Available online at www.sciencedirect.com

ScienceDirect

NasBASE

ELSEVIER

North American Search Based Software Engineering Symposium 00 (2015) 1–16

www.elsevier.com/locate/procedia

Search-Based Refactoring for Layered Architecture Repair: An Initial Investigation

Sue Hickey and Mel Ó Cinnéide

School of Computer Science and Informatics, University College Dublin, Ireland

Abstract

Layered architectures are popular because of the flexibility and maintainability benefits they provide. However, experience shows that as a project grows, architecture erosion can occur resulting in the deterioration of the original layering intended by the system architect. To address this problem, we propose the application of search-based refactoring to fully automate the detection and correction of access violations in a layered architecture. We extend the search-based refactoring framework, Code-Imp, to provide the metrics and refactorings necessary for architecture violation repair and perform a case study on the open source project DSpace, which was previously identified as suffering from architectural erosion. A number of hill-climbing and simulated annealing variants are applied in the investigations. We found that while simulated annealing could remove 78% of the layering violations, it resulted in an incoherent design. First-ascent hill climbing was able to produce a 65% reduction in layering violations and yielded a functionally coherent design where the majority of the refactorings performed moved items to locations that were appropriate based on the architectural documentation. From these results we conclude that search-based refactoring is a useful and promising technique for the automated repair of access violations in layered architectures.

Keywords:
search based software engineering, search base refactoring, architecture repair.

1. Introduction

The layered architecture model is popular because of the maintainability and flexibility benefits it provides. In a layered architecture the system is divided into layers whereby classes in each layer can access only classes in their own layer and classes in the layer directly below. These restrictions make the system more maintainable, as changes to one layer will not require changes to all other layers. A layered system is also more flexible because an entire layer can be replaced with an alternative implementation as long as both implementations follow the same interface.

As a system inevitably changes, architecture erosion can occur due to time constraints, poorly understood architecture, a lack of communication or a lack of architecture conformance checking and enforcement. This leads to a documented architecture that no longer matches the actual architecture of the system, which in turn nullifies the benefits provided by the intended architecture and creates inaccurate documentation that can lead to bad design decisions being made. Existing research has explored architecture conformance checking and enforcement to prevent architecture erosion [1, 2, 3, 4, 5, 6, 7] and architecture erosion repair [8, 9, 10, 11], but most of these processes involve a large amount of manual analysis. In this paper

we explore the possibility of fully-automating architecture erosion repair using search based refactoring techniques.

Search based techniques have been applied successfully to many software problems, and this well-established field is called Search Based Software Engineering (SBSE) [12]. There are two prerequisites to formulating a software problem as a search problem: a representation of the problem solution that can be manipulated to form alternative solutions and a fitness function that can judge the quality of a solution in relation to other solutions. Once the representation and fitness function are defined, techniques such as hill climbing or genetic algorithms can be used to find optimal, or near optimal, solutions to the problem. In the case of search based refactoring there have been a number of studies performed (e.g. [13, 14, 15, 16, 17, 18]) that show that search based techniques can be used to find a good set of refactorings to apply to a codebase in order to improve the quality of the codebase in relation to certain metrics.

In this paper we use Code-Imp, a search-based automated refactoring framework, to investigate the application of search based refactoring to the problem of layered architecture erosion repair. Code-Imp was designed and developed by O'Keeffe, Hemati Moghadam and Ó Cinnéide [15, 19, 37] to facilitate their study into improving code reusability, understandability and flexibility using search based refactoring. In this paper we extend Code-Imp to allow it to take in architectural information and to use this information to find layering violations. We also introduce a metric, Illegal Layer Coupling Factor (ILCF), that measures the degree of layering violation in a software system. This metric, along with a layer coupling metric and a layer cohesion metric, is used to guide our search. In our case study we examine DSpace [20], an open source digital repository system that uses a layered architecture, and which we know from previous work to suffer from architecture erosion [21]. We refactor this application using Code-Imp and analyse the resulting refactorings. The results show that many valid refactorings are performed that remove layering violations while conforming to the functional descriptions of the layers found in the architectural documentation. This leads us to conclude that search-based refactoring is a promising technique for automating the repair of layering violations.

Note that layering violations occur for a variety of reasons including, amongst others, inadequate documentation, poor performance, development under time pressure, programmer error or an overly-constraining intended architecture. In some cases it may be decided that certain layering violations should be permitted for the time being, e.g. in a case where adhering to the as intended layered architecture would lead to unacceptably poor performance. Notwithstanding this observation, many violations will be simply undesirable and should be corrected and this is the focus of our work.

The remainder of this paper is organised as follows: In Section 2 we look at some background concepts such as architectural views and the layered architecture model and describe some of the past and current research around architectural conformance checking and architecture repair. Section 3 describes the functionality and architecture of the Code-Imp refactoring framework and how we tailored it for this paper, while the experimental results of applying this approach to the DSpace project is described in Section 4. Finally, in Section 5, we present our final conclusions and plans for future work in this area.

2. Related Work

We explore two areas of related work in this section. In Section 2.1 we look at software architecture, architecture erosion and architecture repair while in Section 2.2 the area of search based software refactoring is dealt with.

2.1. Architectural Models

The architecture of a system is a view of the components of the system and how these components interact, the most popular being the "4 + 1" model introduced by Khruchten in 1995 [22]. As depicted in Figure 1, this model comprises 4 views, namely the logical, process, development and physical architectures, as well as a set of scenarios that shows how the four views work together.

For the purposes of this paper we focus on the *development* architecture. This view captures the structure and interaction rules of a layered architecture. Garlan and Shaw describe a layered architecture as follows:

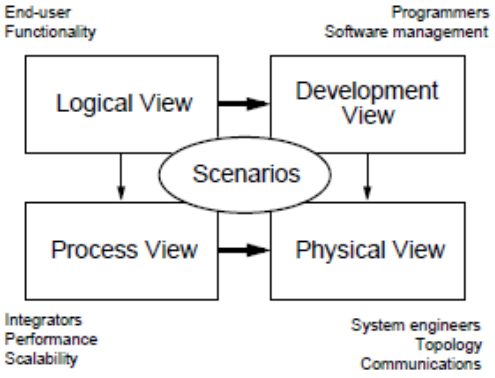


Fig. 1: The "4 + 1" architectural view model (taken from [22]).

"A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below" [23]. This is the basic interaction rule involved in a layered architecture. A class that resides in layer A should access only classes in layer A or in the layer directly below layer A. Bushmann, et al. describe a variation on this strictly layered system in the form of relaxed layered systems [24]. Relaxed layered systems allow each layer to access every layer below it rather than just the layer directly below. This provides additional flexibility but can make the system more difficult to maintain.

