FORMALIZATION AND DETECTION OF COLLABORATIVE PATTERNS IN SOFTWARE

KULDEEP KUMAR

NATIONAL UNIVERSITY OF SINGAPORE

2015

FORMALIZATION AND DETECTION OF COLLABORATIVE PATTERNS IN SOFTWARE

KULDEEP KUMAR
M. Tech. (Distinction), Computer Engineering,
National Institute of Technology, Kurukshetra, India

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE

2015

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Kuldeep Kumar

08 June 2015

ACKNOWLEDGEMENTS

I would like to express my respect, deep sense of gratitude, and indebtedness to my supervisors, Dr. Stanislaw Jarzabek and Dr. Siau-Cheng Khoo for their guidance, encouragement, and support in every stage of my research work. I sincerely appreciate the best of opportunities they provided me, and credit their belief in my capability as an inspiration for achievement of my academic career. Their knowledge, kindness, patience, open-mindedness, and vision have provided me with lifetime benefits.

My deep sense of gratitude is due to the thesis advisory committee members, Dr. Bimlesh Wadhwa and Dr. Wei-Ngan Chin for their continuous blessings, guidance, encouragement, and feedback during the course of this work.

I also owe special thanks to Dan Daniel, Dr. Hamid Basit, and Sim Weiqiang for their wonderful help and feedbacks.

Thanks to my labmates and colleagues for their support, friendship, and all the fun we have had together.

I must mention my family members without whose love, nurturing, and support, I could never accomplish all these. I am thankful to my parents, brother, sister-in-law, niece, and sisters for their loving support and encouragement. I have no words to mention the support, patience, and sacrifice of my parents. With gratitude, I dedicate this thesis to my parents.

TABLE OF CONTENTS

SUMMARY

Code clones play a major role in software maintenance and reuse. Existing code clone detection techniques mainly focus on detecting similar code fragments, methods, functions, or files. But, many design-level similarity patterns appear as recurring configurations of collaborating components such as methods, functions, classes, or any physical or logical groups of program entities. We call such types of recurring configurations as collaborative patterns. Collaborative patterns often represent program structures exhibiting specific behavior meaningful to developers who need to understand programs, reengineer legacy code for reuse, or to refactor or simply maintain programs. Among others, detection of collaborative patterns can help in change impact analysis, code compaction, and creating generics. Unfortunately, unless manually documented, collaborative patterns remain implicit in code. In this thesis, in addition to properly define the concept of collaborative patterns, we present novel methods for detecting and managing them. The main novelty of the research lies in the formulation of the concept of collaborative patterns, in the development of the technique for detecting collaborative patterns, and in the development of the technique for managing such patterns in software systems.

In the thesis, we first formalize the concept of collaborative patterns. We show possible classification of collaborative patterns. Next, we present an approach for detecting these collaborative patterns. The proposed approach for detecting collaborative patterns enhances the value of similarity analysis in the activities such as software maintenance and in the process of re-engineering for reuse that involves finding candidate modules for reuse in legacy code. Collaborative patterns are higher-level clones of large granularity that can be detected by systematically combining small-granular cloned program entities at various levels. They signify use of standardized solutions and/or repetitions that arise naturally in software analysis or design space. As such, collaborative patterns often embody important design information. Since, many existing low-level clones of small granularity are grouped around these high-level repetitions. Therefore, collaborative patterns form a convenient conceptual window for developers to understand overall cloning in software systems. We implemented the proposed approach for collaborative pattern detection into a tool called CoPaD (Collaborative Patterns Detector). We performed experimentation to evaluate the proposed approach. Finally, we propose a methodology to manage such types of high-level clones of large granularity (collaborative patterns as well as other large-granular code clones) by presenting a meta-programming technique and tool, the ART (Adaptive Reuse Technique). It manages families of redundant software systems by providing a common base of non-redundant, adaptable, and reusable meta-components called ART templates. We also evaluated the benefits of managing clones using the ART.

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AOP	Aspect-Oriented Programming
ART	Adaptive Reuse Technique
AST	Abstract Syntax Tree
CoPaD	Collaborative Patterns Detector
cpp	C preprocessor
FOP	Feature-Oriented Programming
GST	Generalized Suffix Tree
GUI	Graphical User Interface
JBD	Journaling Block Device
LCP	Longest Common Prefix
LOC	Lines of Code
MDSOC	Multi-Dimensional Separation of Concerns
MPC	Method Percentage Coverage
MTC	Method Token Coverage
NERF	Non-Extendible Repeat Finder
PCE	Project Collaboration Environment
PDG	Program Dependency Graph
SPC	ART Specification File
SPL	Software Product Line
STL	Standard Template Library
XVCL	XML-based Variant Configuration Language

Chapter 1.

INTRODUCTION

Software reuse is possible through various systematic means such as design patterns, generative programming, component frameworks, program libraries, object composition, and aspect-oriented software developments. It accelerates development process, increases dependability, and reduces development cost [1]. However, because of programmers' limitations and time constraints, sometimes it is needed to reuse software components which have not been designed for reuse [2]. It leads to the use of cut-and-paste programming style instead of system-redesign approach, causing increased maintenance cost [3]. Further, many programming languages lack with inherent support of reuse, resulting in code clones in software systems [3]. Code clones are repeated program structures of considerable size and significant similarities occurring in various forms at different locations in the software system [2].

Detecting code clones (i.e., similar program structures) helps the programmers in reducing maintenance cost, in improving program understanding, and in controlling code changes [2, 3]. Hence, many types of code clones and their corresponding detection techniques are available in the literature (discussed in

Appendix A). In this thesis, we proposed another useful type of code clones, we call them collaborative patterns. The knowledge of collaborative patterns in a software system can lead to better understanding of the design of the system, which helps in day-to-day software maintenance, long-term evolution, and re-engineering [4, 5]. Further, management of these clones with generic program structures can offer interesting opportunities for program simplification and reuse. In this thesis, we first formalize the concept of collaborative patterns along with its possible classification, and then discuss the design methodologies for their detection and management.

This chapter is organized as follows: we begin with background (Section 1.1), problem description, and motivation behind the problem (Section 1.2). Section 1.3 outlines the current status of clone detection research and highlights possible research gaps we address in the proposed research. Open challenges we face in dealing with the proposed work are discussed in Section 1.4. The scope and contributions of the work are presented in Sections 1.5 and 1.6 respectively. Finally, outline of the thesis is given in Section 1.7.

1.1. Background

Cloning exists in almost all kinds of software [6, 7]. Sometimes it is due to programmers' implicit activities like reusing a similar design solution to solve similar types of problems, or sometimes programmers explicitly use same code to save their time. Lack of knowledge in the problem domain, programming language constraints, change in the requirements, and inefficiently using software-reuse mechanisms are some of the other contributing factors that lead to code clones within software systems [8]. With

the advancement of technologies such as product line engineering [9], cloning has spread across multiple systems. It is because many systems developed using product line engineering tend to be similar, resulting in code clones not just within a system, but also across the systems.

Although cloning may not always affect the code functionality, it may have severe impacts on the maintainability, reusability, and quality of the software [3, 10]. There is no consensus on whether benefits of cloning outweigh its detriments [11]. Some literature considered cloning to be harmful because it may make code complex [3], error-prone [12-14], and difficult to change [15]. On the other hand, some works have not found any empirical evidences of harmful effects of cloning [16], and instead claimed cloning as one of the valuable software engineering practices [17, 18]. Nevertheless, whether clones are good or bad is still an open research question, and it is agreed on that clones must be made explicit in the code so that they can be consistently managed and maintained [19]. The knowledge of clones in the code assists programmers in program understanding, detecting library candidates, refactoring, and program analysis [2, 3]. Various forms of code clones are mentioned in the literature depending upon sizes and similarities among cloned code fragments. Following the classification from [2], broadly there are two kinds of similarities between code fragments: textual similarity and functional similarity. If the two code fragments are similar based on the similarity of their program text, they are considered textually similar. On the other hand, if code fragments are similar in their functionalities without being textually similar, it refers to the functional similarity between code fragments. In case of collaborative patterns proposed in the thesis, the corresponding

structures are similar to each other based on the similarity of their program text. Hence, the collaborative-pattern similarity that we address in this thesis falls within the category of textual similarity. Next section describes the problem addressed in this thesis with a motivating example.

1.2. Problem Description and Motivating Example

In this thesis, we aim at detecting an important type of clones in software systems we called *collaborative patterns* that have not been addressed in the software clone research so far.

Collaborative pattern *is defined as a recurring configuration of program entities (e.g., classes or methods) inter-related by means of calling relationships (method calls or message passing). In these configurations, the corresponding entities should be similar to each other based on some selected similarity metrics*.

The selected similarity metrics may be based on the textual similarity or the functional similarity among the program entities. While we may consider any types of similarity metrics, in our current work we focus on the textual similarity among the program entities.

Motivating Example: The motivation for the proposed work comes from the review of an existing web-based application, Project Collaboration Environment (PCE) [20, 21], that supports project planning and execution. PCE supports modular design to manage information about staff, project, product, or task independently. Each PCE module implements operations such as create, edit, delete, display, or save to manage their respective records. In PCE, we encountered examples where cloned modules are collaborating with

each other. For example, Figure 1.1 shows design of features CreateStaff, CreateProject, and CreateProduct. Boxes are PHP files implementing user interface (Level 1), business logic (Level 2), and database aspects (Level 3) of respective features. Boxes of the same shade are similar one to another (i.e., code clones), and arrows indicate calling relationships among PHP functions in the corresponding boxes. Similar to other cloned program structures, knowing this relation between the cloned modules may prove to be useful in better understanding of the system design or may help in finding clone candidates that can be unified at meta-level [4, 22]. In addition, it helps in tracing important calling-relationship information among the cloned modules, which can be useful in finding information flow in the system [5]. It motivated us to work for techniques for detecting such type of clones in software systems, we called collaborative patterns.

