Patterns of Co-evolution between Requirements and Source Code

Mona Rahimi and Jane Cleland-Huang
School of Computing
DePaul University, Chicago, IL, 60604, USA
m.rahimi@acm.org, jhuang@cs.depaul.edu

Abstract—Software systems are characterized by continual change which often occurs concurrently across various artifact types. For example, changes may be initiated at the requirements, design, or source code level. Understanding patterns of co-evolution across requirements and source code provides fundamental building blocks for automating software engineering activities such as trace link creation and evolution, requirements analysis and maintenance, refactoring detection, and the generation of task recommendations. However, prior work has focused almost entirely on the evolution of individual artifact types such as requirements, design, or source code. In this paper we document patterns of co-evolution that occur between requirements and source code. We illustrate the utility of the patterns for detecting missing requirements and for evolving requirements to source code trace links.

Index Terms—Source code evolution, requirements evolution, patterns, co-evolution

I. INTRODUCTION

Software systems are composed of many different types of artifacts including requirements, design, source code, test cases, and bug reports. In a healthy environment, all of these artifacts evolve in tandem as the system matures [8]. In this paper we focus on the co-evolution of requirements and source code. Requirements describe the desired functionality and behavior of the system [25] while the source code represents what the system actually does at any point in time. It is desirable for source code to accurately reflect the functionality specified in the requirements and for requirements to reflect all features implemented in the code. Gold-plating or inclusion of undocumented features should be avoided [1].

Understanding the co-evolution of requirements and code has ramifications for a number of important Software Engineering activities. For example, it helps software trace links between requirements and source code to be accurately maintained, especially in safety-critical software systems [5]. Furthermore, a broad swathe of tools that assist developers by recommending specific tasks such as refactorings or updating design rationales [26] could be improved if both source code and requirements changes were taken into consideration.

Despite the potential benefits that could be realized through accurately co-evolving requirements and code, much of the prior work in this area has focused either on changes to requirements [15], [11] or on evolution of code [6]. For example, Martin Fowler lists over 70 different kinds of source code refactorings such as rename class, promote method, and hide method [24]. Similarly, Mäder et al., identified a set of change activities at the Object-Oriented design level which included refinements of associations, resolving many-to-many associations and association classes; moving, splitting, or merging an attribute, method, class, or package; splitting a class, component, or package; merging, and promoting a class to a component or an attribute to a class [17], [16]. On the requirements side, Cleland-Huang classified change events such as adding, modifying, inactivating, merging, refining, decomposing, and replacing requirements [4]. In each of these three cases, the emphasis was either on the evolution of source code, or on the evolution of requirements, but not on both.

In contrast, our focus in this paper is on the co-evolution of requirements and code. We identify five major classes of change, namely added functionality, deleted (obsolete) functionality, modified functionality, source code refactoring, and requirements modifications. The first three types of classes represent co-evolution of requirements and source code, while the last two represent independent changes within source code and requirements respectively. For each class of change we identify specific scenarios and document them in the form of 18 different patterns. These patterns provide the foundations needed to support activities such as automated trace link evolution, recommending the ‘next’ developer task, and identifying missing requirements.

We used several sources of information in order to formulate these patterns. First we included previously recognized patterns of change related to code refactorings [9], [27] and requirements change [4]. Second, we analyzed source code and requirements changes in several different open source systems. From these observations we propose the patterns described in this paper.

II. PATTERN STRUCTURE

We present the change patterns using the following structured format.

- **Name** and description of change class, e.g., Added functionality.
- **Triggers** of the change, e.g., Gold plating.
- **Impact on Requirements**: The potential and desired impact upon the requirements e.g., requirements reflect changes in code.
- **Impact on Code**: The observed refactoring that occurs in the code. For several of the change classes, multiple
patterns are identified. Each of these patterns is structured as follows:

- **Class level change** e.g. Added class.
- **Pattern description** e.g. The new class is added as a result of a new requirement being added and implemented.
- **Visual representation** of the pattern.

**Potential Usage:** A description of the ways in which identifying the change scenarios could benefit other software engineering activities.