2.1.1. Architecture Conformance

Architecture erosion occurs when the implemented architecture of a system diverges from its intended architecture [25]. One way to prevent architecture erosion is to monitor architectural conformance as the system changes, so that any violations that are introduced can be repaired as they occur and gradual architectural drift can be avoided. A number of techniques exist for architectural compliance checking. Reflexion models, introduced by Murphy et al. [1], compare an extracted model of the implemented architecture to the architect's model using a manually defined mapping. The result is a reflexion model that identifies divergences between the actual and the intended architectures. These were later extended to hierarchical reflexion models by Koschke and Simon [2]. Rosik et al. [3] also introduce an extension of reflexion models called inverted reflexion models. In this case, the implemented architecture is derived from the source code and is updated as the code changes. This makes inverted reflexion models particularly suitable for checking architecture compliance as a system evolves. Postma [4] introduces a method that is similar to reflexion models but that uses relationship constraints among modules to verify conformance. The mapping and the architectural constraints are derived from formal models of the intended architecture.

Aldrich et al. [26] created a framework called ArchJava for unifying architectural specification and implementation, thus creating an executable architecture that enables the ArchJava compiler to validate the architectural constraints. Archium [5] and ArchWare [6] provide alternative implementations of a similar concept. Rainbow [7] is a self-adaptation framework that monitors a system against an architectural model and drives automated repairs if constraints are violated.

There is a number of commercially available tools for checking dependencies between modules. For example, Structure101 [27] provides a visual representation of dependencies between modules and SonarJ [28] uses an XML specification of the intended architecture to detect dependency violations. These checks could be used to monitor the relationships between modules and ensure that architectural rules around module interaction are not violated.

2.1.2. Architecture Repair

In the previous section, existing work in architecture conformance checking and techniques for avoiding architecture erosion were presented. In this section the focus is on repairing systems that have already suffered from architecture erosion.

Bourqun and Keller investigated refactorings that have a large impact on the quality of the system's architecture, which they term "high impact refactorings" [8]. The process through which they find these high impact refactorings is predominantly manual. They begin by examining the system to create an architectural model and a set of mappings from packages in the source code to modules in the model. They use the model and the mappings as input to a tool, Sotograph, that returns the architectural violations, which are then examined and addressed using best practices in architectural design. This research provides more of an experience report than a suggested solution or process for solving violations.

Tran and Holt introduce forward and reverse architectural repair [9] and apply this methodology to the repair of two large open source systems [10]. They define forward architectural repair to be the repairing of the concrete architecture to match the conceptual architecture and reverse architecture repair to involve the repairing of the conceptual architecture to match the concrete architecture. In their studies, they were successful in removing many structural anomalies from the architectures of the software systems examined.

Terra et al. [11] present a recommendation system for the repair of architecture violations called ArchFix. It produces refactoring recommendations to be performed on the concrete architecture to make it conform to the intended architecture. The 32 refactoring types used in the system emerged from an investigation into a training system with a large number of architectural violations. The solution to each violation, provided by a system architect, was analysed and generalised to create a set of violations and their associated recommended refactorings. ArchFix was evaluated using two industrial systems and correct refactorings were recommended for 79% of the 828 architecture violations detected.

Constantinou et al. [29] introduce an approach to layer identification that uses machine learning to derive classification rules for layers. These classification rules relate design metric values to architectural layers. Their process involves using Directed Acyclic Graphs (DAGs) to form tentative architectural layers. They then calculate a number of Chidamber and Kemerer metric values on the classes in their layers, which are then discretized using the Minimal Description Length Principle (MDLP) into value ranges for each layer. Layer identification rules are then derived from these calculated value ranges. In experiments on open source projects they found a correlation between the design metric values and the architectural layers in these systems, although rules could not be derived in every case. This research could be extended to architecture repair by categorising layers based on a given architecture, and then applying search based refactoring to remove layering anomalies, as we do in this paper.

In recent work that is closely related to ours, Herold and Mair [30] present an approach that uses an architecture checking mechanism based on first-order logic combined with a metaheuristic search to recommend a set of refactorings that repairs the detected violations. In their experiments with synthetic examples, their approach could remove all the injected dependency violations. When tested on a real application, the percentage of corrected violations fell to 70%, which is similar to what we report from our case study. Their results use only a single refactoring, Move Class, and the remedy for each type of architectural violation is stated explicitly, whereas we rely on a fitness function to direct the search to the remedy. However, what they present is a broad framework that has the potential to be applied to a larger set of architectural violations than what we are examining in this paper.

2.1.3. Component Clustering

In some legacy systems there may be no architectural documentation or the implemented architecture may have eroded to the point where it is not feasible to repair it to match the documented architecture. In such cases it can be useful to derive a new architectural model from the source code. Search based clustering techniques can be used to derive a cohesive set of subsystems and their interactions based on the implemented architecture of the system.

Mancoridis et al. introduced a tool called Bunch for automatic software modularization [31]. It treats the modularization problem as a graph partitioning problem and uses SBSE techniques to address it. A good partitioning is one in which there is a high amount of internal cohesion between the elements inside a partition and a low amount of coupling between elements in different partitions. It was found that Bunch was capable of producing good subsystem decompositions without any knowledge about the design, and some limitations of the approach were addressed in a later work [32].

Mitchell et al. introduced a two step process for reverse engineering the architecture of a system [33]. The first step uses Bunch to find the subsystem hierarchy of the system, while the second step feeds these results into ARIS (Architecture Relation Inference System) along with user-defined subsystem relationship constraints in order to infer relations from the subsystem hierarchy that satisfy the constraints. A case study demonstrated that ARIS could derive valid style relations that could be used for future maintenance.

Harman et al. investigated the robustness of two fitness functions for module clustering [34]. One of these functions was the same fitness function used in Bunch (MQ); the other function was the EVM function defined by Tucker et al. [35]. They found that EVM outperformed MQ on real systems and that MQ could not find a perfect solution when one existed which suggests that there are problems with the MQ function, and the EVM function could be a viable alternative.

Huynh and Cai implement a cluster-based solution to architecture conformance checking [36]. Their approach involves the use of a design structure matrice (DSM) that contains information on the relationships between the various design decisions. The source DSM is clustered using a genetic algorithm with a fitness function based on the edit distance, i.e. the topological difference between the two graphs. The fitness function also imposes penalties for empty mappings, rules that exist in the model DSM but have no mapping in the source DSM. Case studies show that this approach is capable of finding discrepancies between the intended and implemented architectures.

2.2. Search Based Software Refactoring

Search-based refactoring is fully automated refactoring driven by metaheuristic search and guided by software quality metrics, as introduced by O'Keeffe and Ó Cinnéide [37]. Here we provide an overview of some of the relevant research in this area.