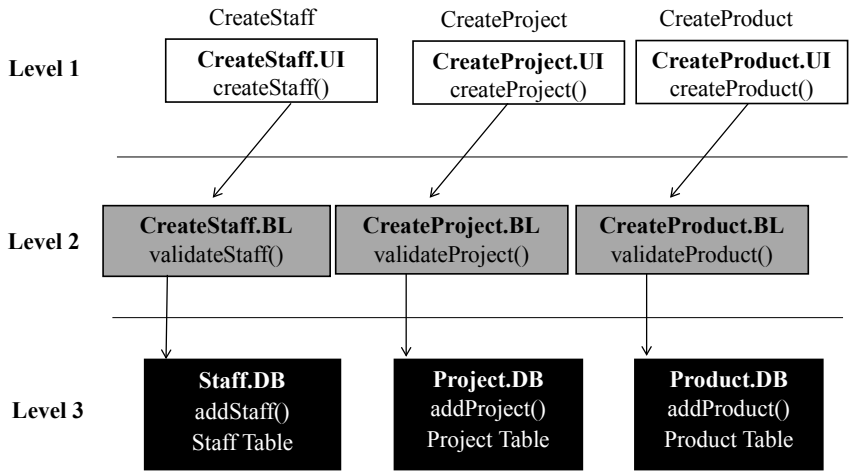


Figure 1.1. Motivating example: collaborative pattern in the PCE

1.3. Current Status of Research on Clone Detection

Much work has been done on software clone detection ranging from the detection of low-level small fragments of duplicated code [12, 23-80] to

higher-level duplicated program structures [5, 81-99]. A detailed discussion on the available literature is presented in Appendix A.

Most of the existing clone detection techniques are limited to detection of low-level small fragments of duplicated code. These techniques detect exact [23, 24], parameterized [25-39], gapped [12, 40-68], or semantically similar code fragments [69-80] by using various techniques such as by analyzing the program text [23-25, 27, 40, 44, 49, 52, 55, 56, 72, 75], by considering the program as a stream of tokens [12, 29-31, 34, 37, 38, 42, 43, 48, 62, 63, 65, 67], using program metrics [60], using abstract syntax trees [28, 41, 45, 47, 50, 53, 76, 77], using program dependency graphs [58, 59, 68-71, 78], using parse tree [46], or using hybrid combination of various program representations [36, 39, 54]. Other techniques address higher-level clones such as structural clones [5, 81], logical clones [82], design patterns [83-94], API usage patterns [95, 96], and others [97-99].

Small-granular code clone detectors generally detect clones larger than a certain threshold value (e.g., 30–50 tokens, 4–6 lines of code, or threshold set by users [2, 3, 42, 100, 101]). Motivated from the concept of structural clones (recurring patterns of duplicated contiguous code fragments), we found that more interesting and useful types of clones can be found by increasing the level of similarity analysis and clone granularity. We called these types of high-level clones as collaborative patterns.

What makes collaborative patterns interesting and useful is that many design-level similarity patterns are implemented as groups of collaborative components such as methods, functions, classes, files, or any physical or logical groups of program entities (Section 1.2, see motivating example). In

addition, standard solutions used across project groups in a company can often be expressed as collaborative patterns [20, 21].

1.3.1. Research Gaps

The research gaps we address in this thesis with their expected solutions are:

- *Type of the code clones detected:* This thesis aims at detecting patterns of collaborative components, so-we-called collaborative patterns. The proposed work initiates a new direction of research in the area of software clone detection by allowing us to detect collaborative patterns. Collaborative patterns are high-level clones. No work addresses such types of code clones.
- *Granularity (size) of the detected code clones—bigger units of clone:* Collaborative patterns are large-granular clones. Hence, they represent program structures exhibiting specific behavior meaningful to developers who need to understand programs, reengineer legacy code for reuse, or to refactor or simply maintain programs [4].

1.4. Open Challenges

Some of the challenges we face while dealing with collaborative patterns are:

Definition of collaborative patterns: A precise definition of collaborative patterns is required with its related terminology. It helps in better understanding of the proposed phenomenon and communicating it effectively with the current research.

Classification of collaborative patterns: There is a need to properly classify the collaborative patterns into different types at different levels of abstraction.

Having knowledge of varying types of collaborative patterns serves many purposes such as studying the more frequently occurring patterns or prioritizing different types of patterns.

Detection of collaborative patterns: Since there is no background work related to the detection of such types of code clone, another big challenge is the proposal of a detection technique which is scalable as well as efficient. We face many challenges related to the detection process:

- The technique to detect collaborative patterns itself is a big challenge. Moreover, different types of collaborative patterns may require different detection techniques.
- Correctness and completeness of the technique.
- Meaningfulness of the detected patterns for the analyst, designer, or implementer.
- Scalability of the technique.

Management of collaborative patterns: Once we found collaborative patterns—the questions arise how we are going to manage them, how can we benefit from their knowledge in terms of easier maintenance or better reuse. Which kinds of project activities can benefit by detecting and managing them? Overall, management aims at organizing existing clones, minimizing their negative effects, controlling their growth and dispersal, and avoiding them altogether [102].

1.5. Research Scope

The scope of the thesis lies in the following directions:

Concept Formalization: We formalized the notion of collaborative patterns in general and described various types of collaborative patterns possible in software systems.

In the area of Clone Detection: We proposed an approach that uses structural clones and method calling-relationship information from the source code of the subject program for detecting collaborative patterns [103, 104]. We implemented the above approach as a prototype tool, called CoPaD (Collaborative Patterns Detector). We performed experimentation to evaluate the usefulness of the proposed approach.

In the area of Clone Management: We proposed a methodology to manage code clones of large granularity (such as collaborative patterns, structural clones, or other large-granular cloned program structures) by presenting a meta-programming technique and tool, the ART (Adaptive Reuse Technique) that can manage families of redundant software systems by providing a common base of non-redundant, adaptable, and reusable meta-components [105-108]. We evaluated the benefits of managing code clones using the ART.

1.6. Research Contributions

Clones convey important information to the developers regarding the structure and the functionality of a system. It makes clones very useful in software maintenance and re-engineering. This thesis extends the clone research from lower-level cloned code fragments to higher-level collaborative pattern. Since collaborative patterns are high-level clones of large granularity, they may indicate a cloned concept, e.g., a cloned design solution [4]. Hence, they signify use of standardized solutions and/or repetitions that arise naturally in

software analysis or design space. Many existing lower-level code clones are grouped around such high-level repetitions [4]. Therefore collaborative patterns form a convenient conceptual window for developers to understand the overall cloning in software systems. Hence, the proposed techniques for detecting and managing collaborative patterns enhance the value of similarity analysis in software maintenance and in the process of re-engineering the software for reuse that involves finding candidate modules for reuse in legacy code.

The novelty of the research lies in the type of clones detected, and the techniques developed for detecting and managing them. The proposed work initiates a new direction of research in the area of software clone detection. To our best knowledge, no work addresses collaborative patterns. We used program execution traces for detecting collaborative patterns in the software systems. Generation and analysis of program execution traces have been used in other areas of research (such as monitoring of software for reliability reasons or in specification mining [109]), but the proposed work harnesses the use of program execution traces for clone detection.

1.7. Thesis Outline

The thesis is divided into seven chapters and two appendices.

Chapter 2 provides details on the formalization of the notion of collaborative patterns. The term collaborative pattern is defined precisely in terms of a directed graph. In the directed graph, nodes are program entities and edges are calling relationships among the program entities. This chapter briefly outlines the methodology for detecting and managing collaborative patterns.

Chapter 3 describes collaborative pattern detection approach in detail. Experimentation results related to the proposed approach are presented in Chapter 4.

To evaluate the benefits of knowing collaborative patterns (and other code clones of large granularity) in programs, we focus on representing clone classes using templates that can be built using the ART. The ART and the methodology of managing code clones using the ART are explained in detail in Chapter 5.

Chapter 6 quantitatively and qualitatively evaluates the strengths, weaknesses, and trade-offs involved in the application of the ART. Finally, Chapter 7 concludes the thesis.

Appendix A provides a comprehensive literature survey on relevant prior work. This survey gives us rudimentary details of state-of-the-art works available in the area of software clone detection. Appendix B at the end gives detail of general terms used in the thesis.

Chapter 2.

THE CONCEPT OF COLLABORATIVE PATTERNS

Clone detection is an active area of research in Software Engineering since about last two decades. As suggested by the literature, various tools and techniques have been proposed for detecting cloned code fragments. Also, some works addressed clones of larger granularity such as cloned methods or cloned files [5, 97, 98, 110, 111]. However through similarity analysis, we observed that cloning in software systems is not limited only to cloned code fragments, methods, or files—as is the focus of most of the current clone detection research—but can also occur at higher levels. One of such cases is the recurring configuration of collaborating program entities (such as methods, classes) where the corresponding entities in the instances of the configuration are code clones of each other. This chapter describes the concept of these higher-level clones, which we call collaborative patterns.

