# Inferring Architectural Evolution from Source Code Analysis

## A Tool-Supported Approach for the Detection of Architectural Tactics

Christel Kapto[1], Ghizlane El Boussaidi[1(✉)], Sègla Kpodjedo[1], and Chouki Tibermacine[2]

[1] Department of Software and IT Engineering, École de Technologie Supérieure, Montreal, Canada
ghizlane.elboussaidi@etsmtl.ca
[2] LIRMM - CNRS and University of Montpellier II, Montpellier, France

**Abstract.** Several approaches have been proposed to study and provide information about the evolution of a software system, but very few proposals analyze and interpret this information at the architectural level. In this paper, we propose an approach that supports the understanding of software evolution at the architectural level. Our approach relies on the idea that an architectural tactic can be mapped to a number of operational representations, each of which is a transformation described using a set of elementary actions on source code entities (e.g., adding a package, moving a class from a package to another, etc.). These operational representations make it possible to: (1) detect architectural tactics' application (or cancellation) by analyzing different versions of the source code of analyzed systems, and (2) understand the architectural evolution of these systems. To evaluate the proposed approach, we carried out a case study on the JFreeChart open source software. We focused on the modifiability tactics and we analyzed a number of available releases of JFreeChart. The results of our analysis revealed inconsistencies in the evolution of the system and some erratic applications and cancellations of modifiability tactics.

**Keywords:** Software evolution · Architectural evolution · Architectural tactics · Tactics detection

## 1 Introduction

Throughout the life of a software system, developers and maintainers will modify the source code in order to add new features, correct or prevent defects. In doing so, they will apply many simple coding techniques and patterns but they will also occasionally introduce higher level elements that will be meaningful at an architectural level. While there are many proposals concerned about evolution data at a low level [1], few approaches have been proposed to analyze and interpret this information at the architectural level. Even though several approaches that tackle the understanding and formalization of architecture evolution have emerged (e.g., [2–8]), there exist very few

© Springer International Publishing AG 2016
B. Tekinerdogan et al. (Eds.): ECSA 2016, LNCS 9839, pp. 149–165, 2016.
DOI: 10.1007/978-3-319-48992-6_11

tools to help designers track and group a set of low-level source code changes and translate them into a more concise high-level architectural intention. A key challenge is that some architectural elements may not be traced easily and directly to code elements (e.g., architectural constraints). In fact, architectural elements include extensional elements (e.g., module or component) and intensional ones (e.g., design decisions, rationale, invariants) while source code elements are extensional [9, 10]. This contributes to the absence of the architectural intention at the source code level and the divergence of the source code from this intention. Moreover, architectural decisions are non-local [9] and often define and constrain the structure and the interactions of several code elements. If the developer is aware of the architectural decisions and constraints, the changes she made to the source code will be consistent with these. In fact, some of these changes may derive from the architecture evolution of the software and they reveal some intentions at the architectural level.

Thus, in this work, we hypothesize that some of the architectural intentions can be inferred from the analysis of the evolution of the source code. Clustering a set of changes made to the source code and analyzing the results may reveal a high level decision. We focus on object-oriented (OO) systems and modifiability tactics [11, 12] as they involve changes that can be detected through the analysis of different releases of a software system. Thus we propose an approach that enables detecting tactics' application (or cancellation) in an OO system and inferring an architectural evolution trend through the system's evolution. To do so, we map high level descriptions of tactics, as introduced in [11], to a number of operational representations (i.e., source code transformations). Tactics are intensional and thus may have several operational representations. An operational representation is a pattern of evolution described using elementary actions on source code entities (e.g., adding a class to a package, moving a class from a package to another, etc.) and a set of constraints describing the structure of the system before or after these actions. Using these operational representations, we analyze available evolution data about the source code to retrieve architectural tactics that were applied or cancelled during development or maintenance. We developed a prototype tool that supports our approach and experimented on a set of modifiability tactics and a number of versions of a Java open source project.

The paper is organized as follows. Section 2 proposes some background and related work about architectural tactics and evolution. Section 3 presents an overview of our approach while Sects. 4 and 5 detail two key aspects of our proposal: the definition of operational representations of tactics and the detection of their occurrences respectively. Section 6 proposes a case study for our approach and discussion of the obtained results. Finally Sect. 7 summarizes our proposal and outlines future work.

## 2 Background and Related Work

### 2.1 Architectural Tactics

Architectural tactics are design decisions that achieve quality attributes [11, 12]. Quality attributes are measurable properties that indicate how well a given system supports specific requirements [11]. Examples of these attributes include performance,

availability and security. Bass et al. [11] introduced the concept of an architectural tactic as an architecture transformation that supports the achievement of a single quality attribute. They catalogued a set of common tactics that address availability, interoperability, modifiability, performance, security, testability and usability. This catalog of tactics aims to support systematic design. For instance, performance tactics aim at ensuring that the system responds to arriving events within some time constraints while security tactics aim at resisting, detecting and recovering from attacks [11]. Examples of performance tactics include increasing computational efficiency, managing the event rate and introducing concurrency. Common security tactics include authenticating users and maintaining data confidentiality. The designer chooses the appropriate tactics according to the system's context and trade-offs, and the cost to implement these tactics.