In all sections below, \( V_i \) refers to the older version \( i \) before the change and \( V_{i+1} \) refers to the newer version \( i + 1 \) after the change. Similarly classes with subscript \( i \) refer to the classes in the older version \( i \) and classes with subscript \( i + 1 \) refer to the classes in the version after the change \( i + 1 \).

### III. Functionality Added

This first set of patterns describes changes related to the addition of new functionality. New functionality is ideally specified first in the requirements and then implemented in the code. However, in practice functionality may be implemented in the code first. Requirements may be updated after the fact, or may not be updated at all leading to missing requirements.

#### A. Triggers

- A new requirement is added and implemented in source code.
- A previously specified requirement is now implemented. Its status is updated accordingly.
- Needed functionality is added to the source code but not reflected in the requirements specification.
- Buggy code is fixed. Evidence is likely found in the bug repository and/or version control log. There is no direct impact upon requirements; however, the fixed code is expected to satisfy existing requirements.
- Unnecessary functionality is added to the source code and not reflected in the requirements specification. Gold plating has occurred.

#### B. Impact on Requirements

As a result of new functionality being added the requirements may be impacted in one of the following ways:

- A **new requirement** is created and added to the requirements specifications.
- The **status** of an existing requirement is changed.
- The requirements become **inconsistent** with the source code.

#### C. Impact on Code

The addition of new functionality can impact source in the following three primary ways as depicted in Figure 1:

- **Added Class:** New functionality may be introduced at the class-level granularity through the creation of a new class.
- **Added Method:** New functionality may appear at the method-level granularity in the form of a new method.
- **Modified Method:** New functionality may be introduced at a lower level of granularity through modifying the body of a method.

#### D. Potential Usage

Recognizing the addition of new functionality can be useful for supporting a number of software engineering activities:

- New trace links may need to be added between the new requirement and the code.
- If it is determined that new functionality has been added to the code without updating the requirements specification, the user could be prompted to provide the missing requirement.
- Design rationales may need to be updated to reflect design decisions behind the new functionality.
- In safety critical systems, the addition of new functionality triggers the need for a new safety analysis.

### IV. Functionality Deleted

Features may be deleted from code for several different reasons. Ideally, changes in the code are initiated by changes in requirements. However, in practice, features may also be removed from the code without requirements being updated.

#### A. Triggers

- A feature specified in the requirements is no longer needed or desired. Its status is changed to ‘obsolete’ or the requirement is removed from the specification. The related feature is deleted from the source code.
- Required functionality is accidentally or deliberately removed from the source code. Requirements are no longer (partially or completely) satisfied. This problem may occur as a side-effect of code maintenance efforts, especially if developers are not kept informed of the relationships between code and requirements.
- Gold plating is removed from the implementation. Individual requirements are unaffected, however, requirements coverage [28] increases.

---

Fig. 1: Three change patterns that might be observed in the source code when functionality is added. Thicker dashed circles represent added class/method while thinner ones represent modified method.
Fig. 2: Three change patterns that might be observed in the source code when functionality is deleted. Thinner dashed circles represent added class/method while thinner ones represent modified method.

B. Impact on Requirements

When the change in code is initiated by a change in requirements - the deletion of code preserves or achieves synchrony between requirements and code. When change is initiated in the code, then requirements must be updated accordingly if required.

C. Impact on Code

The removal and/or obsolescence of functionality can be observed in the source code in three primary ways. These are depicted in Figure 2:

- **Deleted Class**: The deletion of one or more classes.
- **Deleted Method**: The deletion of one or more methods.
- **Modified Method**: Modifications to one or more methods.

D. Potential Usage

Recognizing when features are deleted and/or requirements are made obsolete is important for the following reasons:

- Existing trace links may need to be made obsolete or deleted.
- The status of an existing requirement may need to be changed to obsolete.
- In a safety critical environment, safety cases may need to be checked to ensure that the deleted functionality does not impact safety.

V. FUNCTIONALITY IS MODIFIED

Modifying a requirement could result in features being added, deleted or modified in the code. We do not document this as a unique set of patterns but treat it as a special combination of additions and deletions of classes and methods, and of modifications to methods.

A. Triggers

- An existing requirement is modified and source code is updated accordingly.
- Source code is modified to reflect required changes that have been communicated informally. However, the original requirement(s) is not updated.