This chapter is organized as follows: Section 2.1 discusses the motivation. In Section 2.2, we formally define the term collaborative pattern in terms of a

graph and its relation with structural clones. Related work is presented in Section 2.3. Section 2.4 presents the classification of collaborative patterns. Section 2.5 highlights importance and benefits of collaborative patterns. Section 2.6 outlines the proposed approach for detecting and managing collaborative patterns that is explained in detail in the forthcoming chapters. Finally, Section 2.7 concludes the chapter.

2.1. Motivation

In the Second International Workshop on Detection of Software Clones (IWDSC'03), 57 open questions related to clone detection research were raised during the brainstorming session [112]. A few of them were related to detection of higher-level clones, for example, "Can we detect higher-level clones well?" or "Do we understand "other" level clones?". There have been a few attempts in this direction of research [5, 82, 97, 98, 110, 111].

We found that certain types of configurations of cloned code fragments or program entities signify some higher-level patterns of similarities. One such case is the recurring configuration of collaborating program entities such as methods, classes, or files, where the corresponding entities in the instances of the configuration are clones of each other. We call such configurations of collaborating program entities as collaborative patterns.

Detection of collaborative patterns can enhance the values of similarity analysis. The knowledge of the locations of collaborative patterns in the software system may lead to a better understanding of the design of the system, which can help in day-to-day software maintenance, long-term evolution, reuse, and re-engineering. Further, management of these patterns

with generic program structures can offer interesting opportunities for program simplification and reuse.

2.2. Introduction to Collaborative Patterns

In Section 1.2, we defined the term collaborative pattern as:

Collaborative pattern is defined as a recurring configuration of program entities (e.g., classes or methods) inter-related by means of calling relationships (method calls or message passing). In these configurations, the corresponding entities should be similar to each other based on some selected similarity metrics.

In this section, we precisely define and formalize the term collaborative pattern in detail.

2.2.1. Preliminary Definitions

This subsection details definitions of some of existing important terms which are used in formalizing the term collaborative pattern.

Definition 2.1 (program entity): In general, a *program entity* represents any program element that can be clearly identified in a program such as a statement, code fragment, function, class method, class, source file, directory, module, subsystem (last two are designated groups of files and/or directories).

Definition 2.2 (clone relation): A *clone relation* exists between two program entities *e1* and *e2*, if and only if they have significant similarities between them. The threshold of the similarity depends on the context and the nature of the program entities. Beside this, human judgment is also an important factor in deciding whether two program entities are clones of each other or not.

Similar to [42], the clone relation defined is an equivalence relation (i.e., reflexive, transitive, and symmetric relation).

Definition 2.3 (clone pair): For a given clone relation, a pair of program entities is called *clone pair* if a clone relation holds between the two program entities.

Definition 2.4 (clone class): An equivalence class of the clone relation is called *clone class*. It means that a clone class is a maximal set of program entities in which a clone relation holds between any pair of the program entities.

Definition 2.5 (simple clone): Segments of contiguous code are the simplest type of program entities that can participate in a clone relation, in such a case called as *simple clones* [5].

Definition 2.6 (program structure): Following definitions from Basit and Jarzabek [113], a *program structure* is a *connected mixed multigraph* where nodes are program entities, and (directed or undirected) edges are relationships between the program entities. A relationship represents any meaningful physical or logical connection between two program entities in a structure. Multiple edges between same pair of nodes can be useful in characterizing certain types of structures.

To define *program structure hierarchies*, Basit and Jarzabek [113] introduced the terms *atomic entity* and *abstract entity*. An *atomic entity* (**Definition 2.7)** is one whose internal structure has no relevance. On the other hand, an *abstract entity* (**Definition 2.8**) is one whose internal structure is abstracted away to form a building block for higher level structures. In this way, abstract entities

allow us to define higher-level program structures in the hierarchy in terms of lower-level program structures at as many levels as is useful.

Definition 2.9 (structural clone relation): A *structural clone relation* holds between two program structures S1 and S2 if (and only if):

(a) S1 and S2 have the same graph structure,

(b) A *clone relation* has already been established between corresponding program entities in S1 and S2, and

(c) Corresponding relationships in S1 and S2 are of the same type.

Depending upon the nature of program entities, a *static relationship* or *dynamic relationship* can exist among program entities. A static relationship can be program entities belonging to *same location* (e.g., functions defined in the same source file, methods belonging to the same class, or files in the same directory). A dynamic relationship can be *calling relationship* among program entities (e.g., a method calling another method, any method from a class calls some other method(s) of another class). The work on *structural clones* by Basit and Jarzabek [5] is a special case where relationship is the "same location" of interrelated program entities. *Collaborative patterns* described in this thesis are another special case where relationship is the "calling relationship" among the interrelated program entities.

Further, whether a relation is *symmetric or not*, it depends of the nature of the relation. For example, "same location" is a symmetric relation (if entity e1 is in same location as entity e2, e2 is also in same location as e1), but "calling relationship" is not (entity e1 is calling e2 does not implies that e2 is surely

calling e1). The rest of the section elaborates the collaborative pattern concept in terms of a directed graph.

2.2.2. Collaborative Structure

Definition 2.10 (collaborative structure): A *collaborative structure* is a *connected* and *directed* graph where nodes are program entities, and edges are calling relationships among the program entities.

Figure 2.1 gives an example of collaborative structure S_1 that consists of five program entities e1, e2, e3, e4, and e5. The five program entities are inter-related to each other by a calling relationship denoted by arrows.

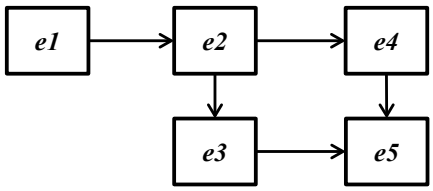


Figure 2.1. An example of a collaborative structure S_1

2.2.3. Collaborative Clone Class

Definition 2.11 (collaborative clone relation): *Collaborative clone relation* is a clone relation between collaborative structures. A collaborative clone relation exists between two collaborative structures S_1 and S_2 if and only if:

(a) S_1 and S_2 have the same graph structure, and

(b) A clone relation has already been established between the corresponding program entities in S_1 and S_2.

Definition 2.12 (collaborative clone class/ collaborative pattern): *Collaborative clone class* is a maximal set of collaborative structures that are

in collaborative clone relation of each other. In the rest of the thesis, we will call collaborative clone class a *collaborative pattern* for short.

Figure 2.2 shows some illustrative examples of collaborative structures S_2 to S_{13}. Each program entity is labeled with an entity name and represented by a rectangular box. Also, program entities represented by same color boxes are in clone relation with each other. For example, program entities $e1_2$, $e1_3$, $e1_4$, $e1_5$, $e1_8$, and $e1_9$ are in clone relation with each other, but $e6_{13}$ is not in clone relation with the rest.

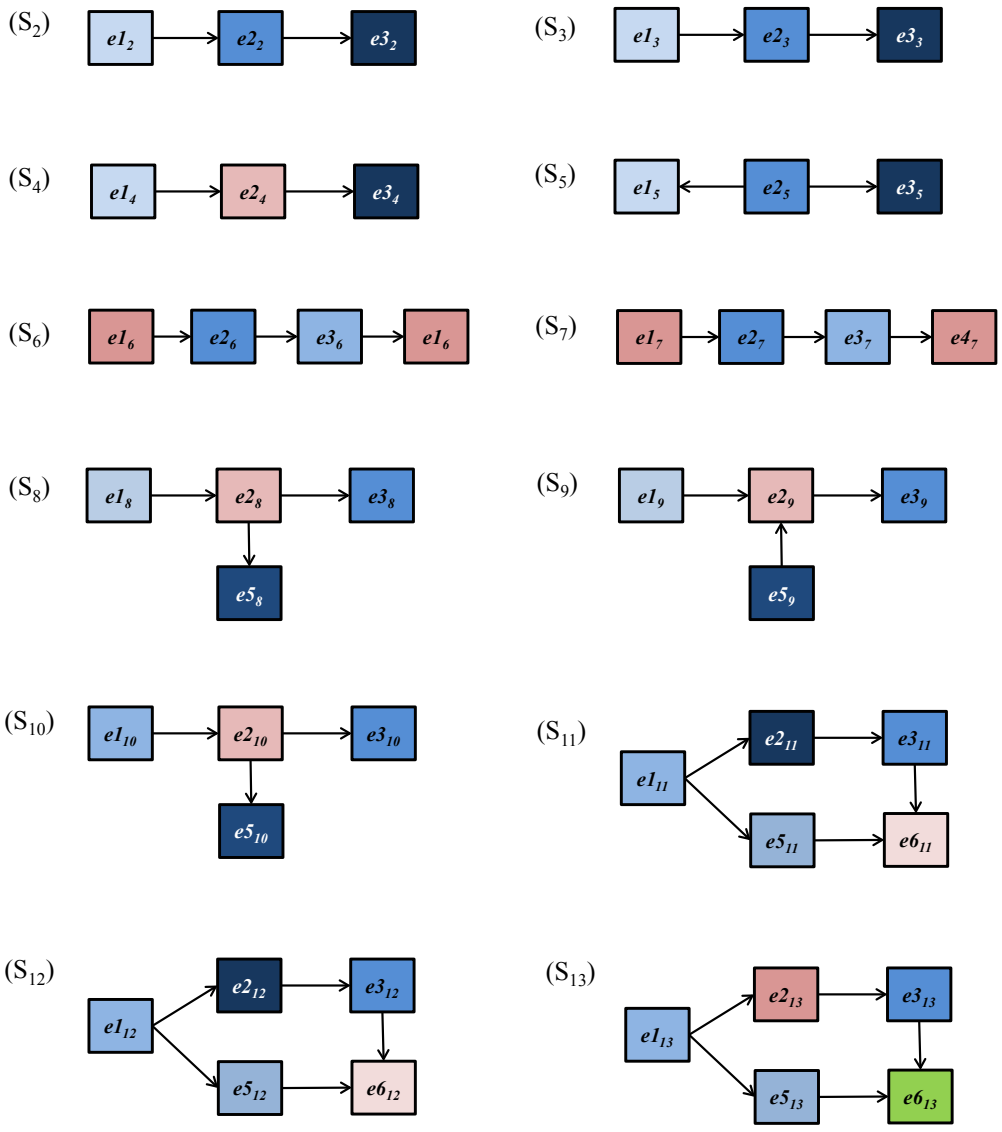


Figure 2.2. Examples of collaborative structures S_2–S_{13}

As shown in the Figure 2.2, collaborative structures:

a. S_2, S_3, and S_4 have same graph structure.
b. S_6 and S_7 have same graph structure.
c. S_8 and S_{10} have same graph structure.
d. S_{11}, S_{12}, and S_{13} have same graph structure.

