecc-3.1 |
| sandmark-3.4 | springframework-1.2.7 |
| squirrel_sql-2.4 |.struts-1.2.9 |
| sunflow-0.07.2 | tomatc-5.5.17 |
| trove-1.1b5 | velocity-1.5 |
| webmail-0.7.10 | weka-3.5.8 |
| xalan-j-2.7.0 | xerces-2.8.0 |

TABLE 1: Data set
is only feasible for very small motifs or graphs and that the combinatorial explosion on the number of possible paths makes a scalable implementation impossible for graphs of a realistic size. Formally, we consider only classes of instances \((\text{inst}_v, \text{inst}_p)\) modulo \((\text{instance}_p)\), i.e., two instances are considered similar iff they have the same vertex bindings.

The problem arising from this is that, whenever the computation is repeated, in some cases different path role bindings are computed for the same binding of vertex roles as 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. The respective containers used by JUNG use hashing, and the order in which edges are returned is unpredictable. For this reason, we have modified the JUNG API and added references to outgoing and incoming edges directly to the vertex type used. We have also added a method to set a comparator to statically sort incoming/outgoing edges for all vertices in the graph. Statically, here, means that this has to be done only once for each vertex before querying starts, in a preparation step. This is sufficient to make the results repeatable, and adds very little overhead to the computation.

In the experiment presented here we have used a comparator that sorts edges according to their betweenness score [18]. If the betweenness value is the same for two edges, they are sorted by the fully qualified names of start and target edge. 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 removal of edges with high global impact.

5 Results

5.1 Impact of Edge Removal

Figure 6 shows the decline of numbers of pattern instances after removing the edges with the highest score. Data were obtained using the simple scoring function \(\text{score}_1\).

The number of instances is scaled to 100%. Initially, all programs have 100% of their pattern 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 pattern 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 start and end vertex, 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 representing atomic architectural refactorings.

3. The DEGINH motif is using this feature to compute all possible paths connecting sub- and superclass, the attribute used for this purpose is \texttt{computeAll}, see listing 3, line 7.

![Fig. 7: Number of patterns instantiated by edge with highest score](image)

Fig. 7: Number of patterns instantiated by edge with highest score

![Fig. 8: Number of programs with highest scored edge instantiating a given pattern](image)

Fig. 8: Number of programs with highest scored edge instantiating a given pattern

The chart is a boxplot produced with MatLab. 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 refactorings removing edges are necessary to remove half of the pattern instances. The argument is purely statistical: this method works well for most, but not all, programs — the chart shows several outliers.

5.2 Pattern Distribution

The question arises of whether high scores are caused by single patterns, or whether there is an “overlay effect” — edges have high scores because they participate in instances of more than one pattern. 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 pattern. For the majority of programs (51/95), this edge participates in instances of three different patterns (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

Fig. 6: Number of pattern instances by number of refactorings performed

Fig. 9: Comparison of pattern instance removal using analysis based on single and combined patterns

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 zero otherwise. That means that 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 patterns scoring strategies flatten out — indicating that all instances of the respective patterns are eventually removed, but a significant number of instances of other patterns remains.

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. Program size is measured using the numbers of vertices in the dependency graph. The results are shown in figure 10 for the set of smaller programs (program 1-47 in data set ordered by size), and in figure 11 for the set of larger programs (program 48-95). These figures show that while the trend is essentially the same, more edge removals are necessary for larger programs to remove the same percentage of patterns.

Figure 12 shows the number of iterations that are necessary to remove 50% of pattern instances, depending on program size measured by the number of vertices in the DG. This chart shows that, for most programs,
Fig. 10: Number of motif instances by number of refactorings performed for bottom 50% of programs by vertex count

Fig. 11: Number of motif instances by number of refactorings performed for top 50% of programs by vertex count

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

5.4 Variations

The experiments presented in the previous sections are based on the use of the simple scoring function $score_1$. We have also investigated whether the use of alternative scoring functions can improve the results, i.e. are there alternatives that compute edge removals leading to a steeper decline in the numbers of pattern instances. We have experimented with the alternative scoring function $score_2$. Figure [3] shows the results of the comparison for the first 50 iterations. Surprisingly, the results obtained with the simple scoring function are slightly better. Note that this does not imply that $score_1$ is optimal.

5.5 Scalability

We have found that for average size programs our implementation of the algorithm scales very well. Analysis 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 (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 2 shows some 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 graphs become more and more disconnected and the solver has to iterate over fewer sets of paths.

We have also tried to analyse the Java Runtime Environment itself, consisting of the three libraries rt.jar, jce.jar and jsse.jar. The DG 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 elastic cloud (EC2).