O'Keeffe and Ó Cinnéide [15, 16] use search-based refactoring to address the problem of automating design improvement. They used a collection of 12 metrics to measure the improvements achieved when methods are moved between classes, new classes created and associations between classes changed. In later work, similar approaches were used to to improve program testability [38] and program security [39] and to try to refactor one program to more closely match the quality attributes of another [40].

Fatiregun et al. [13] showed how search based transformations could be used to reduce code size and construct amorphous program slices. The work of Kessentini et al. also aims to transform a software system by improving software quality metrics and reducing instances of code smells [17, 18, 41]. The same authors also use optimisation techniques and code development history to guide a search towards an optimal refactoring sequence for minimisation of code smell instances [42].

Ghannem et al. [43] use an Interactive Genetic Algorithm (GA) which interacts with users while allowing feedback to a normal GA. The implemented tool was used to suggest sequences of refactorings which could be applied to models in the form of class diagrams. The extent to which an interactive approach could be applied and validation of the correctness of suggested refactorings were explored and both showed promise. Moghadam and Ó Cinnéide [44] present an approach that refactors a program based on its desired design and source code. Open-source Java was used as an empirical basis and results from the study showed that the original program could be refactored to that design with 90% accuracy.

Seng et al. [14] propose an indirect search-based technique that uses a genetic algorithm over refactoring sequences. In contrast to O'Keeffe and Ó Cinnéide [15], their fitness function is based solely on coupling measures. Both these approaches used a weighted-sum approach to combining separate metrics into a fitness function, which is of practical value but is a questionable operation on ordinal metrics. A solution to the problem of combining ordinal metrics was presented by Harman and Tratt, who introduced the concept of Pareto optimality to search-based refactoring [45]. They used it to combine two metrics into a fitness function and demonstrated that it has several advantages over the weighted-sum approach. Jensen and Cheng [46] use genetic programming to drive a search-based refactoring process that aims to introduce design patterns.

In summary, source-level search-based refactoring has been used for a broad variety of purposes, but to the best of the authors' knowledge it has not yet been applied to the problem of software architecture repair.

3. Extending Code-Imp to Repair Layering Violations

Code-Imp is a fully automated source-to-source refactoring framework developed by the authors and others to investigate the use of automated refactoring to improve software quality as expressed by a metric suite [15, 19]. *Code-Imp* takes Java version 7 source code as input and produces as output a refactored version of the program, a list of applied refactorings and detailed metrics information gathered during the refactoring process. There are three aspects to the refactoring that takes place: the set of refactoring types that can be applied, the fitness function that directs the search, and the type of search technique employed.

Code-Imp supports over twenty refactorings that operate at the detailed design level. The supported refactorings split and merge classes, move methods and fields about the inheritance hierarchy and change accessibility levels of methods and fields. The fitness function employed depends on the quality that is to be optimised. *Code-Imp* is frequently used to improve software design, so in this case the fitness function is a combination of software quality metrics. Over 40 software quality metrics have been implemented in *Code-Imp* and can be combined using either weighted-sum or Pareto optimal approaches. *Code-Imp* supports first and steepest ascent hill-climbing, and simulated annealing to explore the space of alternative designs.

Further details about Code-Imp can be found in our previous work [15, 19]. In this section we detail the aspects that are relevant to the topic of automated correction of layering violations. In section 3.1 we show how Code-Imp is informed of the "as intended" layered architecture of the software application being refactored. In section 3.2 the set of layering metrics employed is discussed and in Section 3.3 the refactorings employed in the case study are described. Finally, in Section 3.4 we describe the search techniques employed in this paper.

3.1. Layer Identification

In order to identify layering violations, we extended Code-Imp to read layering information from a configuration file. The user provides this information in the form of a layer number followed by a list of packages that map to that layer. For example the layering information for the DSpace project, described further in Section 4, is provided in the form:

```
1:org.dspace.storage; 2:org.dspace; 3:org.dspace.app
```

This means that any classes in the `org.dspace.storage` package or any of its subpackages belong in layer 1. Any classes in the `org.dspace` package or any of its subpackages, excluding `org.dspace.storage` and `org.dspace.app`, belong in layer 2. Any classes in the `org.dspace.app` package or its subpackages belong in layer 3. This information is parsed into a map and a utility method is used to identify the layer that any class in the AST belongs to using the class package information and the layering information.

Whenever a class accesses another class, e.g. through calling a method or by accessing a variable in that class, we can determine whether this access is in accordance with the intended layered architecture or not, based on the layer to which each class belongs. An access that is in accordance with the intended layered architecture is termed a *legal* access, otherwise it is an *illegal* access. This legal access check is used extensively in the metrics described in the following subsections.

3.2. Metrics For Measuring Layering Properties

Our aim is to produce a refactored codebase that conforms more closely to a given layered architecture. To this end we require a metric that measures the level of layer violation, or illegal layer coupling, in the system. A natural approach is to select an existing class-based coupling metric and modify this to make it applicable to packages rather than to classes. We thus considered several class-based coupling metrics, namely Coupling Between Objects (CBO), Response For Class (RFC), Message Passing Coupling (MPC) and Coupling Factor (COF). We finally chose to modify the Coupling Factor (COF) metric from the MOOD set of metrics of Abreu et al. [47], partly as it is the only one of these metrics to yield a value in the range $[0, 1]$. The COF metric measures the proportion of actual couplings between classes in a system to the maximum number of possible class couplings in the system.

Mapping the COF metric to the notion of layers is straighforward. We name this new metric *Illegal Layer Coupling Factor* (ILCF). It measures the illegal class couplings in a system (i.e. the number of

classes accessing classes outside of their own layer or the layer directly below their own layer) in proportion to the maximum possible number of illegal class couplings in the system. See Equation 1.

$$ILCF = \frac{actual\ number\ of\ illegal\ couplings}{maximum\ possible\ number\ of\ illegal\ couplings} \quad (1)$$

ILCF yields a normalised value between 0 and 1, where 0 represents a system with no illegal coupling and 1 represents a system where every class is coupled to every other class outside of its legally accessible layers. This normalised value allows us to compare easily the level of illegal layer coupling in systems of different sizes.

Although ILCF is suitable for measuring the extent of layering violation, it is not very effective for guiding a search-based refactoring process that is trying to fix layering violations. This is because in the ILCF perspective, each class can only be coupled to another class once, regardless of how many methods or fields of that class it uses. This means that we will not see an improvement in this metric for refactorings that reduce the amount of coupling between two classes but do not remove it altogether. For example, if class A illegally accesses two methods of class B and we move one of the methods from class B to a class that is legally accessible from class A, this will not produce an improvement in the ILCF metric as class A is still coupled to class B by the call to the second method.