Similarly, corresponding program entities in collaborative structures:

i. S_2, S_3, and S_5 have a clone relation.
ii. S_6 and S_7 have clone relation.
iii. S_8, S_9, and S_{10} have clone relation.
iv. S_{11} and S_{12} have clone relation.

Based on that, collaborative structures S_2 and S_3 have collaborative clone relation and form a collaborative pattern. But, S_4 has no collaborative clone relation with S_2 and S_3 (condition 'b' in definition 2.11 fails). Similarly, S_5 is not in collaborative clone relation with S_2 and S_3 (condition 'a' fails).

For collaborative clone relation, it is not necessary for the program entities to be unique. For example, in the collaborative structures S_6 and S_7, program entity $e1_6$ corresponds to both entities $e1_7$ and $e4_7$. But, by satisfying conditions 'a' and 'b', these collaborative structures form a collaborative pattern.

Given that corresponding program entities in S_8, S_9, and S_{10} have clone relation. But, S_9 is not in collaborative clone relation with S_8 and S_{10} (i.e., condition 'a' fails). Thus, only program structures S_8 and S_{10} form collaborative patterns. Similarly, beside collaborative structures S_{11}, S_{12}, and S_{13} have same graph structure, but only S_{11} and S_{12} form collaborative

patterns. It is because, S_{13} has a program entity $e6_{13}$ which is not in clone relation with the corresponding program entities in S_{11} and S_{12} (i.e., condition 'b' in definition 2.11 fails).

2.3. Related Work

The proposed collaborative pattern concept is built upon the concept of structural clones described in [5, 81]. Following works in [5, 81], similar program parts are termed as software clones. Software clones may include simple clones and structural clones. Simple clones are formed by fragments of textually similar contiguous code whereas structural clones are formed by configurations of these simple clones.

Structural clones are cloned program structures whose respective elements (i.e., program entities) are similar and relationship is the "same location" of interrelated program entities. Based on the nature of the program entity, structural clones may exist at different levels of granularity [114]. Structural clones at higher-level of granularity can be constructed from structural clones of lower granularity. For example, structural clones at file level can consist of method-level structural clones, where methods are the abstract entities whose internal structure is abstracted away to form a building block for higher-level structures. Code fragments are atomic entities in case of structural clones. Based on the idea of structure hierarchy in terms of atomic and abstract entity, structural clones can be defined at many levels as it is useful.

Collaborative patterns discussed in this thesis enhance the clone phenomenon further by exploring the calling relationships among these cloned structures. In case of collaborative patterns, instead of contiguous code fragments, we

considered methods as the atomic entities. For example, suppose that we have three structures (A1, B1, C1), (A2, B2, C2), and (A3, B3, C3) as shown in Figure 2.3. Each of these structures consists of methods as shown by the rectangular boxes. Suppose a substantial part of each of the corresponding methods in the three structures is covered by contiguous cloned code fragments. Then, A1, A2 and A3 are considered to be method-level structural clones as shown be same shade in the corresponding rectangular boxes. Similarly, further suppose we have other method-level structural clone <B1, B2, B3> and <C1, C2, C3>. We consider a group of such collaborative structures as a collaborative clone class (or collaborative pattern for short). Thus, collaborative pattern is formed by a group of collaborative structures whose respective elements are similar and inter-related by means of calling relationships. Hence, collaborative patterns are higher-level clones that can be found by further increasing the level of similarity analysis and clone granularity.

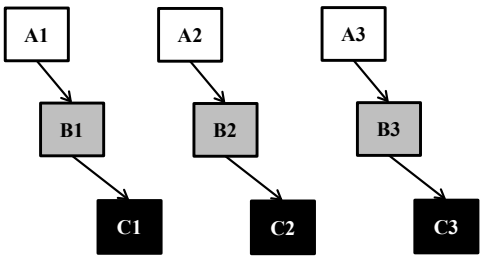


Figure 2.3. Collaborative pattern as high-level pattern of collaborative structures

A recent work, Clonepedia [19] targets on mining commonalities of syntactic contexts in which code clones occur. Similar to the work on structural clones, they also considered spatial relationships such as “contain”, “reside_in”, “diff_use”, “extend” in the program structures. These relationships describe location and usage characteristics of cloned code fragments. However,

depending on the relationship types, atomic entities may be classes, methods, or code fragments. Compared to the Clonepedia work, collaborative patterns described in the thesis are cloned program structures with calling relationship between the corresponding program entities.

There are many works that deal with detection of design patterns [115]. Among few initial works, an attempt to find structural design patterns (adapter, proxy, composite, bridge, and decorator) in object-oriented software was by Antoniol et al. [116]. Tsantalis et al. [87] presented a solution to design pattern detection problem that uses similarity score between design patterns and graph representation of the program to detect occurrences of the design patterns. They used a large set of well-known design patterns which includes adapter, command, composite, decorator, factory method, observer, prototype, singleton, state, strategy, template method, and visitor. Yu et al. [94] in 2013 presented an approach for detecting decorators design patterns using graph isomorphism. Recently in 2014, a semantic web based technique for detecting 11 types of design patterns is proposed by Alnusair et al. [117]. Compared to collaborative patterns presented in the thesis, these works assume pre-defined descriptions about the behavior and structure of the particular design pattern to be detected. Further, compared to collaborative patterns where program structures are clone of each other, design patterns need not necessarily be similar at the code level.

2.4. Classification of Collaborative Patterns

Based on the type of the program entity, we can further classify collaborative patterns. Such classification of collaborative patterns is useful in further

analysis of this phenomenon. As discussed earlier, possible types of program entities can be methods, classes, files, modules, components, directories, sub-systems, or any physical or logical groups of program entities. Based on these, we can classify collaborative patterns into different levels of abstraction. For example, suppose we have six classes A, A1, B, B1, C, and C1 with methods f(), f1(), g(), g1(), h(), and h1(), respectively as shown in Figure 2.4. We say that method configurations <f(), g(), h()> and <f1(), g1(), h1()> form a collaborative pattern at the method-level if the following conditions hold:

- methods f() and f1(), g() and g1(), and h() and h1() have been identified as method clones of each other.
- methods call each other as indicated by arrows in Figure 2.4 representing control flow in the program, i.e., A.f()→B.g()→C.h() and A1.f1()→B1.g1()→C1.h1().

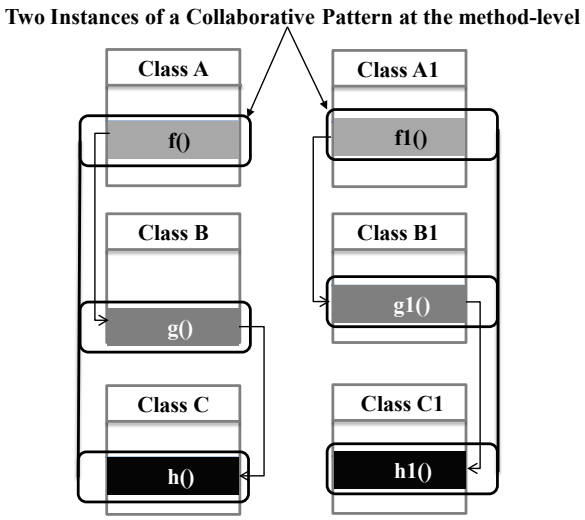


Figure 2.4. Collaborative pattern at the method-level

Further, assume that classes A and A1, B and B1, and C and C1 are class clones of each other. Arrows indicate calling relationship such that any of the methods from a class calls any of the other methods of another class (for

example, arrow from class A to class B indicates that any of the method, say f(), calls any other method, say g(), in class B). Then, the class configurations <A, B, C> and <A1, B1, C1> form a collaborative pattern at the class-level (Figure 2.5).