### 2.2 Related Work

Developing approaches and tools that support the designers in understanding architectural evolution involves many theoretical and practical challenges [20]. Several approaches were proposed to tackle the architectural evolution of software systems. These approaches can be classified according to their goal: (1) supporting architects in building software evolution plans at the architectural level (e.g., [2, 3]); (2) understanding and visualizing the evolution [5, 6, 13, 14]; and (3) evaluating architectural stability [4, 8]. With the goal of supporting architects in building software evolution plans at the architectural level, the concept of evolution paths was introduced in [2, 3]. An evolution path is a sequence of intermediate architectures starting from the initial architecture of the system and leading to the desired architecture once the evolution is complete. These evolution paths can be represented in an evolution graph where nodes are (intermediate) architectures and edges are transitions among these architectures. To support the architect in finding the optimal path, the authors propose analysis based on constraints on the path evolution and functions that evaluate the path qualities. Even if our focus is on tactics' detection, our work can be seen as complementary as we analyze existing software systems to infer architectural decisions that were applied through the evolution of these systems and to check if the changes made to a given system represent a consistent pattern of evolution.

In [6], the authors propose a method for differencing and merging component and connector architecture views by comparing the structural elements composing these views. The comparison and matching between different views may help to identify architectural violations and synchronize the views. The proposed approach does not tackle the particular problem of identifying architectural tactics when comparing architecture views. The case studies presented in their paper are related to the synchronization of an implementation-level architecture view (obtained using architecture recovery) with a conceptual one (described using an ADL). This feature can be perceived as complementary to our work. With the focus on visualization, both [5, 13] propose techniques that exploit source code modifications to understand software evolution at architectural level. In particular, McNair et al. [5] propose a diagram, called architectural impact view, which is basically an entity-relationship diagram

enhanced with colors to depict the impact of the code changes under study on the entities and relationships of the system (e.g., added, deleted, etc.). D'Ambros et al. [13] describe a general schema to analyze software repositories for studying software evolution. This schema includes three essential steps: (1) modeling various aspects of the software system and its evolution, (2) retrieving and processing the information from the relevant data sources, and (3) analyzing the modeled and retrieved data using appropriate techniques depending on the targeted software evolution problem. Though we do not target the visualization of architecture evolution, our approach follows this general schema and we also aim to help designers and developers understand and be aware of the architectural evolution of a given system.

Le et al. [8] propose an approach called ARCADE (Architecture Recovery, Change, And Decay Evaluator) which relies on various architecture recovery techniques to build different views of the analyzed system and three metrics for quantifying architectural changes at the system-level and component-level. ARCADE was used in an empirical study. An interesting outcome of this study was that considerable architectural change is introduced both between two major versions and across minor versions. In [4], a metric-based approach is proposed to evaluate architectural stability. To do so, the approach starts by analyzing different releases of the system under study and extracting facts from these releases. These facts are then analyzed using some software metrics that are indicators of architectural stability (e.g., change rate, growth rate, cohesion and coupling). Our approach can be complementary to these metric-based approaches as it relies on the detection of tactics applications or cancellations to assess the architectural evolution of software systems.

Kim et al. [14] proposed Ref-Finder, an Eclipse plug-in, that automatically detects refactorings that were applied between two versions of a given program. To do so, Ref-Finder extracts logic facts from each program version and used predefined logic queries to match program differences with the constraints of the refactorings under study. This approach is more focused on the refactorings introduced in Fowler's book [15]. Unlike Ref-Finder, our goal is to detect evolution patterns that match architectural tactics and to support the designer in defining any evolution pattern that might be of interest in her context/domain.

## 3 An Approach for Inferring Architectural Evolution from Source Code

In this paper, we propose an approach that supports the detection of architectural tactics' application (or cancellation) and the inference of the architectural trend through the system's evolution. Our approach assumes that high level descriptions of tactics, as introduced in [11], can be mapped to a number of operational representations, i.e., source code transformations described using elementary actions on source code entities (e.g., adding a package, moving a class from a package to another, etc.). Once these operational representations are identified and precisely defined, it becomes possible to use evolution data about the source code to retrieve architectural tactics that were applied or cancelled during development or maintenance.