5.6 Classifying Edge Removals
An edge in the DG 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 is a simulation of 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.

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 DG contain one of the three different labels i.e. uses, extends and implements. An 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 14 shows the distribution of percentage of non-zero values in every dependency category. We have 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 the figure that most of the dependencies are caused by inheritance relationships, while the least number of dependencies comes from the method exception types.
Table 2: Performance Data

<table>
<thead>
<tr>
<th>program</th>
<th>vertices</th>
<th>edges</th>
<th>iter. 1</th>
<th>iter. 10</th>
<th>iter. 20</th>
<th>iter. 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>azureus-3.1.1.0</td>
<td>6445</td>
<td>35392</td>
<td>717718ms</td>
<td>275599ms</td>
<td>232516ms</td>
<td>73288ms</td>
</tr>
<tr>
<td>jruby-1.0.1.jar</td>
<td>2093</td>
<td>11016</td>
<td>90713ms</td>
<td>89101ms</td>
<td>79921ms</td>
<td>31226ms</td>
</tr>
<tr>
<td>derby-10.1.1.0</td>
<td>1198</td>
<td>11174</td>
<td>27882ms</td>
<td>84688ms</td>
<td>65258ms</td>
<td>38988ms</td>
</tr>
<tr>
<td>xerces-2.8.0.jar</td>
<td>875</td>
<td>4782</td>
<td>3448ms</td>
<td>1533ms</td>
<td>1289ms</td>
<td>738ms</td>
</tr>
<tr>
<td>ant-1.70</td>
<td>752</td>
<td>3326</td>
<td>6553ms</td>
<td>1838ms</td>
<td>1093ms</td>
<td>699ms</td>
</tr>
<tr>
<td>lucene-1.4.3.jar</td>
<td>231</td>
<td>930</td>
<td>479ms</td>
<td>456ms</td>
<td>445ms</td>
<td>430ms</td>
</tr>
<tr>
<td>junit-4.5.jar</td>
<td>188</td>
<td>648</td>
<td>439ms</td>
<td>432ms</td>
<td>430ms</td>
<td>426ms</td>
</tr>
</tbody>
</table>

In order to see how often we have multiple reference types, we calculated the participation of the first removed edge of every program in different dependency categories. We found 41% of the programs have edges that participate in multiple dependency categories. This reflects that refactoring of these programs will be more challenging.

5.7 Interpreting Edge Removals

Interpreting edge removals 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 (version 1.6.0 for Windows) is the reference from java.lang.Object to java.lang.Class. This is probably 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:


These references point to a real problem. While it is understandable that Swing references the older AWT toolkit, it is hard to see while 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.

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 (846147) that suddenly drops to 271166 (32.05% of the initial count) after only 5 edge removals. The first five edges removed are:

1) org.gudy.azureus2.plugins.PluginManager uses org.gudy.azureus2.pluginsimpl.local.PluginManagerImpl
2) com.aelitis.azureus.core.AzureusCoreFactory uses com.aelitis.azureus.core.impl.AzureusCoreImpl
3) org.gudy.azureus2.core3.config.impl.ConfigurationManager uses org.gudy.azureus2.core3.config.impl.ConfigurationChecker
4) com.aelitis.azureus.core.AzureusCore uses com.aelitis.azureus.core.nat.NATTraverser
5) org.gudy.azureus2.core3.disk.impl.piecemapper.DMPieceMapEntry uses org.gudy.azureus2.core3.disk.impl.DiskManagerFileInfoImpl

The first reference is a reference from the plugin manager interface that orchestrates the application to a concrete subclass extending it. The specification class references a concrete implementation class. This dependency could be easily removed through dependency injection. The second reference is a reference between an abstract factory and a concrete product. Again, this violates established design principles to use abstraction to separate components, and could be easily removed using dependency injection.

6 Conclusion

We have presented an algorithm that can be used to detect refactorings based on the participation of references in sets of patterns that are seen as design flaws. We have validated our approach by using a set of four patterns that are known to compromise modularisation of programs, and have applied it 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 did not discuss how the actual refactorings to break dependencies can be performed. This question has to be explored in future investigations. 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 breaking dependencies represents a cost that
could be taken into account when defining the scoring functions used in our approach.

Investigating alternative combinations of pattern 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 that can be explored further, for instance, how the pattern density found in the DG’s of typical Java programs compares 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.

7 Acknowledgements

This project has been supported by funding from the New Zealand Foundation for Research, Science and Technology (FRST) for the Software Process and Product Improvement (SPPI) project.

References