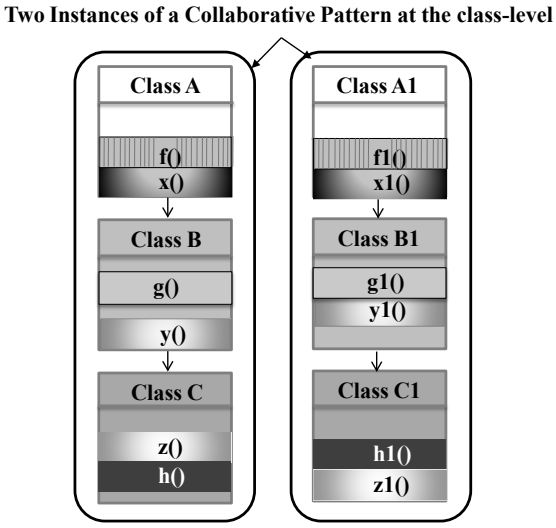


Figure 2.5. Collaborative pattern at the class-level

Similarity, we can define collaborative patterns at the levels of files, directories, or components, where component is any physical or logical grouping of program elements (Figure 2.6).

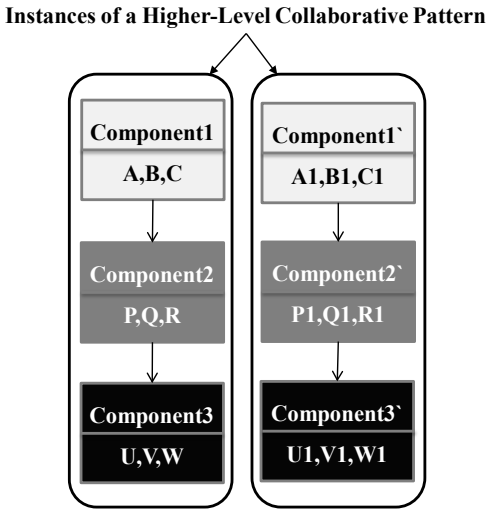


Figure 2.6. Higher-level collaborative pattern

2.5. Importance and Benefits of Collaborative Patterns

Detection of various types of code clones, in general, helps in program understanding, program refactoring, error detection, quality assessment, reuse, plagiarism detection, software evolution, maintenance, and others [3, 8]. Similar to structural clones, collaborative patterns being large-granular program structures, facilitate better context-analysis and represent important design information [5]. Thus, collaborative patterns can prove to be even more useful in all these scenarios. The detected clone units are large enough to exhibit conceptual similarities that help in better understanding of the cloning in the software system. One can expect better code compaction due to bigger units of detected clones. In addition, detection of collaborative patterns can help in creating generic representation of the entire system using technique such as the ART (explained in Chapter 5). This generic representation can extend the scope of reuse beyond the conventional architecture-centric, component based methods. Sections 4.4 and 6.1 explore the benefits of detecting and managing collaborative patterns in detail.

2.6. Methodology for Detecting and Managing Collaborative Patterns

In the previous sections, we discussed the concept of collaborative patterns in detail. This section briefly outlines the proposed approach for detecting and managing these collaborative patterns. The detailed approach is presented in the forthcoming chapters.

Figure 2.7 gives an overview of the collaborative pattern detection and management approach that includes four phases: pre-detection analysis,

collaborative pattern detection, post-detection analysis with user involvement, and collaborative pattern management. The proposed approach performs collaborative pattern detection by finding small-granular code clones first, and then gradually raising the level of detection to higher-level collaborative patterns. Each phase is explained briefly in the forthcoming subsections.

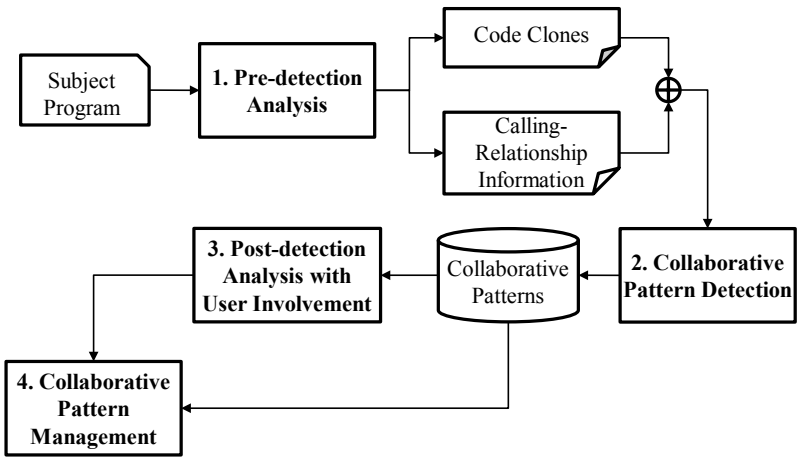


Figure 2.7. Methodology for detecting and managing collaborative patterns

2.6.1. Phase 1: Pre-detection Analysis

This phase deals with finding all the relevant information from the subject program needed for detecting collaborative patterns. Based on the definition of collaborative patterns, we identified two pieces of information that we must have for detecting collaborative patterns: small-granular code clones and calling-relationship information among program entities. Sections 3.2.1 and 3.2.2, in the next chapter explain pre-detection analysis phase in detail.

2.6.2. Phase 2: Collaborative Pattern Detection

We are using token-based string pattern matching algorithm for detecting collaborative patterns. We represent each method-calling sequence (extracted from the calling-relationship information) as a string of tokens. During this

step, a unique token is assigned to all the methods that have been detected as code clones in the previous phase. These token strings are then concatenated into a single token-sequence. Then, repeated substrings of tokens in the concatenated token-sequence are found. Multiple occurrences of a repeated substring in the concatenated token-sequence indicate the occurrences of different instances of the same collaborative pattern. Section 3.2.3 in the next chapter explains the detection phase in detail.

2.6.3. Phase 3: Post-detection Analysis with User Involvement

Automated detection may result in many collaborative patterns in large software systems. Even larger numbers of clones are reported when clone detectors are used to find clones in a family of software systems, a usual prelude to re-engineering such families into Software Product Lines (SPLs) [9] for systematic reuse. However, among these large numbers of clones, users should have to pay most attention to those recurring structures whose knowledge in the software is likely to benefit the user. Post-detection analysis phase deals with filtering the patterns in order to isolate the beneficial ones. It helps users zoom into the areas that are of their interest. During this phase, user's task is to analyze patterns manually and identify ones that can be removed, prevented, unified to generic templates, or re-engineered for reuse. We discuss this phase in detail in Section 4.3.

2.6.4. Phase 4: Management of Collaborative Patterns

After detecting and analyzing detected collaborative patterns, the next important issue is of managing them. This phase deals with using detected collaborative patterns for easier program maintenance and better reuse. We

proposed a methodology for managing code clones of large granularity (such as collaborative patterns, or other large-granular cloned program structures) by presenting a meta-programming technique and tool, the ART. The ART is an enhanced, lightweight and XML-free version of the XVCL (XML-based Variant Configuration Language) [118]. The proposed methodology of managing code clones using the ART is based on the concept of representing the clones in the form of generic, adaptable, and reusable templates; we called them ART templates. The software systems represented using ART templates are easier to maintain due to smaller size of the code and have reduced conceptual complexity as perceived by the developers. The ART and mechanism of managing clones using the ART are explained in Sections 5.2 and 5.3 respectively.

2.7. Conclusions

This chapter formalized the concept of collaborative patterns in detail. Collaborative patterns are higher-level clones of large granularity that can be identified by systematically combining small-granular cloned program entities at various levels. We also presented a brief overview of the collaborative pattern detection and management approach. A detailed description of the collaborative pattern detection approach is presented in the next chapter. Experimentation results pertaining to detection approach are presented in Chapter 4. The approach for managing code clones is described in Chapter 5. Chapter 6 presents evaluation results related to the management approach discussed in Chapter 5.

Chapter 3.

DETECTING COLLABORATIVE PATTERNS

In the previous chapter, we formalized the concept of collaborative patterns and outlined the methodology for detecting and managing them. In this chapter, we present the collaborative pattern detection approach in detail. This chapter is organized as follows: Section 3.1 gives the scope of the approach. The proposed collaborative pattern detection approach is elaborated in Section 3.2. Section 3.3 presents the implementation details, and finally Section 3.4 concludes the chapter.

3.1. Scope of the Approach

In Section 2.2 in Chapter 2, we presented a detailed well formalized concept of collaborative patterns in terms of a directed graph. As illustrated in Section 2.4, collaborative patterns may have many variations depending upon the types of program entities considered. We restricted our current approach for collaborative pattern detection to cloned methods as interrelated program entities. Further, as shown in Figure 2.2 in Chapter 2, a collaborative pattern may have linear method-calls or it may have branched method-calls. But, it is

possible that collaborative patterns containing branched method-calls can be represented in the form of collaborative patterns having linear method-calls. For example, the collaborative pattern shown in left side of Figure 3.1 can be represented as two linear collaborative patterns as shown in the right side of Figure 3.1. So, we restricted the scope of the proposed approach to the detection of linear configurations of methods.