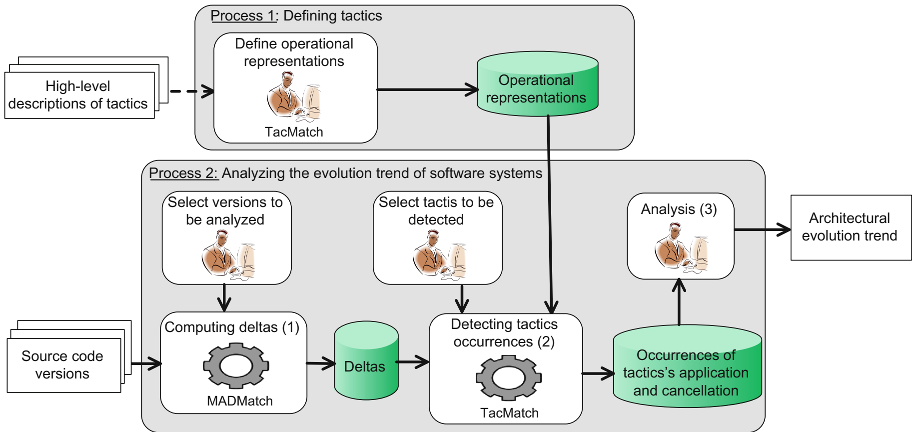


**Fig. 1.** Overview of the approach

Figure 1 presents an overview of our approach which defines two processes. The first enables the designer to specify operational representations of a given tactic; this process is described in Sect. 4. The second process aims at supporting the designer in analyzing the evolution trend of a software system. It uses the operational representations of tactics and the available versions of the system under study and proceeds in three steps (numbered 1 to 3 in Fig. 1). In the first step, a differencing tool is applied to multiple versions of the system and generates deltas that are expressed using a number of source code changes (e.g., removed package, added package, added class, removed class, moved class, etc.). For this purpose, our approach uses MADMatch [16] a tool that enables a many-to-many approximate diagram matching approach. The second step matches the generated deltas to the operational representations of tactics to detect applied or cancelled tactics. We designed and implemented a tool TacMatch which generates on the fly detection algorithms from the operational representations of tactics and executes these detection algorithms to find occurrences of tactics in the analyzed delta of the source code. In the third step, the resulting occurrences are analyzed by the designer to infer the architectural evolution trend of the analyzed system. The whole process is described in detail in Sect. 5.

## 4 Defining Operational Representations of Tactics

### 4.1 High-Level Descriptions of Tactics

As stated above, a tactic can be seen as a transformation undergone by software architecture to satisfy a specific quality attribute. Thus a tactic can be described as a set of actions that may change the structure and behavior of the components of the system. The type and magnitude of these actions depend on the tactic and the current architecture of the system to which the tactic is to be applied. We roughly divide these actions into two types:

- Actions on components: create, delete or modify components. A component may be modified by adding new responsibilities, deleting its responsibilities or moving some of its responsibilities to another component.
- Actions on connectors: add, modify or remove a connector.

Consider the modifiability quality attribute. Modifiability refers to the property of changing easily the software with a minimal cost (i.e., time and resources). Tactics that ensure this property are linked to four concerns that impact the modifiability [11, 12]: the size of the modules, the cohesion of the modules, the coupling between the modules, and the binding time of modification. Thus modifiability tactics are categorized according to the concern they address: reducing the size of a module, increasing cohesion, reducing coupling between modules, and deferring binding time of modification. We focus on these tactics as they involve actions that can be detected through static analysis of different releases of a software system; i.e., common modifiability tactics involve splitting responsibilities, moving them from a component to another, introducing intermediaries between components and encapsulating components.

For instance, the modifiability tactic "Abstract Common Services" (ACS) states that common services should be abstracted so that modifications to them would be localized to a single module. Figure 2 gives a high level representation of this tactic. A and B are responsibilities that can be split respectively to A' and A", and B' and B" and where A' and B' provide a variant of a similar service to A" and B", respectively. In this case, the ACS tactic merges A' and B' into a more general and common service (called C in the figure) and updates A" and B" to depend on the general service. Applying the ACS tactic enables to localize modifications of the common services and to prevent ripple effects as changes made to a module using the common services will not impact other modules [11, 12].

**Fig. 2.** A high level representation of Abstract Common Services, adapted from [12].

Table 1 presents the high level description of the ACS tactic in terms of actions on architectural components and connectors.

**Table 1.** High-level description of Abstract Common Services

| Type of action | High-level description |
|---|---|
| Actions on components | Create C |
| Actions on components | Move A' from A to C |
| Actions on components | Move B' from B to C |
| Actions on connectors | Modify A" to depend now on C |
| Actions on connectors | Modify B" to depend now on C |

### 4.2 Operational Representations of Tactics: Actions and Constraints

High-level descriptions of tactics must be refined in order to generate concrete design/implementation strategies, while taking into consideration the system's context. In this paper, we target the analysis of object oriented (OO) systems. Thus, architectural components involved in tactics' application are matched with the entities of the system such as packages and classes. The responsibilities of a given component are mapped to fields and methods implemented by the classes that are part of this component. This mapping introduces multiple possible concrete instances for a given tactic; e.g., we may map the modules of the ACS tactic to packages in a concrete instance and to classes in another instance. As for architectural connectors, they are not explicitly supported by typical OO languages [17]; they are indirectly specified through method calls, references and events. Thus our operational representations of tactics are expressed as a set of **actions** (i.e., adding, deleting, modifying and moving) on packages, classes, methods, fields, object references, method calls and events.

Furthermore, the same set of actions may be common to different tactics. For instance, both Split Responsibility (SR) and Abstract Common Service (ACS) tactics involve moving responsibilities from a module (i.e., package or class in our context) to another. However, in case of ACS, the moved responsibilities belonged to different modules before applying the tactic while in SR the moved responsibilities belonged to the same module before applying the tactic. To distinguish these tactics, we added a set of constraints on the elements or actions involved in a given tactic. Thus we express an operational representation as a set of actions on architectural elements and a set of constraints relating these elements or actions. Once an operational representation is defined, its cancellation is simply derived by reversing the source and destination of the different actions and constraints used in its definition. For example, if a tactic definition involves adding a class, its cancellation would involve deleting a class. Table 2 lists some examples of operational representations for four modifiability tactics in the context of an object oriented system. For instance, Table 2 lists three different operational representations of the ACS tactic.

### 4.3 Tool Support

To support the developer in defining the operational representations of tactics or any other relevant evolution pattern, we use a language that resembles the natural language and eases the translation of the concrete representations into detection algorithms. In fact, we wanted to provide a way for a user to specify the tactics (or any targeted evolution pattern) without having to know a specific language to do so. The user has only to know the actions of the tactic (or any targeted evolution pattern) on architectural elements and how these elements are constrained.