- Features that were previously implemented in the source code, perhaps due to gold-plating, are removed. There is no impact upon requirements.

B. Impact on Requirements

Ideally, modifications are triggered by changes in requirements. However, if source code is changed first, then requirements will need to be updated to maintain consistency.

C. Impact on Code

Modifications can result in a broad range of additions, deletions, and modifications to code at the class or method level.

D. Potential Usage

Same as for adding and deleting functionality. However, the level of difficulty of recognizing a series of additions and deletions and of subsequently performing related tasks can increase when changes are a result of modifications – causing problems such as architectural degradation [10] and bad code smells [2]. Therefore, the ability to deconstruct a series of modification edits into their constituent parts, by reverse engineering source code refactorings and requirements refinements is particularly important. Specific usage scenarios include:

- Updating trace links automatically
- Keeping developers informed of underlying functional requirements and quality concerns as they modify existing source code.
- Monitoring the impact of bug fixes versus the introduction of modified features upon the quality of the design.

VI. SOURCE CODE IS REFACTORED

Martin Fowler defines refactoring as “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior” [9]. While refactoring does not directly impact requirements, it is important to understand refactoring behavior and to differentiate it from behavior in which requirements and source code co-evolve.

A. Triggers

Refactoring may be triggered in a number of ways such as:

- Recognition that a bad smell has been introduced, for example: redundant code, excessive use of switch statements, excessive complexity.
- A bug fix that requires the developer to refactor in order to increase understanding of the code.
- As part of a code review.
- As part of a regularly scheduled activity (i.e. a developer dedicates 10% of her time to refactoring efforts.

B. Impact on Requirements

While refactoring does not directly impact requirements, it may be triggered by the degradation of code that occurs when features are added or modified. The challenge is to differentiate between pure refactorings and other change scenarios which necessitate requirements co-evolution.
C. Impact on Code

Fowler has identified over 70 different types of refactorings. Some of the major ones are visually depicted in 3 and include:

- **Extracted Class**: A new class is added to source code. The new class has been extracted from an existing one.
- **Merged Classes**: A new class is added to source code. The new class represents a merging of existing ones.
- **Promoted Method**: A new class is added to source code. The new class was created as a result of promoting method.
- **Extracted Subclass**: A new class is added to source code. The new class is an extracted subclass.
- **Extracted Superclass**: A new class is added to source code. The new class is an extracted superclass.
- **Divided Class**: A new class is deleted from source code. The deleted class has been divided into new classes.
- **Merged Classes**: Classes are deleted from source code. The deleted classes have been merged into a new class.
- **Extracted Method**: A new method is added to source code. The new method is an old promoted method.
- **Divided Methods**: A new method is added to source code. The new method represents a merging of old ones.
- **Divided Methods**: A new method is added to source code. The new method represents a division of the old method.
- **Divided Method**: A new method is deleted from source code. The deleted method has been divided into new methods.
- **Merged Method**: A new method is deleted from source code. The deleted methods have been merged into a new method.

D. Potential Usage

Refactoring source code, and reverse engineering refactorings, has numerous benefits [27].

- Ability to capture and replay changes in order to understand how and why a system has evolved to its current state.
- Ability to measure the impact of various types of changes upon system quality.
- Ability to differentiate between changes which are pure refactorings, bug fixes, or which represent real changes in functionality and should be reflected in changes to requirements. In the latter case, the benefits are described in the previously described patterns for add functionality, delete functionality, and modify functionality.

VII. ONLY REQUIREMENTS ARE CHANGED

It is common for requirements to be specified in advance of code implementation. For example in agile projects user stories are often written and then stored into a backlog. They are only implemented when selected for a specific iteration. Similarly, in other projects, an initial set of requirements are typically elicited and specified during a requirements phase. Requirements are then carefully prioritized and scheduled for release in a specific version or iteration. As a result, requirements may be added, deleted, and/or modified without any immediate impact upon the code.

A. Triggers

All activities are characterized by requirements-only changes, typically discovered through a requirements elicitation process [25]. Such requirements-only changes typically result in the creation of an initial software requirements specification.

B. Impact on Requirements

The following patterns of change were identified in prior work [4].