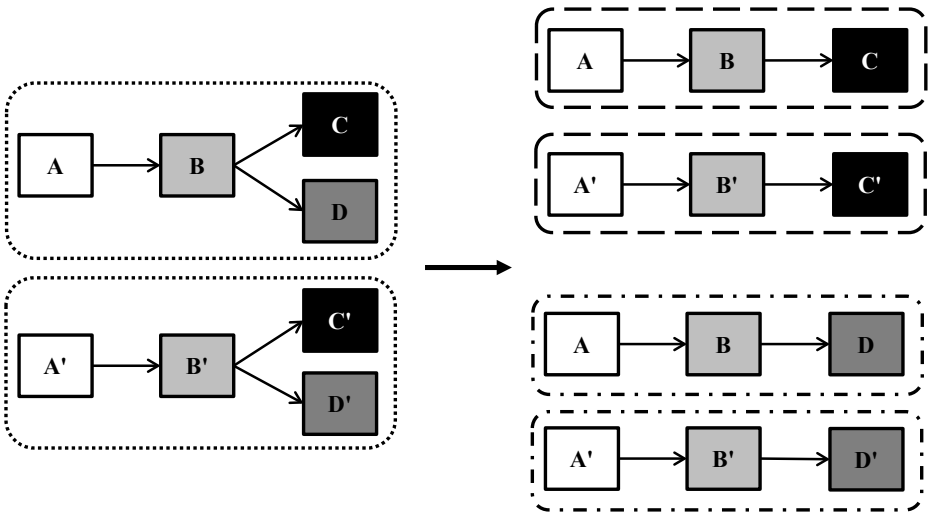


Figure 3.1. Branched collaborative pattern to linear collaborative patterns

As discussed in Chapter 2, collaborative patterns may have many variations depending upon the types of similarity metrics selected. In our current research, we focus on textual similarity among the program entities.

3.2. Detailed Approach

This section describes the collaborative pattern detection approach in detail. As highlighted in the previous chapter, two pieces of information needed for detecting collaborative patterns are code clones and calling-relationship information. So, we divided the detection process into three steps: code-clone

finder, calling-relation retriever, and collaborative-pattern detector (Figure 3.2).

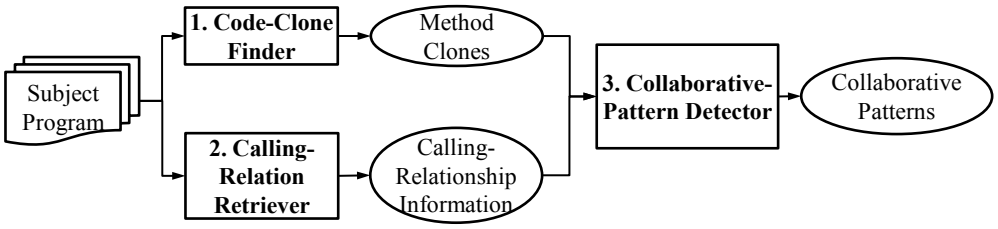


Figure 3.2. An overview of collaborative pattern detection approach

As shown in the figure, code-clone finder (component 1) finds method clones from the subject program. Calling-relation retriever (component 2) finds calling-relationship information from the subject program by analyzing its source code. The output of these two components is then used by the collaborative-pattern detector (component 3) to find collaborative patterns from the subject program. Next subsections discuss each of these components in detail.

3.2.1. Step 1: Code-Clone Finder

This component deals with finding method clones from the subject program. A group of methods are method clones of each other if and only if each member of the group has significant similarity with each of the other members of the group.

For finding method clones, we used an existing token-based clone detection algorithm proposed by Basit and Jarzabek [5]. The algorithm is implemented as a tool, Clone Miner. Clone Miner first detects cloned contiguous code fragments by using an efficient suffix-array based Non-Extendible Repeats Finding (NERF) algorithm [29], and then applies Frequent Closed Itemset

Mining technique for finding method clones from the detected cloned continuous code fragments. Some of the reasons for selecting the above technique and tool for finding method clones are:

- The proposed technique is very fast and scalable. This is because it uses suffix-array based code-tokenization technique followed by data mining approach for finding method clones. This makes it very fast and scalable.
- Clone Miner outputs method clones directly from the subject program very efficiently.

Although we used Clone Miner for finding method clones. Yet, this step is independent of it. Any technique or tool that can detect method clones can be used for this step.

3.2.2. Step 2: Calling-Relation Retriever

This component deals with finding calling-relationship information from the subject program by analyzing its source code. Literature suggests that finding all calling-relations from the source code using just static code analysis is hardly exhaustive [119, 120]. It is due to the reason that different programming features such as reflection or bytecode modification tooling, some information relevant to calling-relation retrieval is known only at the runtime. On the other hand, runtime analysis provides us calling-relations from specific program executions only. Hence, to overcome the disadvantages of static analysis over dynamic analysis and vice-versa, the proposed approach uses both static code analysis and dynamic code analysis for finding calling-relationship information. Based on these requirements, the calling-relation

retriever component is divided into different sub-components as shown in Figure 3.3.

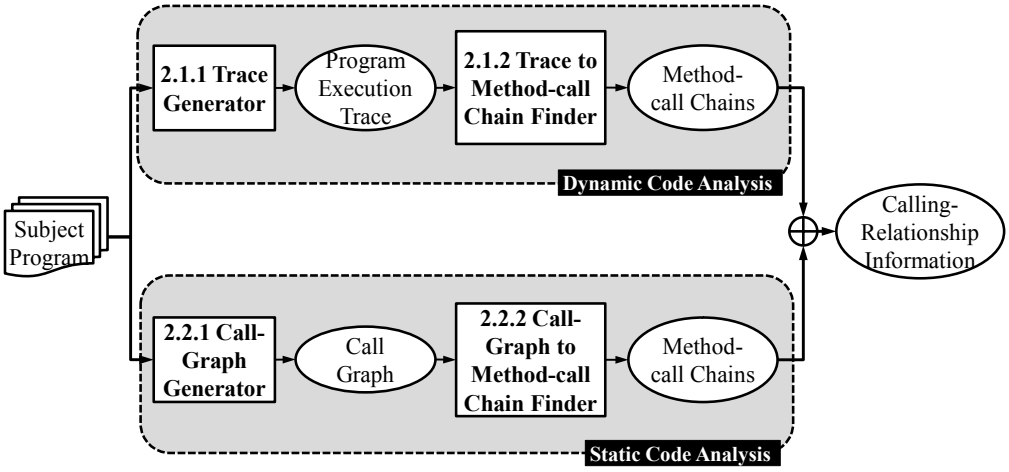


Figure 3.3. Detailed overview of calling-relation retriever component

During dynamic code analysis, the subject program is instrumented with trace-generator code to get program execution trace (Section 3.2.2.1), which in-turn is used to find calling-relationship information in the form of method-call chains (Section 3.2.2.2). Static code analysis allows getting calling-relationship information from the subject program by first generating call graph (Section 3.2.2.3), and then using the generated call graph to find method-call chains (Section 3.2.2.4).

3.2.2.1. Trace Generator

This component generates program execution trace from the subject program by analyzing its source code at the runtime.

Besides other information, a program execution trace contains ordered list of methods which are called, then executed, and finally returned during a particular run of the program. Figure 3.4 shows possible example of a program execution trace for a given program *P*.

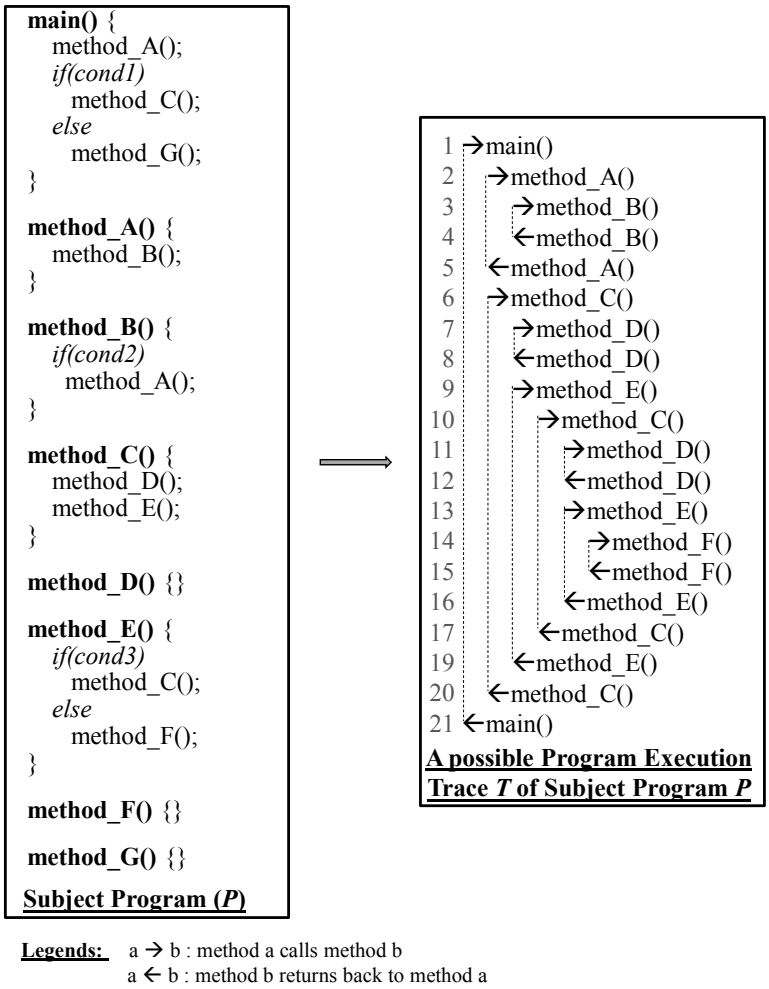


Figure 3.4. Example of a program execution trace

As shown in the figure, program *P* consists of a main() method and seven auxiliary methods: method_A(), method_B(), method_C(), method_D(), method_E, method_F(), and method_G(). The main() method calls some of the auxiliary methods, i.e., method_A(), and method_C() or method_G(). The auxiliary methods call other methods as shown in the left side of the figure. A possible program execution trace *T* during a particular run of the program *P* would be like as shown in the right side of the figure. In the given program execution trace, main() calls method_A() first (line 2). method_A() during its current execution calls method_B() (line 3). Assuming cond2 evaluates to false, on finishing its current execution, method_B() returns program-

execution control back to method_A() (line 4). Similarly, method_A() returns program-execution control back to main() method (line 5). Given cond1 evaluates to true, main() method calls method_C() (line 6) which in-turn first calls method_D() (lines 7 and 8), and then calls method_E() (line 9). Assuming cond3 to be true, method_E during its current execution calls method_C() (line 10). During this execution, method_C() again first calls method_D() (lines 11 and 12), and then calls method_E() (line 13). Assuming this time, cond3 evaluates to false, current instance of method_E() calls method_F() (lines 14 and 15). On finishing its current execution, current instance of method_E() returns program-execution control back to method_C() (line 16). The rest continues until the execution control reaches the end of the program (lines 17–21).