Thus, to define operational representations of tactics, we designed and implemented a custom interface that was inspired by query languages such as SQL and QBE (Query By Example). Figure 3 displays the TacMatch interface for defining operational representations of tactics. This interface is divided into four parts: (1) the name of the tactic and the variant if there are many variants of the tactic; (2) a selector zone that enables the user to select the type of changes/actions the tactic introduces (i.e., a set of predefined actions are provided to the user); (3) a filter zone that enables the user to

**Table 2.** Examples of operational representations

| Tactic | Concrete representation (s) | Tactic | Concrete representation (s) |
|---|---|---|---|
| Abstract common services **(ACS)** | **P**: added or existing package<br>**C**: moved classes to **P**<br>Classes in **C** did not belong to the same package in the previous release | Split responsibilities **(SR)** | **P**: added package<br>**C**: moved classes to **P**<br>All classes in **C** belonged to the same package in the previous release |
| Abstract common services **(ACS)** | **C**: added class or existing class<br>**M**: moved methods to **C**<br>Methods in **M** did not belong to the same class in the previous release | Split responsibilities **(SR)** | **C**: added class<br>**E**: moved elements (attribute and method) to **C**<br>All elements in **E** belonged to the same class in the previous release |
| Abstract common services **(ACS)** | **C**: added class<br>**Inherits C**: added inheritance<br>All classes involved in "**Inherits C**" existed in the previous release<br>These classes belong to at least two different packages in next release | Use encapsulation **(UE)** | **C**: added class<br>**Inherits C**: added inheritance<br>All classes involved in "**Inherits C**" existed in the previous release<br>These classes belong to the same package in the next release |
| Increase cohesion **(IC)** | **C**: moved classes to package $\mathbf{P_{dest}}$<br>All classes in **C** belonged to the same package ($\mathbf{P_{src}}$) in the previous release<br>$\mathbf{P_{dest}}$ existed<br>Cohesion of $\mathbf{P_{src}}$ increased | Increase cohesion **(IC)** | **E**: moved elements (attribute and method) to class $\mathbf{C_{dest}}$<br>$\mathbf{C_{dest}}$ existed<br>All elements in **E** belonged to the same class ($\mathbf{C_{src}}$) in the previous release<br>Cohesion of $\mathbf{C_{src}}$ increased |

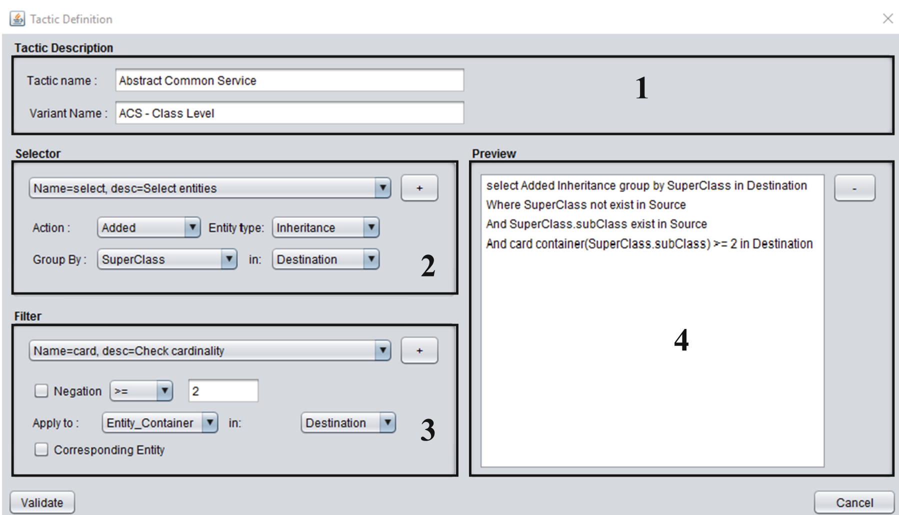


**Fig. 3.** Defining an operational representation of a tactic using TacMatch

specify the constraints on the selected elements; and (4) the preview zone that displays the tactic's specification in a form similar to an SQL query[1]. Figure 3 displays an example of the ACS tactic (i.e., the variant described in row 3 of Table 2) where multiple constraints were defined by the user using the filter zone (the "+" button enables to add a constraint at a time to the specification). These declarative specifications are used by our tool TacMatch to generate on the fly (when the user launches an analysis of a given system) the algorithm that retrieves the set of elements (from deltas) that match the tactic's application. This process is described in detail in Sect. 5.2.

## 5 Detecting Tactics Occurrences in Software Systems

Using the operational representations of tactics and two different versions of the software system under study, TacMatch supports the designer in detecting occurrences of these tactics in the system. To do so, TacMatch relies on MADMatch [16], a tool that enables diagram matching, to compute the deltas between two different versions of the same system. TachMatch uses the operational representation to generate on the fly detection algorithms for the tactics selected by the designer in the current analysis of the system. TachMatch executes these algorithms on the analyzed delta of the system and returns tactics' occurrences or cancellations. These occurrences can be used by the designer to carry out different types of analysis and to evaluate the architectural evolution of the analyzed system.