- A new requirement is added.
- A requirement is modified.
- Two or more requirements are merged.
- A new requirement is derived from an existing one.
- A requirement is replaced with another requirement.
TABLE 1: Requirement specifications for Domain Analysis application

<table>
<thead>
<tr>
<th>Req</th>
<th>Requirement Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>A document describing concepts in the domain shall be imported.</td>
</tr>
<tr>
<td>R2</td>
<td>A set of non-domain related documents shall be imported.</td>
</tr>
<tr>
<td>R3</td>
<td>All of words in the documents shall be stemmed to their root forms.</td>
</tr>
<tr>
<td>R4</td>
<td>The frequency of all nouns and noun phrases shall be computed.</td>
</tr>
<tr>
<td>R5</td>
<td>The domain specificity of each noun and noun phrase will be computed.</td>
</tr>
<tr>
<td>R6</td>
<td>A part-of-speech tagger will be used to tag all words.</td>
</tr>
<tr>
<td>R7</td>
<td>A list of domain terms shall be printed.</td>
</tr>
<tr>
<td>R8</td>
<td>Sort the output by DESCENDING Domain Specificity with subset of DESCENDING Frequency (ADDED BY DEVELOPER)</td>
</tr>
<tr>
<td>R9</td>
<td>Instead of listing all domain files individually – just point to a directory and read all files from the directory (ADDED BY DEVELOPER)</td>
</tr>
</tbody>
</table>

C. Impact on Code

There are no immediate impacts in the code. However, architectural design decisions may be made early in the design process and may constrain future coding decisions [20], [22].

D. Potential Impact

Depending upon the development process and philosophy, the discovery and specification of a new requirement could potentially lead to a re-evaluation of the architectural design. Changes in unimplemented requirements could lead to changes in the release plan.

VIII. USING THE CO-EVOLUTION CHANGE PATTERNS

The patterns presented in this paper provide the foundations for automating numerous software engineering activities. However, in order to accomplish this goal we need the ability to detect when a specific run pattern has occurred.

A. Detecting Change Patterns

We provide a brief discussion of how change patterns could be detected, and leave the implementation and evaluation for future work.

- **Changes in code** can be identified using one of the many refactoring detection tools [7], [13], [23].
- **Changes in Requirements** can be identified by analyzing deltas between versions of requirements and leveraging structural information (i.e., requirements numbering schemes) and text analysis to determine how the requirements have changed.

It is outside the scope of this workshop paper to describe these techniques in more detail or to demonstrate their use for recognizing various types of change. However, all of the above techniques have been demonstrated in prior work to detect source code or requirements changes, and/or to discover the relationships between them.

Fig. 4: Changes and their impact on code-to-requirement trace links done by a Java developer including: Addition of two classes in gray; Addition of 5 methods in bold and underlined; Deletion of crossed-out method; Change of five method signatures in bold and underlined.

B. Example of Evolutionary Change

For illustrative purposes, Figure 4 shows an example of refactoring changes made by a Java developer for an application called Domain Term Extraction and also the impact of these changes on code-to-requirement trace links. The refactored Java application consisted of 237 lines of Java code, four classes, 14 methods, nine requirements, and 11 requirements-to-code trace links. The application reads a set of domain-related files (e.g., documents about electronic health care records), and a set of general documents (e.g., books about business, astronomy, romance, etc), performed natural language part-of-speech tagging using QTag, and then outputs a list of domain-specific nouns and noun phrases. Refactoring changes illustrated in Figure 4 include: addition of gray classes; addition of bold and underlined methods; elimination of crossed-out methods; and the creation of new associations (dependencies) represented by solid arrows. In Figure 4, small black arrows represent trace links between classes and requirements. The ones marked with a check mark should be retained after the change, while those marked with an X should be made obsolete. Finally, bolded arrows without any annotation represent new links. Table I lists requirements for the Domain Analysis application. The two final requirements (R10 and R11) are added and implemented in the code by the Java developer during modification.
C. Maintaining Synchrony between Requirements and Source Code

The problem of loss of synchronization between requirements and source code occurs in both directions. In open-source systems, stakeholders often complain that they lose track of which feature requests have been implemented. In all types of system, code is sometimes updated without making corresponding changes to the requirements. This affects future impact analysis, scheduling, and coverage assessment activities [8]. By detecting non-refactoring changes in source code which lack corresponding changes in requirements, a recommender system could prompt the user to update the requirements accordingly.