Trace generator uses the concept of aspect-oriented programming (AOP) [121] to get the program execution trace. Three features of AOP that are used for the proposed approach are: Joinpoints, Pointcuts, and Advices. Joinpoints are well-defined points (e.g., method call, method execution, exception handlers, or class/object initializations) in the flow of program-execution control. Certain set of Joinpoints makes a Pointcut. Advices define the code that is applied when a particular Pointcut is reached. For our approach, we are interested in keeping the track of methods' executions only. So, the Joinpoints of our interest are limited to those that point to the executions of methods in the subject programs. Hence, we define Pointcuts to keep track of the methods executed during runtime. We define Advices that allow storing the information about entering and exiting of these executed methods. Thus, the generated program execution trace contains an ordered list of methods that are called and

returned during the run of the program (as shown in the right side of Figure 3.4).

3.2.2.2. Trace to Method-call Chains Finder

For detecting collaborative patterns, we need only method calling-relationship information from the generated program execution trace. Other information such as returning-method information is of no use during detection process. Hence, this component processes the trace to get method calling-relationship information from it. For this, we split the program execution trace in the form of method-call chains. A method-call chain in a program execution trace is defined as:

Definition 3.1 (method-call chain in a program execution trace): *In a given program execution trace, a method-call chain is a sequence of methods* $f_1(),f_2(),\ldots,f_i,\ldots,f_{n-1}(),f_n()$*; n ≥ 2 executed in such a way that for all* $1 \leq i \leq n-1$*:*

- *Method* $f_1()$ *is the first executing method in the program execution trace.*
- *Method* $f_i()$ *during its current execution calls method* $f_{i+1}()$*, and*
- *Method* $f_n()$ *during its current execution does not calls any other method and returns the program-execution control back to the method* $f_{n-1}()$*.*

The first condition allows starting each method-call chain with the first executing method of the programs (main() method, for example). It is helpful in finding longest method invocation sequences from the subject program and thus reduces the number of total unique method-call chains. Figure 3.5 gives an example of method-call chains constructed from the program execution

trace *T* of the subject program *P*. As shown in the figure, we have four method-call chains corresponding to the given program execution trace. Each method-call chain starts with the main() method of the program *P*, and extends until there is not a method which does not call any other method during its current execution.

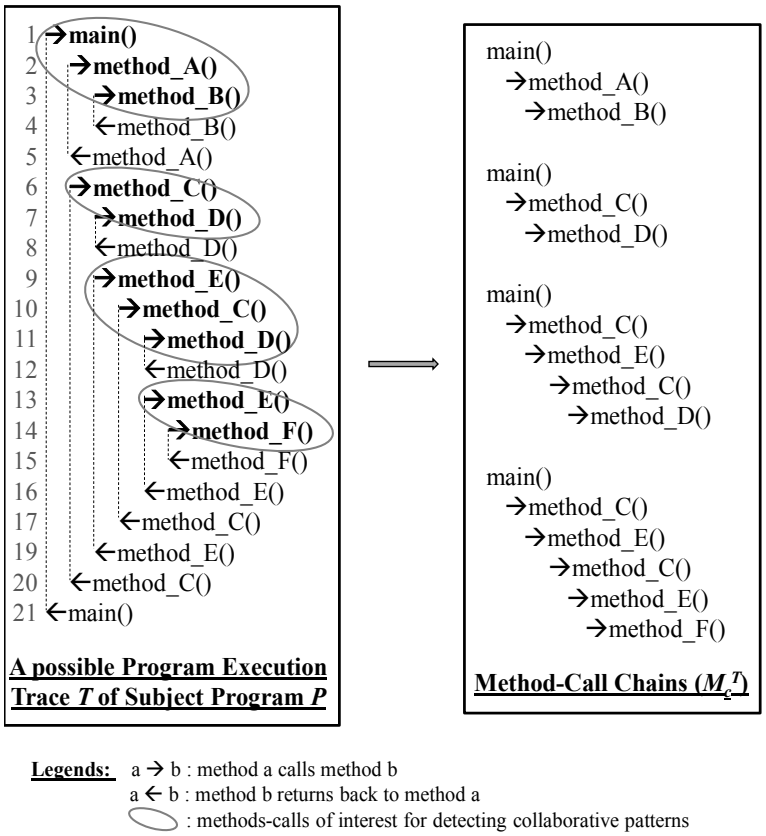


Figure 3.5. Splitting of a program execution trace into method-call chains

To find the method-call chains, the program execution trace is parsed sequentially from the start, i.e., the first method (the main method) of the program. This first method is added to an empty list. Since the program execution trace keeps track of only entering and exiting of the methods, we have two cases: either there is a method-call or a method is returning back to the caller during its execution. In the first case, i.e., when there is a method-call during the execution, the called method is appended to the list. In the

second case, i.e., when the method is returning back to the caller after finishing its execution, we check whether during its current execution, this method called any other method before returning. If yes, just remove the last method from the list. If no, output the current list as an instance of the method-call chain and remove the last method from the list.

Figure 3.6 shows the algorithm for finding method-call chains from the program execution trace. Following steps describe the algorithm in detail:

1. Create an empty list *Record* that keeps track of the methods whose instances are currently active during the program execution.
2. Create an empty list of lists, *ChainList* that stores method-call chains constructed from the program execution trace.
3. Create an integer counter *Counter* initially set to 0.
4. Parse the program execution trace from the start (i.e., from the first executed method of the program) till the end. Perform Steps 5 and 6 during the parsing.
5. On entry to a method, append the method to the *Record* and update the *Counter* to the size of the *Record*.
6. On exit from a method, check whether the size of the *Record* is equal to the *Counter*.
    6.1. If yes, add the current content of *Record* to the *ChainList*. After that remove the last element from the *Record*.
    6.2. Otherwise, remove the last element from the *Record*.
7. Repeat Steps 5 and 6 until the end of the program execution trace.
8. Remove duplicate records from the *ChainList*.

Algorithm 1: Trace to Method-call Chains Finder

Input: Program Execution Trace (*T*). *T* is a 2-tuple list with first element being the method-signature and second element is a flag (set to either *entry* or *exit)* specifying whether method is entering or exiting during the program execution.

```
TRACE-TO-MCCHAINS(T)
1.  ▷ Initialize the variables
2.  Record ← ϕ              ▷ A list
3.  ChainList ← ϕ           ▷ A list of list
4.  Counter ← 0
5.  ▷ Parse the elements of the list T from the start.
6.  for each pair (method_signature, flag) ∈ T do
7.      if flag == entry then
8.          ▷ Add the method_signature to the end of the Record
9.          APPEND(Record, method_signature)
10.         Counter = SIZE(Record)
11.     endif
12.     if flag == exit then
13.         if SIZE(Record) == Counter then
14.             ▷ Add the current content of Record to the ChainList
15.             APPEND(ChainList, Record)
16.             ▷ Remove last element form the Record
17.             REMOVE-LAST-ELEMENT(Record)
18.         endif
19.         else
20.             ▷ Remove last element form the Record
21.             REMOVE-LAST-ELEMENT(Record)
22.         endelse
23.     endif
24. endfor
25. REMOVE-DUPLICATES(ChainList)
26. return ChainList
27.end
```

Output: A list of method-call chains (M_c^T)

Figure 3.6. Trace to method-call chains finder algorithm

3.2.2.3. Call-Graph Generator

Call-graph generator takes the subject program as input and generates a call graph from it by statically analyzing its source code.

A call graph is a directed cyclic graph that represents calling-relations among methods. Vertices in the graph correspond to methods and edges between the vertices indicate calling-relations among methods. For example, an edge from

method *f* to method *g* indicates that some call-site in method *f* calls method *g*. Similarly, a cycle in the graph indicates recursive method-calls. Considering the same sample program *P* as in Figure 3.4, Figure 3.7 shows the call graph corresponding to program *P* that can be constructed by using existing techniques and tools such as CGC [120] or WALA [122].