### 5.1 Computing and Storing Deltas Between Versions

Our approach relies on differencing tools able to supply our technique with elementary source code changes that we can then analyze, regroup and possibly match to architectural tactics. One such tool is MADMatch [16], which is a recent tool that takes as input graph representations of two different versions of the source code and generates the delta between these versions. In our case, these graphs represent class diagrams that were recovered using the Ptidej tool suite [18]. A generated delta describes the source code changes that occurred between the two analyzed versions (e.g., removed package, added package, added class, moved class, etc.). Deltas are serialized in CVS files. Our proposed tool TacMatch analyzes these CVS files to extract relevant information on the delta and saves this information in a database to which we will ultimately send customized queries to detect tactics' occurrences.

### 5.2 Detecting Tactics Occurrences

Given a generated delta from the system under study and a set of tactics chosen by the user for her current analysis, TacMatch retrieves corresponding tactics specifications and generates the corresponding detection algorithms on the fly and then execute them

[1] For lack of space, we do not discuss in this paper the predefined actions and constraints that TacMatch provides, nor the specification language used to describe the tactics.

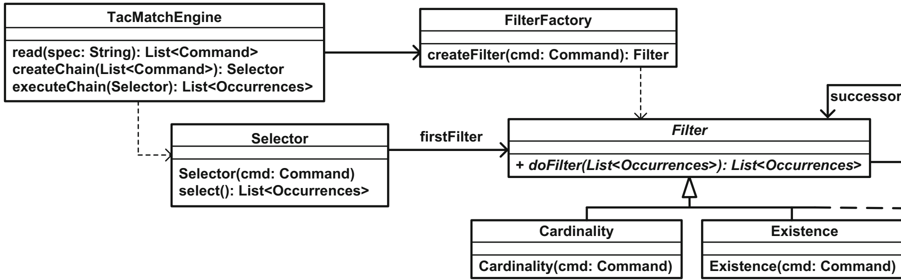


**Fig. 4.** Generating the detection algorithms using a chain of responsibility

on the delta. To generate the detection algorithms, TacMatch relies on a set of classes that read the specification of a tactic and generate different parts of the corresponding algorithm. Figure 4 gives an excerpt of the core classes of TacMatch, which were organized using the Chain of Responsibility (CoR) design pattern [19]. The Selector class enables to select occurrences of the changes undergone by the system and that correspond to those specified in the Select clause of the operational representation of the tactic (e.g., see the first line of the preview in Fig. 3). The Filter type defines an interface for filtering occurrences of the changes undergone by the system according to a given constraint; i.e., sub-classes of Filter implement different constraints. We used the CoR design pattern so that we can instantiate and configure, at runtime, the subset of filters that correspond to the constraints defined by the tactic at hand. Moreover, using the CoR design pattern makes it easy to add new filters (i.e., constraints).

TacMatch's entry point is the class TacMatchEngine which reads the tactic's specification as entered by the designer and generates a collection of commands corresponding to the lines of the specification. These commands are then used to create an ordered list of objects that starts with an instance of the Selector class followed by a chain of the appropriate subset of the filters. This is done using the createChain method which relies on the FilterFactory class to instantiate and set the appropriate filter for each command[2]. The appropriate selector object and chain of filters are instantiated and ordered in a dynamic way according to the operational representation of a tactic. This corresponds to generating on the fly the skeleton of the detection algorithm for the given tactic. For instance, given the operational representation described in the preview zone of Fig. 3, TacMatch generates a selector object that is set to retrieve inheritance relationships grouped by their superclass followed by a chain of two instances of the Existence filter[3] and one instance of the Cardinality filter.

The method executeChain enables execution of the detection algorithm related to a given tactic. This method takes as input the selection object corresponding to the tactic and it calls first the select() method of this object to retrieve the relevant occurrences of

[2] Both the Selector and the filter classes have their own fields which are set during their respective instantiation using the command parameter received by their respective constructor.

[3] In some tactics, the same filter class can be instantiated more than once using different parameters (i.e., commands). Moreover, we use a filter class to instantiate a constraint or its opposite depending on the tactic's definition.

changes from the delta. These occurrences are then sent to the first filter referenced by the selector object and from one filter to its successor in the chain; each filter filters the occurrences according to the constraint it implements (i.e., using the doFilter() method) and passes the resulting occurrences to its successor in the chain.

## 6 Case Study: Analyzing the Architectural Evolution Trend of JFreeChart

Our approach aims at mapping high-level descriptions of tactics to operational representations that can be detected at the source-code level, and inferring the architectural evolution trend of a software system by analyzing its available versions and detecting occurrences of the operational representations. To evaluate the effectiveness of our approach, we implemented a prototype tool that supports the definition and detection of operational representations and we conducted a retrospective case study using an open source software system.

In particular, the goal of our case study was to answer the following research questions:

- **RQ1: How effective is our technique at detecting applied tactics?** To answer this question, we used our prototype tool to analyze a number of versions of an open source Java system in order to detect a common set of the modifiability tactics. These tactics are: Split Responsibility (SR), Abstract Common Services (ACS), Use Encapsulation (UE) and Increase Cohesion (IC). We focused on these tactics as they involve actions that can be detected through static analysis of different releases of a software system. We computed the precision and recall of the obtained results by manually analyzing the changes made to the versions under study as reported by the differencing tool MADMatch.
- **RQ2: Are we able to derive an architectural evolution trend using our approach and interpret this trend at the architectural level?** To answer this question, we studied the detected applications and cancellations of tactics to check if the changes made to the system follow a comprehensible pattern of architectural evolution. We also compared the results of our detection process when applied to major releases versus minor releases versus revisions.