Thus a fitness function using only the ILCF metric would not create any search pressure to move fields and methods that reduce the illegal coupling between classes but do not remove it. For this reason, two additional metrics were added to measure illegal method level coupling and illegal field level coupling. These metrics, *ILCFM* and *ILCFF*, are identical to the ILCF metric except that they measure coupling at the method and field level rather than at the class level.

When faced with a layering violation there may be many potential refactorings that would repair the violation and improve the ILCF metric. Some of these refactorings will be more appropriate than others. We attempt to guard against moving items to inappropriate new locations by including an overall layer coupling metric and a layer cohesion metric in the fitness function. Refactorings that remove violations at a high cost to the layer coupling and cohesion metrics will be rejected.

The layer coupling metric we employ is also derived from the MOOD COF metric. We refer to it as the Layer Coupling Factor (LCF). It is essentially the ratio of the actual number of external layer couplings (classes that are coupled to classes outside of their own layer) to the maximum number of possible external layer couplings. We subtract this from 1 to yield a value in the range 0..1 where larger values correspond to improvements in coupling. See Equation 2.

$$LCF = 1 - \frac{actual\ number\ of\ external\ layer\ couplings}{maximum\ possible\ number\ of\ external\ layer\ couplings} \quad (2)$$

The layer cohesion metric we employ is based on the component cohesion metric suggested by Vernazza et al. [48], which is defined as the ratio of the number of internal classes a class is coupled with to the maximum number of possible coupling relationships among the classes. Most of the commonly used cohesion metrics, such as Lack of Cohesion of Methods (LCOM) and Tight and Loose Class Cohesion (TCC and LCC), measure the internal cohesion of a class by examining the relationships between the methods in the class. Vernazza's component cohesion metric is more applicable to our scenario because it measures cohesion at the component (i.e. package) level by examining the relationships between classes in the component. We implemented this metric as described by Vernazza et al., with each layer being viewed as a component, and refer to this metric as Layer Cohesion (LCOH); see Equation 3

$$LCOH = \frac{actual\ number\ of\ internal\ layer\ couplings}{maximum\ possible\ number\ of\ internal\ layer\ couplings} \quad (3)$$

The metrics described above are combined to form the fitness function that will guide our search, as defined in Equation 4.

$$fitness_{new} = \begin{cases} 0, \text{if } (ILCF + ILCFM + ILCFF) <= (ILCF + ILCFM + ILCFF)_{old} \\ (LCF + LCOH) + 2 * (ILCF + ILCFM + ILCFF), otherwise. \end{cases} \quad (4)$$

In this study we are only interested in improving the layer violation metrics. The cohesion and coupling metrics are included only because we do not want these values to degrade drastically; however, we are not interested in refactorings that improve only the cohesion and coupling, so their improvement is a secondary check. Hence in the fitness function we regard a refactoring as an improvement only if the sum of the layer violation metrics has improved, and there is an overall improvement. The overall improvement is calculated using a weighted sum approach where the violation metrics have twice the weight of the coupling and cohesion metrics. The violation metrics are weighted higher than the coupling and cohesion metrics to allow some disimprovement in the coupling and cohesion metrics if the layer violation metrics improve somewhat. The choice of 2 as the weighting for the violation metrics is arbitrary and further experimentation would be required to determine what the most suitable weighting is, or indeed if a completely different formulation of the fitness function is more appropriate.

Note that the fitness function we employ is comparable to the MQ function used in the Bunch clustering system [32] as it rewards high internal layer coupling (our LCOH metric) and penalizes high external layer coupling (our LCF metric).

3.3. Refactorings Employed

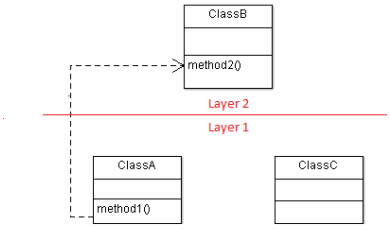
Most of the original refactorings provided by Code-Imp will have no impact on layering violations. We therefore introduced a number of new refactorings to Code-Imp, namely Move Method, Move Field and Move Class, as these are the refactorings most likely to remove layering violations. For the reasons provided below, we had to implement non-standard versions of these refactorings.

The simplest case is the Move Class refactoring. This normally moves a class from one package to another target package. We constrained it so that it the target packages for movement are obtained from the packages of classes that are illegally accessing the class to move. The Move Method and Move Field refactorings proved more challenging for two reasons. Firstly, selecting the target class for the move needs to be constrained and secondly, these refactorings normally involve the use of delegation, which is not appropriate in our case. We expand on these issues in the following paragraphs.

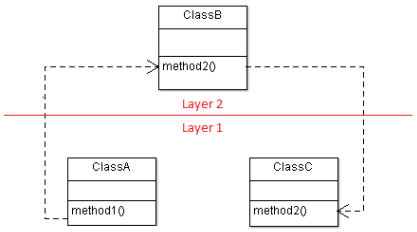
In the Move Method refactoring the target class is normally restricted to the classes of the parameters to the method and the classes of instance variables in the source class. In the Move Field refactoring, the target class is normally limited to the classes of the instance variables in the source class. This results in the majority of refactorings attempting to move fields and methods to classes that are in the same layer as the original class and so could not reduce the number of layer violations. We therefore allow as a target class only a class that is illegally accessing the field or method to move. In this way each refactoring has the potential to move a method or field out of its original layer to improve the layer violation metrics. Also, the search space is reduced and only fields and methods that are involved in layering violations will be considered for movement.

The second problem is the use of delegation in the refactorings. In the Move Method refactoring, if there is no link between the original class and the target class then a delegate method is normally added to the original class which delegates to the method to the target class. This was not useful for our purposes as the dependency between the calling class and the original class would not removed by this refactoring and so the violation would remain. We adjusted the Move Method refactoring to avoid this delegation and to instead create an instance of the target class in the calling class and call the method through this.

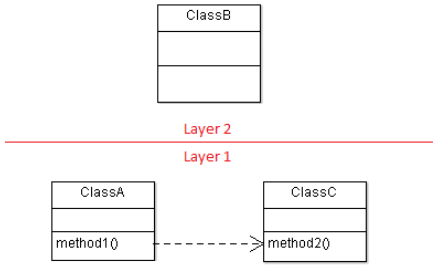
Figure 2 shows an example of a method-based layering violation and the results of a move method refactoring with and without delegation. In Figure (a) `ClassA.method1` in Layer 1 is calling `ClassB.method2` in Layer 2, causing a layering violation. Figure (b) shows the result of a 'normal' move method refactoring with delegation. `method2` is moved to `ClassC` and a delegation method is added to `ClassB` to invoke `ClassC.method2`. `ClassA` is not updated and still calls `method2` in `ClassB` so the layering violation between `ClassA` and `ClassB` is not removed. Figure (c) shows the result of a move method refactoring without



(a) An example of a layering violation



(b) After move method refactoring with delegation



(c) After move method without delegation

Fig. 2: The results of a move method refactoring with and without delegation

delegation, as we use in this paper. `method2` is moved to `ClassC` and `ClassA` is updated to invoke `method2` in `ClassC`. Since both `ClassA` and `ClassC` are in Layer 1, the layering violation is removed.