The patterns therefore provide a fundamental building block for generating requirements recommendations. To clarify further, in our illustrative example of Figure 4, we could either monitor the development environment or perform an offline comparison between the old and new versions of code in order to detect changes in the code. The granularity at which the change is detected depends upon instrumentation strategies.

If the system detects that the new class TermComparator.java (class-level granularity) or the new method compare() (method-level granularity), responsible for sorting, has been added it must determine whether the change falls under the category of “Added Functionality” or “Refactoring tools” using a combination of the techniques described in Section VIII-A. If a non-refactoring change is identified but no related requirement is found, the system could recommend that the developer should add a relevant requirement describing the need to sort data. An example is provided in R10 (in table I).

D. Evolving Trace Links Automatically

A second useful application is the automated evolution of trace links. We provide two examples, covering both class and method level granularity. The first example is at class-level granularity. We assume that an existing set of pre-change trace links exists between classes and requirements while in the second example changes are detected at the method level and the assumption is that an initial set of trace links is provided between methods and requirements.

Class-level granularity: In Figure 4, after the system detects that a new class Program.java has been extracted from the old class NounsExtractor.java, the system can recommend that the developer creates two new trace links between related requirements $R_1$ and $R_2$ and the new class.

Method-level granularity: In this case, after detecting that the method getFileFromDirectory() is promoted to class Program.java, the system can recommend new trace links from the related requirements $R_1$ and $R_2$ to the method getFileFromDirectory().

IX. RELATED WORK

Most of the work in the area of code evolution is related to source code refactoring [14], [19], [18]. Martin Fowler lists over 70 different kinds of refactorings such as rename class, promote method, and hide method [24]. While the primary purpose is to help developers maintain and modify code, refactoring catalogs provide a useful vocabulary for describing changes in source code. Furthermore, several tools exist for detecting after-the-fact refactorings [12], [21]. Such tools can be used to capture the intent behind code changes by differentiating between bug fixes and refactorings, capturing and replaying changes to understand how and why a system has evolved to its current state, and for evaluating the impact of various changes upon system quality [27]. Such approaches focus almost entirely on changes to the code and do not take the co-evolution of requirements into account.

Cleland-Huang et al. identified patterns of requirements change [4], and integrated them into Event-Based Traceability (EBT) [3] which allowed users to register their interest in specific requirements and receive notification messages when various change events occurred. Specifically, the change events of adding, modifying, inactivating, merging, refining, decomposing, and replacing requirements were identified.

In more closely related work, Mäder et al. developed a tool and related algorithms for the semi-automated maintenance of trace links between requirements and UML class diagrams [17], [16]. They identified a set of change activities that occur in UML class diagrams including refining unspecified associations into directed associations, or into aggregation or composition associations; resolving many-to-many associations and association classes; moving, splitting, or merging an attribute, method, class, or package; splitting a class, component, or package; merging, and promoting a class to a component or an attribute to a class. They captured these activities through a series of 21 rules with 67 variants. Many of the rules are similar to Fowler’s refactorings, but applied to UML design instead of to code. However, the work focused on evolving existing trace links to UML classes, and did not explore patterns of co-evolution between requirements and code.

To the best of our knowledge, this paper is the first attempt to document patterns of co-evolution between requirements and source code.

X. CONCLUSION AND FUTURE WORK

In this paper we have identified and explicitly modeled 18 patterns of co-evolution between requirements and source code. These patterns provide the foundations for advancing several different research areas including automating the evolution of trace links, integrating requirements knowledge into the refactoring process, maintaining a closer integration between requirements and source code, and ultimately keeping requirements and source code synchronized. Documenting common types of co-evolution changes as patterns increases accessibility to other researchers.

While the identified patterns focus upon the object-oriented domain, we believe that many of the patterns are also relevant in structured domains at the file and function level. We leave examining this notion to future work.
ACKNOWLEDGMENTS

The work in this paper was partially funded by the US National Science Foundation Grant CCF:1319680.

REFERENCES