Our case study is focused on the analysis of JFreeChart, a Java open source framework which was previously studied in many publications, including the MADMatch paper [16]. JFreeChart is a library that supports developers in displaying various charts in their applications and it was used to develop a number of open-source and commercial products. We analyzed 37 versions of JFreeChart including revisions, minor and major releases starting from version 0.5.6[4] till version 1.0.6. The size of the analyzed versions varies from 26 to 141 packages and from 100 to 1196 classes.

[4] In this three sequence-based schema, the first sequence is the major number (incremented when there are significant changes to the system), the second sequence is the minor number (incremented when there are minor changes to the system or significant bug fixes) and the last sequence is the revision number (incremented when minor bugs were fixed).

### 6.1 Effectiveness for Detecting Architectural Tactics

Overall, using the 36 deltas generated from the 37 analyzed versions we detected 103 occurrences of tactics' applications and 33 tactics' cancellations. To compute the precision and recall of our results, we used the output of MADMatch to manually identify all the changes that correspond to true tactics applications or cancellations. Regarding the occurrences of tactics applications, we were able to confirm that 85 of these occurrences were true positives resulting in a precision of 82.52 %. We also identified 3 occurrences of tactics applications that our tool did not detect, resulting in a recall of 96.59 %. Interestingly, only 19 among the 33 occurrences of tactics cancellations were true positives, giving a precision of 57.57 % while manual analysis of MADMatch's output did not reveal any false negatives, resulting in a recall of 100 %.

These results suggest that our operational representations are effective in detecting the application of architectural tactics but may not be enough to automatically infer cancellations. Indeed, our simple technique for inferring the opposite evolution pattern from an operational representation of a tactic is not enough to precisely define the tactic's cancellation. The opposite evolution pattern may lead to a high number of negatives identified as positives (high recall and low precision) or to a misinterpretation of the appropriate tactic that was cancelled. For instance, during the transition from version 0.7.0 to version 0.7.1, the Separate Responsibility tactic was applied by moving a number of classes from the package com.jrefinery.chart into a new package com.jrefinery.chart.combination. However, during the transition from version 0.8.1 to version 0.9.0, the package com.jrefinery.chart.combination was deleted and its classes were moved back into two different packages (com.jrefinery.chart and com.jrefinery.data). This was recognized by our detection process as a cancellation of the Abstract Common Services tactic. Indeed, the SR tactic that was detected the first time was in fact part of the application of an ACS that was incrementally introduced through several transitions from versions 0.7.0 to 0.8.1 and then cancelled later in version 0.9.0. Future work is needed to define the relationships between operational representations so that we can aggregate and correctly interpret a number of successive applications of some tactics and thus define and appropriately trace cancellations to tactics.

To identify the factors that influence the effectiveness of our operational representations, we examined in detail the false results (i.e., false positives and false negatives) returned by our detection process. We uncovered that all these errors were due to the external tools MADMatch (85 %) for the deltas and PtiDej (15 %) for the reverse engineering of the project binaries. MADMatch sometimes returns incorrect matching in its deltas in part because its default parameters, which we used, promote recall over precision. We decided to leave these parameters unchanged in order to get more data for our manual analysis and thus a better approximation of the recall. Experimentation with different parameters is planned for future work.

### 6.2 Detecting Architectural Evolution Trends

Regarding our second research question, we investigated the applications and cancellations of tactics that were manually confirmed. Table 3 displays the distribution of both tactics applications and cancellations per deltas (i.e., the table displays true

positives). To reduce the size of the table, we have omitted the deltas that do not have any occurrences. In purely quantitative terms, if we consider the total numbers of the tactics that were applied (85) and those cancelled (19) through all the analyzed versions, cancellations represent 22 % of applications. We further investigated the observed cancellations to understand the causes of such a high percentage.

Our analysis revealed that out of the 19 cancellations of tactics, 11 cancellations were related to tactics already present in the first available release 0.5.6 while 8 cancellations are related to tactics that were introduced during the subsequent versions. For instance, in the revision from versions 0.9.16 to 0.9.17, the class org.jfree.chart.renderer.AbstractSeriesRenderer was introduced as a superclass for two other existing sub-classes but was deleted two revisions later (i.e., in 0.9.19). We also observed an interesting evolution pattern which involves the introduction, through different versions, of a number of super-classes that centralize a number of common constants and the deletion of these classes later in other versions. For instance, from 0.8.1 to 0.9.0, the classes CategoryPlotConstants and ChartPanelConstants (both in the package com.jrefinery.chart) were created to centralize a number of constants. CategoryPlotConstants was deleted later in the revision from 0.9.9 to 0.9.10 and its content was moved back to the class com.jrefinery.chart.CategoryPlot. Likewise ChartPanelConstants was deleted later in the transition from 0.9.20 to 1.0.0 and its content was moved to org.jfree.chart.ChartPanel. This tendency to apply and cancel tactics raises some questions about the consistency of the evolution of the system in general and its conformance to architectural decisions in particular. In fact, this could be construed as a motivational case for the importance of detecting architectural tactics and reminding them to developers (especially in open-source and collaborative settings) in order to prevent seemingly erratic modifications.