In the Move Field refactoring, delegation is always used. When a field is moved to a new location all classes using this field are updated so that instead of calling `originalClass.field` they call `originalClass.getTargetClass().field`. Again, this is not useful for our experiments as the dependency between the calling class and the original class would not be removed. Solving this problem in the general case while ensuring behaviour preservation is challenging, so for the experiments in the paper we elected to restrict the Move Field refactoring to move only static fields, which meant that delegation could be avoided by updating the calling classes to access the field directly in the target class.

In summary, our layer targeted approach supports three refactorings: Move Class, Move Method and Move (static) Field, with each one being defined in different way from manner in which it is described in the standard refactoring manual [49]. It should be noted that this is not surprising and that any refactoring that is to be used as part of an automated search process will have different requirements from a refactoring to be used directly by a developer.

3.4. Search Heuristics Employed

In a landmark paper in the SBSE field, Clark et al. [12] recommend that initial investigations should be performed using a simple hill climb in order to gain insight into the suitability of the problem for the SBSE approach before employing more sophisticated global search techniques. Accordingly we employ the following search techniques in our case study:

- Steepest ascent hill climbing (HCS)
- First-ascent hill climbing (HCF)
- Simulated Annealing (SA)
- Low Probability Simulated Annealing (SAL)

In steepest ascent hill climbing (HCS), all neighbours of the current solution are examined and the one with the greatest increase in the fitness function is selected. This means that in each iteration of the search,

every class in the codebase must have every refactoring applied to it and the refactoring must be rolled back. Once all refactorings have been evaluated, the refactoring that produced the greatest increase in the fitness function is reapplied. This search generally involves performing many more evaluations than other searches but it can be useful as a base result as it is deterministic.

First-ascent hill climbing (HCF) iterates over each class in the given codebase and attempts to refactor each one. After each refactoring is applied, the fitness function is evaluated and if the value is improved then the refactoring is accepted and the hill climb continues from that point. If it is not improved then the refactoring is rolled back and alternative refactorings are attempted on the same class. This process terminates when it reaches a point where none of the classes can be refactored to improve the fitness function.

Simulated annealing uses a similar approach to first-ascent hill climbing, but rather than rejecting all refactorings that produce a lower value in the fitness function, some of these refactorings will be accepted with a probability that decreases as the search continues. In this way the search can accept some drops in quality in order to escape from local maxima. The probability of accepting an inferior solution is calculated using a standard formula based on the amount of disimprovement in the fitness value and the temperature. The temperature decreases as the search proceeds; in our case this occurs each time a refactoring is accepted. Low Probability Simulated Annealing (SAL) involves using an adjusted formula so that the probability of accepting a quality-reducing refactoring is lower.

The results produced by each of these searches will be discussed in the the following section.

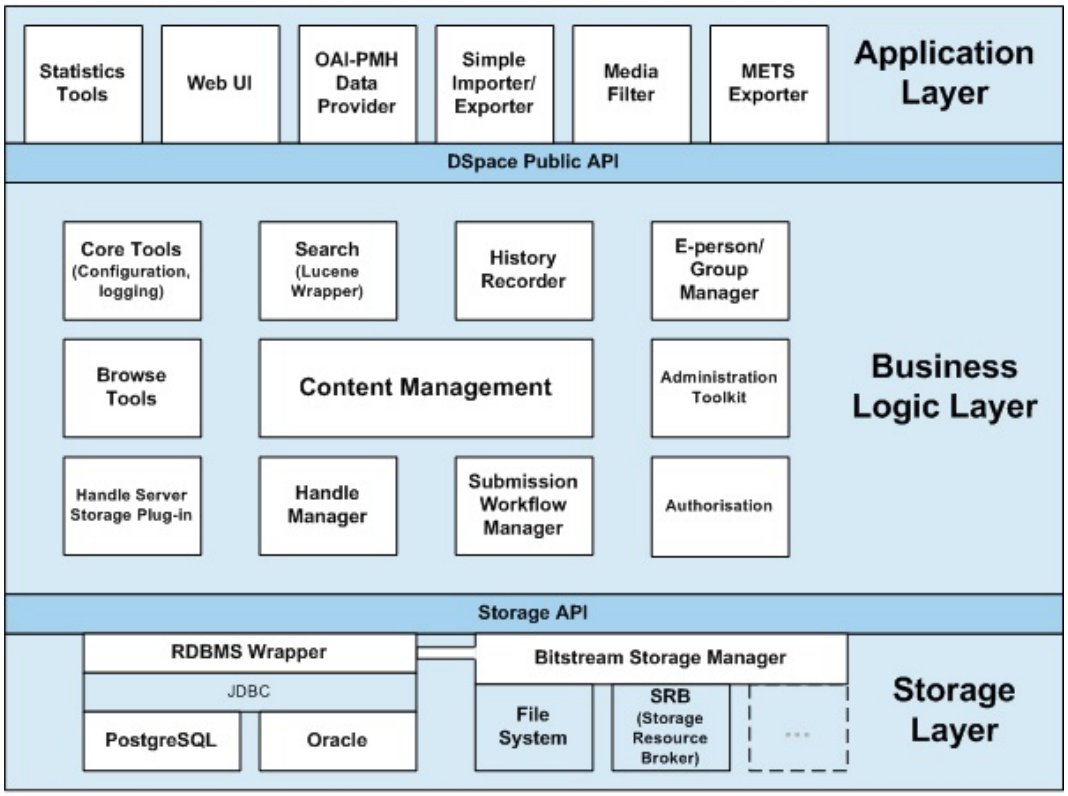


Fig. 3: The documented "as intended" DSpace architecture.

4. Case Study

In this section we evaluate the approach described in Section 3 by applying each of the search techniques of Section 3.4 to Version 1.8 of the open source content management system, DSpace [20]. DSpace has a

layered architecture, and in an earlier work [21] we discovered that a number of architectural constraints have been violated during its development, so we anticipated that it may also suffer from some layering violations.