We also compared the results of our detection process when applied to the deltas from two successive minor (respectively major) releases versus those generated by the intermediate revisions between these minor (respectively major) versions. We presume that if the developer consistently evolves the system through the intermediate revisions between two successive minor (respectively major) versions, the aggregated results of our detection process through these revisions would lead to the same result than the one generated using the two minor (respectively major) versions. Table 4 displays the number of occurrences of both applications and cancellations of tactics generated from successive minor or major revisions. Similar to Tables 3 and 4 displays true positives and it omits minor and major releases for which no occurrences were found (e.g., from 0.6.0 to 0.7.0) and successive minor releases for which there was no intermediate revisions (e.g., from 0.5.6 to 0.6.0).

From 0.7.0 to 0.8.0, the only tactic occurrence (out of 7) that was detected in the delta between these two minor versions but not in the revisions between them, is an incremental application of the User Encapsulation (UE) tactic; i.e., a class (SignalsDataset) was created in 0.7.1 and an inheritance relationship was added later in 0.7.2 between this class and an existing subclass (SubSeriesDataset). As for the detected tactics applications and cancellations from 0.8.0 and 0.9.0 (i.e., 9 occurrences), they match the aggregated results of the detection when applied to the revisions from 0.8.0 to 0.8.1 and from 0.8.1 to 0.9.0. Finally, we found 34 occurrences of applications and cancellations of tactics from 0.9.0 to 1.0.0 which is a major revision.

**Table 3.** Number of tactics applied or cancelled per deltas generated from successive versions

| Delta | Application of tactics | | | | Cancellation of tactics | | | |
|---|---|---|---|---|---|---|---|---|
| | SR | UE | ACS | IC | SR | UE | ACS | IC |
| v0.5.6_v0.6.0 | 1 | 2 | 1 | | | | | |
| v0.7.0_v0.7.1 | 1 | | | | | | | |
| v0.7.3_v0.7.4 | 1 | 2 | | | | | | |
| v0.7.4_v0.8.0 | | 1 | | | | | | |
| v0.8.0_v0.8.1 | | 1 | | | | | | |
| v0.8.1_v0.9.0 | | 3 | 1 | 1 | 1 | 1 | 1 | |
| v0.9.1_v0.9.2 | | | 1 | | | | | |
| v0.9.2_v0.9.3 | | 1 | | | | | | |
| v0.9.4_v0.9.5 | 3 | 5 | 2 | | | 1 | | |
| v0.9.6_v0.9.7 | 1 | 4 | | 1 | | | | |
| v0.9.8_v0.9.9 | 1 | 1 | | 1 | | 6 | | |
| v0.9.9_v0.9.10 | | 1 | 1 | | | 1 | | |
| v0.9.11_v0.9.12 | 1 | 1 | 1 | 2 | | | | |
| v0.9.12_v0.9.13 | 1 | 2 | | | | | | |
| v0.9.13_v0.9.14 | | | 2 | | | 1 | | |
| v0.9.14_v0.9.15 | 1 | 1 | | | | | | |
| v0.9.15_v0.9.16 | 1 | | | | | | 1 | |
| v0.9.16_v0.9.17 | | 2 | | 5 | | | | |
| v0.9.18_v0.9.19 | | 3 | 2 | | | 2 | 1 | |
| v0.9.19_v0.9.20 | | | 1 | | | | | |
| v0.9.20_v1.0.0 | 9 | 3 | 2 | 4 | | 2 | 1 | |
| v1.0.2_v1.0.3 | | | 1 | | | | | |
| v1.0.4_v1.0.5 | | 1 | | | | | | |
| v1.0.5_v1.0.6 | | 1 | | | | | | |

**Table 4.** Number of tactics applied or cancelled per deltas generated from successive minor or major versions

| Delta | Application of tactics | | | | Cancellation of tactics | | | | Total |
|---|---|---|---|---|---|---|---|---|---|
| | SR | UE | ACS | IC | SR | UE | ACS | IC | |
| v0.7.0_v0.8.0 | 2 | 5 | | | | | | | 7 |
| v0.8.0_v0.9.0 | | 4 | 1 | 1 | 1 | 1 | 1 | | 9 |
| v0.9.0_v1.0.0 | 10 | 5 | 14 | 1 | | 4 | | | 34 |

However, the aggregation of the results from all the intermediate revisions between 0.9.0 and 1.0.0 yields 85 occurrences. We identified three main reasons for this discrepancy some of which were already discussed above. First, some tactics were applied through one or several revisions but all the entities involved in these tactics appear as

added in the major revision (i.e., the evolution pattern is visible through revisions but not at the major versions level). For example, the UE tactic was incrementally applied by adding a set of classes (e.g., ObjectList) in the revision from 0.9.9 to 0.9.10 and their superclass (AbstractObjectList) in the revision from 0.9.11 to 0.9.12. This whole evolution pattern is not detectable when we analyze the delta from 0.9.0 to 1.0.0; the entire inheritance hierarchy appears to be newly created at the same time. Second, some tactics were applied in an incremental way through changes spread over several revisions starting from the revision 0.9.0. These occurrences are only detectable when we analyze the delta from 0.9.0 to 1.0.0. Finally, as discussed before, several tactics were applied and then cancelled through the revisions; these tactics are not present at major versions level.

### 6.3 Threats to Validity

**External validity:** Our case study was carried out on a subset of the modifiability tactics that we were able to detect through static analysis of different releases of a software system. This is possible for most of the modifiability tactics and some other tactics such as exception handling (for availability) and creating additional threads or reducing the number of iterations (for performance). However, other tactics may require a dynamic analysis of the code or are even not present in the source code (e.g., increasing computational efficiency or maintaining multiple copies of data). Thus, our approach is limited to those tactics that have an observable impact on the source code. As future work, we plan to extend our work to other tactics and identify precisely the type of tactics to which our approach may be applied.