DSpace is organised into three layers as depicted in Figure 3. The documentation describes the responsibilities of each layer as follows:

- Storage Layer (Layer 1): responsible for physical storage of metadata and content
- Business Logic Layer (Layer 2): manages content, users, authorisation and workflow
- Application Layer (Layer 3): responsible for external communication e.g. through the user interface

The remainder of this section is organised as follows. In Section 4.1 we describe the violations that we found in DSpace and discuss the results of refactoring the codebase to try to fix these violations. In Section 4.2 we provide a qualitative analysis of refactorings applied in the case study and in Section 4.3 the violations that could not be repaired by the refactoring process are analysed. The conclusions of the study are presented in Section 4.4.

4.1. Layer Violations in DSpace

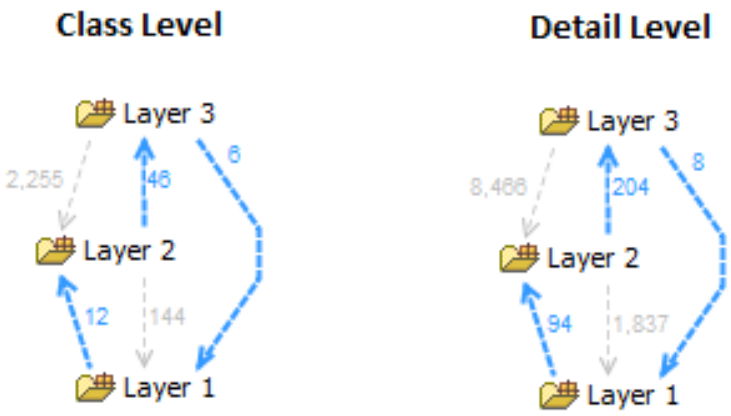


Fig. 4: The layer coupling count of the DSpace system before refactoring, as produced by the Structure 101 tool [27]. Detail level couplings are those from methods to other methods and fields. Illegal couplings are marked in blue (dark grey).

Figure 4 shows the coupling between the layers of the DSpace system before it is refactored. The couplings that violate the layered architecture are shown in blue (dark grey). DSpace has 64 (12+46+6) class level violations and 306 detailed level violations (94+205+8). This shows that there has been considerable architectural drift in the system and that the documented architecture no longer matches the actual architecture. The high level of interconnectivity between the layers mean that the system is less flexible. For example it would be difficult to replace an entire layer in this system with an alternative implementation because of the dependency tangles. The system will also be more difficult to maintain since changes to one layer may require changes in all other layers.

The DSpace codebase was refactored once with the deterministic HCS search and 10 times with each of the HCF, SA and SAL searches. The runs which produced the best ILCF values were selected in each case and the change in metrics for each one is shown in Table 1.

The results show that the searches that explore more of the search space apply more refactorings and remove more violations. HCS removes the fewest violations. It is deterministic and will follow one path through the search space until it reaches a local maximum. On the other hand, the HCF search will accept the first improvement it finds and with multiple runs it can follow multiple different paths and possibly

Table 1: Results of refactoring DSpace for the various searches. Each percentage indicates the extent to which the metric improved.

Search	No. of Refactorings	ILCF	LCF	LCOH
HCS	13	42%	-0.1%	2.6%
HCF	20	62%	-1.6%	0.9%
SAL	53	69%	-2.2%	2.4%
SA	66	78%	-1.7%	5.4%

reach several local maxima. This allows it to examine solutions that will not be found by the HCS search and results in better solutions being found. The SA searches can escape from local maxima by accepting inferior solutions. The SAL search does this with a lower probability than the SA search but both searches find better solutions than the HCS search because of the additional search space that is explored. The results in each case show relatively little impact to the LCF and LCOH metrics. In each case there is a slight degradation in the LCF metric which means that coupling between the layers has increased slightly. There is also a slight increase in the LCOH metric in each case which means that the cohesion of the layers has been slightly improved.

It seems as if the SA searches have produced the best results based on the improvement in the ILCF metric that was achieved. However, when we inspect the refactorings that were performed by each search we find that many of the refactorings performed by the SA searches move classes, methods and fields to new locations that are less suitable than their previous locations from a functional perspective. We judge this suitability based on the functional layer descriptions given in the architectural documentation. For example, the documentation tells us that Layer 1 is the storage layer, responsible for the physical storage of content, and that Layer 2 is the business logic layer. The results of the SA searches contain some refactorings that move JDBC classes and classes that deal with setting up the database connection from the storage layer to the business logic layer, which is incorrect from a functional point of view of the layers.

It is interesting to consider why this problem arises only with SA. We try to prevent these unsuitable refactorings from being accepted by using the layer cohesion and coupling metrics (LCF anf LCOH in Equation 4). Using these metrics, the suitability of a new location is judged based on the interaction of the moved class with other classes in the new layer and other classes outside the layer. However during the SA search a refactoring that has a negative impact on these metrics may be accepted, thus causing a class to be moved to an inappropriate layer. This changes the fitness landscape and has a "knock-on" effect that classes that interact with this misplaced class may be inappropriately moved to this layer as well in later refactorings. This problem is partly due to the nature of the SA algorithm, and also suggests that the fitness function needs to be enriched to take the functional aspects of the layers in to account.

Due to these issues with the SA and SAL results, we focus on the HCF results which remove a high number of violations and, as we will see in the refactoring analysis in Section 4.2, perform more functionally suitable refactorings.

Architectural diagrams were generated from the HCF refactored codebase and are shown in Figure 5. After refactoring, the system has 22 class level violations and 121 detail level violations. This is a reduction of 65% in class level violations and a reduction of 60% in detail level violations.

In the next section we examine the refactorings that were applied by the HCF search and analyse whether or not these movements were appropriate. We will also look some of the unresolvable violations to determine why they could not be resolved.

4.2. Qualitative Analysis of Refactorings

The HCF search performed twenty refactorings. These refactorings comprised fifteen class moves, four method moves and one field move. Some of these refactorings are analysed below.

Six classes were moved from Layer 2 (the business layer) to Layer 3 (the UI layer). These classes contained methods that were taking in and processing `HttpServletRequests` and throwing `ServletExceptions`.

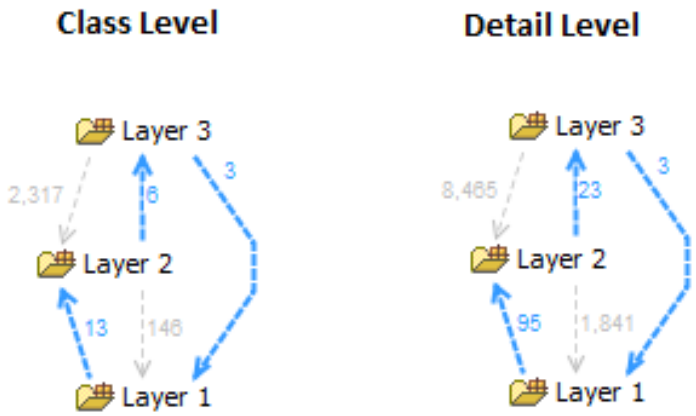


Fig. 5: The layer coupling count of the DSpace system after refactoring. Illegal couplings are marked in blue (dark grey).

They seem to be related to UI interaction and so a move to Layer 3 is in keeping with the architectural layer descriptions.

The class `Util` was moved from Layer 3 to Layer 2. It contains miscellaneous utility methods used in Layer 2 and Layer 3. Since these utility methods are not UI related a move to the business layer is appropriate.

An authorisation method was moved from the class `AuthorizeUtil` (Layer 3) to the class `Collection` (Layer 2). Subsequently the class `AuthorizeUtil` was moved from Layer 3 to Layer 2. The documentation states that Layer 2 is responsible for authorisation so these movements seem correct from a functional perspective. While the method movement is correct from a layer perspective it may be less correct from a class cohesion perspective and could suggest the need for class level metrics to be included in the fitness function.

The classes `LogLine` and `LogAnalyzer` were moved from Layer 3 to Layer 2. Both are used by `ClassicDSpaceLogConverter` in Layer 2. These three log-related classes belong in the same layer but it is unclear from the layer descriptions whether they should all be in Layer 2 or Layer 3. In some runs of the experiment `ClassicDSpaceLogConverter` is moved to Layer 3 rather than moving the others to Layer 2. This group of classes is quite independent with only `LogAnalyzer` being accessed by a class outside the group so the cohesion and coupling values are similar whether they are all moved to Layer 2 or Layer 3. Which layer they end up in depends simply on which class is refactored first.

The class `DSpaceContextListener` was moved from Layer 3 to Layer 1. According to class comments it is used to initialize or clean up resources when the web application is started or stopped. It accesses `DatabaseManager` in Layer 1. It seems not to belong in Layer 1 and a refactoring that provides delegation through Layer 2 may be a better alternative here.

4.3. Analysis of Unresolved Violations in DSpace

There are three class level violations remaining between Layer 3 and Layer 1 that were not repaired by the refactoring process. In these cases it is not appropriate to move either the accessing classes in Layer 3 or the accessed class in Layer 1 to Layer 2 so a refactoring that provides delegation through an intermediate class in Layer 2 could be an appropriate solution. This suggests that Delegate Through Intermediate Layer is a possible refactoring that should be provided in this type of architecture repair process.

There are 23 remaining violations between Layer 2 and Layer 3 that were not repaired by the refactoring process. 22 of these violations involve the `SubmissionInfo` class in Layer 3. They are coming from classes in Layer 2 that extend the class `AbstractProcessingStep`. A number of other classes extending

`AbstractProcessingStep` were indeed moved to Layer 3 but some were prevented from moving by a significant drop in the cohesion metric. It seems that this entire package should be moved to Layer 3 in one single step, rather than assessing each class individually. This suggests that Move Package is another possible refactoring that should be provided in this type of architecture repair process.

There are 95 violations remaining between Layer 1 and Layer 2. 62 of these violations are on the `Context` class. This is used to store a database connection and information about the current user. It is in Layer 2 but is used in all layers. Layer 3 should not be concerned with database connections so should not need to use this context object. It is clear that this class should be split into two classes: one containing what is needed for the storage layer and the other containing what is needed for the presentation layer. This is a non-trivial refactoring that would require extensive knowledge of the system to perform.

4.4. DSpace Case Study Conclusion

The DSpace system was refactored using four different search techniques and all of the searches improved the ILCF metric with minimal impact on the LCF and LCOH metrics. While the simulated annealing searches removed the most violations, it was discovered that these searches may not be suitable for use with our metrics as they are inclined to repair violations by moving classes to functionally inappropriate locations. The first ascent hill climbing search removed 65% of class level layering violations, and, critically, the majority of the class movements it performed were appropriate although method movements could be improved by the addition of class level metrics to the fitness function. These results could be further improved by the addition of more refactorings to allow for package movement, the splitting of classes and delegation through intermediate layers. Some violations require radical refactorings that cannot be solved by Code-Imp and would require human intervention.

5. Conclusions and Future Work

Architecture erosion is a common problem, as evidenced by the number of architectural layer violations seen in our case study of DSpace. Current approaches in architecture erosion repair tend to involve a significant amount of manual analysis. In this paper we apply search-based refactoring techniques to this problem in order to fully automate architecture repair for layered architectures. To this end, we extended the Code-Imp refactoring framework to parse architectural information and measure layered architecture violations, as well as adding a number of refactorings specifically geared for architecture repair. Using a focussed architecture violation metric as part of the fitness function, we conducted a case study on the DSpace application and found that, through the movement of classes, method and fields, the search based approach was capable of removing up to 78% of layering violations.

In future work we plan to extend Code-Imp with further ‘architecture-aware’ refactorings such as Move Package, Split Class, Delegate Through Intermediate Layer as well as a Move Field refactoring that can act on non-static fields. Now that the approach has been proven using local search, it makes sense to consider a global search technique like a genetic algorithm to determine if better solutions can be found with a deeper search. This will probably require the extension of the fitness function to include some class level metrics to reduce the possibility that poor quality refactorings are chosen. Finally, our case study involved only a single software application; in order to determine if our approach is of general value, and if the fitness function we used is of general applicability, it will be necessary to experiment with a larger set of applications.

Acknowledgement

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

References

[1] Murphy, G. C., Notkin, D. and Sullivan, K. J. *Software reflexion models: Bridging the gap between design and implementation*. IEEE Transactions on Software Engineering, 27.4 (2001): 364-380.

[2] Koschke, R., and Simon, D. Hierarchical Reflexion Models. *Proceedings of the 10th Working Conference on Reverse Engineering*. 2003.

[3] Rosik, J. et al. *An industrial case study of architecture conformance*. Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement. ACM, 2008.

[4] Postma, André. *A method for module architecture verification and its application on a large component-based system*. Information and Software Technology 45.4 (2003): 171-194.

[5] Jansen, A., Bosch, J. and Avgeriou, P. *Documenting after the fact: Recovering architectural design decisions*. Software Engineering, Journal of Systems and Software 81.4 (2008): 536-557.

[6] Morrison, R. et al. *Support for evolving software architectures in the ArchWare ADL*. Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on. IEEE, 2004.

[7] Garlan, David, and Bradley Schmerl. *Model-based adaptation for self-healing systems*. Proceedings of the first workshop on Self-healing systems. ACM, 2002.