**Internal validity:** Some tactics (e.g., ACS) may be composed of several other more elementary tactics (e.g., SR). Since we did not implement yet a mechanism that enables to relate and aggregate detected tactics through a number of releases, we tend to interpret each detected tactic locally and individually. This may have an impact on our interpretation of the overall architectural evolution trend. Thus, as discussed in Sect. 6.1, future work is needed to define the relationships between operational representations and exploit these relationships to correctly aggregate and interpret a number of successive applications of related tactics. Finally, our results are dependent on the effectiveness of the other tools used, notably MADMatch that was used to compute the deltas. We selected MADMatch because it is a recent tool which compared favorably to other techniques [16] but other tools may provide different (better or worse) results. Future work is planned for experimentation with different parameters of MADMatch and different tools.

## 7 Conclusion and Future Work

In this paper, we present a first iteration of a tool-supported approach that allows the definition and detection of architectural tactics or more general evolution patterns using basic changes extractable from the differencing of software versions. Once these

architectural tactics or patterns are defined, our technique automatically generates algorithms able to parse the differencing data in order to detect occurrences of the application or cancellation of these tactics. A case study conducted on a well-studied open source system (JFreeChart) suggest that the technique is effective at detecting the occurrences of the application of defined tactics but is not as successful at detecting their cancellation. While few occurrences of these tactics are missed by our technique, there is some noise (lack of precision), especially for the detection of cancellations. Many of these errors are related to the parameterizing of the external tool selected to provide differencing data. Nevertheless, the study revealed many instances of cancellations of tactics that may be ill-advised and could have been prevented if the developers had access to the history and present of tactics involving the code they are working on or plan to work on.

The conclusions of this study are still preliminary and future work with case studies involving different parameters, tools and systems is needed to confirm our findings. Additionally, we intend to experiment with more evolution patterns and eventually discover desirable or harmful patterns through analyses of the change and defect proneness of the components they involve.

## References


1. Negara, S., Vakilian, M., Chen, N., Johnson, R.E., Dig, D.: Is it dangerous to use version control histories to study source code evolution? In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 79–103. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31057-7_5
2. Garlan, D., Barnes, J.M., Schmerl, B.R., Celiku, O.: Evolution styles: foundations and tool support for software architecture evolution. In: WICSA/ECSA, pp. 131–140 (2009)
3. Garlan, D., Schmerl, B.: Ævol: a tool for defining and planning architecture evolution. In: the 31st International Conference on Software Engineering, pp. 591–594 (2009)
4. Tonu, S.A, Ashkan, A, Tahvildari, L.: Evaluating architectural stability using a metric-based approach. In: CSMR 2006, 22–24 March 2006, p. 10, 270 (2006)
5. McNair, A., German, D.M., Weber-Jahnke, J.: Visualizing software architecture evolution using change-sets. In: WCRE 2007, 28–31 October 2007, pp. 130–139 (2007)
6. Abi-Antoun, M., Aldrich, J., Nahas, N., Schmerl, B., Garlan, D.: Differencing and merging of architectural views. ASE **15**(1), 35–74 (2008)
7. Breivold, H.P., Crnkovic, I., Larsson, M.: A systematic review of software architecture evolution research. IST **54**(1), 16–40 (2012)
8. Le, D.M., Behnamghader, P., Garcia, J., Link, D., Shahbazian, A., Medvidovic, N.: An empirical study of architectural change in open-source software systems. In: IEEE/ACM 12th Working Conference on Mining Software Repositories, Florence, pp. 235–245 (2015)
9. Eden, A.H., Kazman, R.: Architecture design implementation. In: 25th International Conference on Software Engineering, pp. 149–159 (2003)
10. Fairbanks, G.: Just Enough Software Architecture: A Risk-Driven Approach. Marshall & Brainerd, Boulder (2010)
11. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley, Boston (2003)
12. Bachmann, et al.: Modifiability tactics, CMU Software Engineering Institute Technical Report CMU/SEI-2007-TR-002

13. D'Ambros, M., Gall, H., Lanza, M., Pinzger, M.: Analysing software repositories to understand software evolution. In: D'Ambros, M., Gall, H., Lanza, M., Pinzger, M. (eds.) Software Evolution, pp. 37–67. Springer, Heidelberg (2008)
14. Kim, M., Gee, M., Loh, A., Rachatasumrit, N.: Ref-Finder: a refactoring reconstruction tool based on logic query templates. In: FSE 2010, Santa Fe, New Mexico, USA, pp. 371–372 (2010)
15. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston (1999)
16. Kpodjedo, S., et al.: Madmatch: many-to-many approximate diagram matching for design comparison. IEEE Trans. Softw. Eng. **39**(8), 1090–1111 (2013)
17. Aldrich, J., Sazawal, V., Chambers, C., Notkin, D.: Language support for connector abstractions. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 74–102. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45070-2_5
18. Gueheneuc, Y.G., Antoniol, G.: DeMIMA: a multilayered approach for design pattern identification. IEEE Trans. Softw. Eng. **34**(5), 667–684 (2008)
19. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
20. Barnes, J.M., Garlan, D.: Challenges in developing a software architecture evolution tool as a plug-in. In: 3rd International Workshop on Developing Tools as Plug-ins (TOPI), San Francisco, CA, pp. 13–18 (2013)