[8] Bourquin, Fabrice, and Rudolf K. Keller. *High-impact refactoring based on architecture violations*. Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on. IEEE, 2007.

[9] Tran, J. B., and Richard C. Holt. *Forward and reverse repair of software architecture*. Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research. IBM Press, 1999.

[10] Tran, J. B., et al. *Architectural repair of open source software*. Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on. IEEE, 2000.

[11] Terra, R. et al. *A recommendation system for repairing violations detected by static architecture conformance checking*. Software: Practice and Experience (2013).

[12] Clark, J. et al., 2003, Reformulating software engineering as a search problem, *IEEE Proceedings Software*, vol. 150, no. 3, pp. 161175.

[13] Fatiregun, Deji, Mark Harman, and Robert M. Hierons *Evolving transformation sequences using genetic algorithm*.Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on. IEEE, 2004.

[14] Seng, Olaf, Johannes Stammel, and David Burkhart. *Search-based determination of refactorings for improving the class structure of object-oriented systems*. Proceedings of the 8th annual conference on Genetic and evolutionary computation. ACM, 2006.

[15] O'Keeffe, Mark, and Mel Ó Cinnéide. *Search-based refactoring for software maintenance*. Journal of Systems and Software 81.4 (2008): 502-516.

[16] O'Keeffe, Mark, and Mel Ó Cinnéide. *Searchbased refactoring: an empirical study*. Journal of Software Maintenance and Evolution: Research and Practice 20.5 (2008): 345-364.

[17] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum. Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering*, 20(1):47–79, 2013.

[18] W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, ASE'14. IEEE Press, 2014.

[19] Moghadam, Iman Hemati, and Mel Ó Cinnéide. *Code-Imp: A tool for automated search-based refactoring*. Proceedings of the 4th Workshop on Refactoring Tools. ACM, 2011.

[20] DSpace. *DSpace repository application* http://www.dspace.org/

[21] M. Ziane and M. Ó Cinnéide, *The Case for Explicit Coupling Constraints*. http://arxiv.org/abs/1305.2398, 2013.

[22] Kruchten, Philippe B. *The 4+ 1 view model of architecture*. Software, IEEE 12.6 (1995): 42-50.

[23] Garlan, David, and Mary Shaw. *An introduction to software architecture*. Advances in software engineering and knowledge engineering 1 (1993): 1-40.

[24] Bushmann, F. et al. *Pattern-oriented software architecture: A system of patterns.*. John Wiley&Sons (1996).

[25] De Silva, Lakshitha, and Dharini Balasubramaniam. *Controlling software architecture erosion: A survey*. Journal of Systems and Software 85.1 (2012): 132-151.

[26] Aldrich, Jonathan, Craig Chambers, and David Notkin. *ArchJava: connecting software architecture to implementation*. Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on. IEEE, 2002.

[27] Structure101, 2001 *Software architecture management*. http://www.headwaysoftware.com/

[28] Hello2Morrow, *SonarJ overview*. http://www.hello2morrow.com/products/sonarj, 2011.

[29] Constantinou, Eleni, George Kakarontzas, and Ioannis Stamelos. *Open Source Software: How Can Design Metrics Facilitate Architecture Recovery?* arXiv preprint arXiv:1110.1992 (2011).

[30] Herold, S. and Mair, M. Recommending Refactorings to Re-establish Architectural Consistency In *Software Architecture*, Lecture Notes in Computer Science, vol. 8627, 2014.

[31] Mancoridis, S. et al. *Using automatic clustering to produce high-level system organizations of source code*. 6th International Workshop on Program Comprehension. Published by the IEEE Computer Society, 1998.

[32] Mancoridis, S. et al. *Bunch: A clustering tool for the recovery and maintenance of software system structures*. Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on. IEEE, 1999.

[33] Mitchell, Brian S., Spiros Mancoridis, and Martin Traverso. *Search based reverse engineering*. Proceedings of the 14th international conference on Software engineering and knowledge engineering. ACM, 2002.

[34] Harman, Mark, Stephen Swift, and Kiarash Mahdavi. *An empirical study of the robustness of two module clustering fitness functions*. Proceedings of the 2005 conference on Genetic and evolutionary computation. ACM, 2005.

[35] Tucker, Allan, Stephen Swift, and Xiaohui Liu. *Variable grouping in multivariate time series via correlation*. IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics, 31.2 (2001): 235-245.

[36] Huynh, S. et al. *Automatic modularity conformance checking*. Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on. IEEE, 2008.

[37] O'Keeffe, M. and Ó Cinnéide, M., A stochastic approach to automated design improvement, *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 59–62, 2003.

[38] M. Ó Cinnéide, D. Boyle, and I. Hemati Moghadam. Automated refactoring for testability. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2011)*, Berlin, Mar. 2011.

[39] S. Ghaith and M. Ó Cinnéide. Improving software security using search-based refactoring. In *Proceedings of the 4th International Symposium on Search Based Software Engineering (SSBSE '12)*, pages 121–135, Riva del Garda, Italy, 2012.

[40] Mark Kent O'Keeffe and Mel Ó Cinnéide. Getting the most from search-based refactoring. In *Proceedings of the 9th annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1114–1120, New York, NY, USA, 2007. ACM.

[41] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi. Search-based refactoring: Towards semantics preservation. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM '12)*, pages 347–356. IEEE, 2012.

[42] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi. The use of development history in software refactoring using a multi-objective evolutionary algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '13)*, pages 1461–1468, Amsterdam, The Netherlands, July 2013.

[43] A. Ghannem, G. El-Boussaidi, and M. Kessentini. Model refactoring using interactive genetic algorithm. In *Proceedings of the 5th International Symposium on Search Based Software Engineering (SSBSE '13)*, pages 96–110, St. Petersburg, Russia, 2013.

[44] I. Hemati Moghadam and M. Ó Cinnéide. Automated refactoring using design differencing. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR '12)*, pages 43–52, Szeged, Hungary, 2012.

[45] Harman, Mark, and Laurence Tratt *Pareto optimal search based refactoring at the design level*. Proceedings of the 9th annual conference on Genetic and evolutionary computation. ACM, 2007.

[46] A. Jensen and B. Cheng. On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Proceedings of the Conference on Genetic and Evolutionary Computation*, page 1341–1348. ACM, July 2010.

[47] Abreu, F. Brito, Miguel Goulão, and Rita Esteves. *Toward the design quality evaluation of object-oriented software systems*. Proceedings of the 5th International Conference on Software Quality, Austin, Texas, USA. 1995.

[48] Vernazza, T. et al. *Defining metrics for software components*, Proceedings of the World Multi-conference on Systematics, Cybernetics and Informatics, 2000.

[49] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.