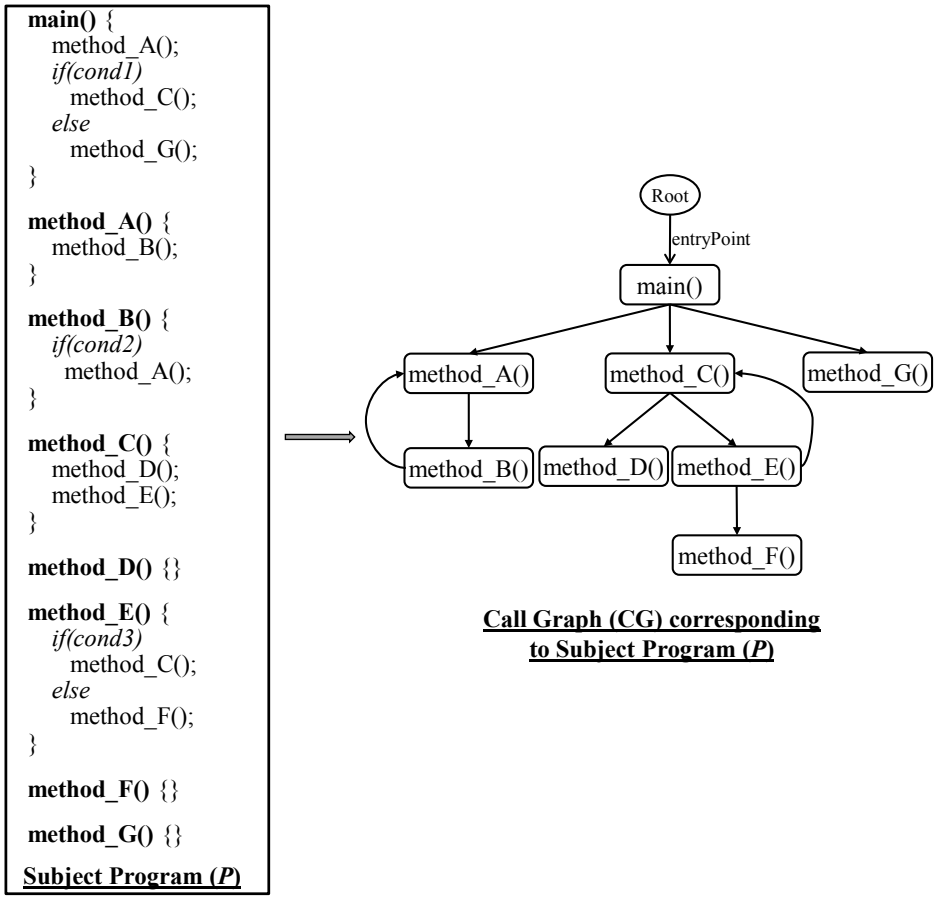


Figure 3.7. Example of a call graph

3.2.2.4. Call-Graph to Method-call Chains Finder

This component splits the generated call graph into method-call chains, which have required calling-relationship information for detecting collaborative patterns. A method-call chain in a call graph is defined as:

Definition 3.2 (Method-call chain in a call graph): *In a call graph CG(V, E), a method-call chain is defined as a sequence of vertices $V_1, V_2, \ldots, V_n$; $n \geq 2$ such that:*

- *The sequence begins with a vertex V_1 representing the entry-point of the program (e.g., its main() method),*
- *For all $1 \leq i \leq n-1$, there is a directed edge $e \in E$ from vertex V_i to vertex V_{i+1}, and*
- *The sequence ends with a vertex V_n that is either a sink vertex (i.e., a vertex with out-degree zero) or is the first repeating vertex along the sequence.*

Figure 3.8 shows an example of method-call chains constructed from the call graph of program *P*.

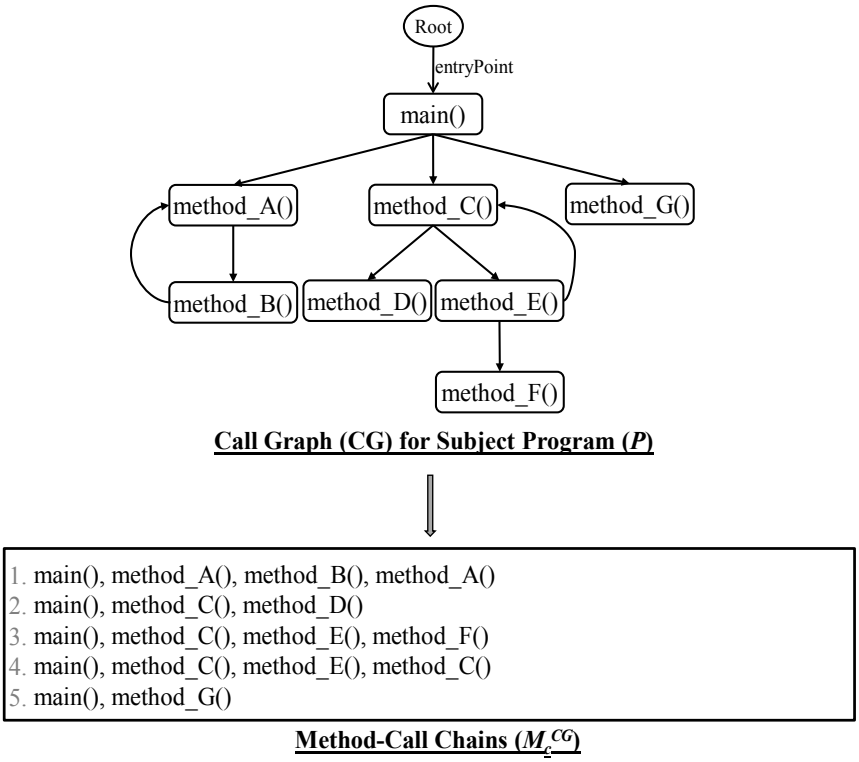


Figure 3.8. Splitting of a call graph into method-call chains

As shown in Figure 3.8, we have five method-call chains corresponding to the given call graph. Each method-call chain starts with the entry-point of the program (i.e., main() method). Out of the five method-call chains, three chains (numbered 2, 3, and 5) end with a sink vertex (method_D(), method_F(), and method_G(), respectively) of the call graph. While the remaining two chains (numbered 1 and 4) end at the first repeating vertex of respective sequences (method_A() and method_C(), respectively).

The proposed algorithm for finding method-call chains traverses the call graph starting from the vertex indicating the entry-point of the program. For the most recently traversed vertex (say u), all the edges that have u as the tail are explored and appended to a list. During this traversal, following set of data-structures is maintained to keep track of the information needed for finding method-call chains:

- A pair-list *NodeSuccessorsPairList*, with element of the format (nodeX, nodeXSuccessor) to keep track of the edges explored from the call graph during traversal. For a given explored vertex u, if there is a directed edge from vertex u to v, a pair (u, v) is appended to the list.
- A list *Chain*, with element of the format nodeX keeps sequence of vertices explored during traversal that can be a part of possible method-call chain.
- A list of lists *ChainList*, with element of the format *Chain*, to store all the method-call chains constructed during call-graph traversal.
- An integer array *counter*[] of size $|V[CG]|$. Each element of the array *counter*[V_i] keeps the integer value indicating the number of times the corresponding vertex V_i is included in the *Chain* list.

Figure 3.9 shows the algorithm for finding method-call chains from a call graph. Following steps describe the algorithm in detail:

1. Append vertex *r* indicating the entry-point of the program to the *Chain* and increment the corresponding *counter*.
2. For each adjacent directed edge from *r* to *u*, append the pair (*r*, *u*) to the *NodeSuccessorsPairList*.
3. Repeat Steps 4 to 7 until the *NodeSuccessorsPairList* is not empty.
4. Remove the last element from the *NodeSuccessorsPairList* and store it in a pair (π, *u*).
5. Remove the last element from the *Chain* until the last element is not equal to π (i.e., predecessor of *u*) and decrement the corresponding *counter*.
6. If the *counter* corresponding to vertex *u* is 0 (it means the vertex is not yet added to the *Chain*): append the vertex *u* to the *Chain* and increment the corresponding *counter*. Then, check whether *u* is a sink vertex.
    6.1. If no (i.e., vertex *u* is not sink vertex): for each adjacent edge from *u* to *v*, append (*u*, *v*) pair to the *NodeSuccessorsPairList*. Go to Step 3.
    6.2. If yes (i.e., vertex *u* is sink vertex): add the current content of *Chain* to the *ChainList* as an element. Then, remove the last element from the *Chain* and decrement the corresponding *counter*. Go to Step 3.
7. If the *counter* corresponding to vertex *u* is not 0 (it means the vertex is already added to the *Chain*): first, append the vertex *u* to the *Chain*.

Then, add the current content of *Chain* to the *ChainList* as an element.

After that, remove the last element from the *Chain*. Go to Step 3.

Algorithm 2: Call Graph to Method-call Chains Finder

Input: Call graph *CG(V, E) with vertex r as the entry-point of the program*

```
CALLGRAPH-TO-MCCHAINS(CG, r)
1.  Chain ← ϕ                          ▷ A list
2.  ChainList ← ϕ                      ▷ A list of lists
3.  NodeSuccessorsPairList ← ϕ ▷ A pair-list of format (nodeX, nodeXSuccessor)
4.  for each vertex u ∈V[CG] do
5.  |   counter[u] ← 0
6.  APPEND(Chain, r)
7.  counter[r]++
8.  for each u ∈adj[r] do
9.  |   APPEND(NodeSuccessorsPairList, (r, u))
10. while NodeSuccessorsPairList ≠ ϕ do
11. |   (π, u) ← GET-AND-REMOVE-LAST-PAIR(NodeSuccessorsPairList)
12. |   while (π ≠ GET-LAST-ELEMENT(Chain)) do
13. |   |   v ← GET-AND-REMOVE-LAST-ELEMENT(Chain)
14. |   |   counter[v]--
15. |   endwhile
16. |   if (counter[u] == 0) then
17. |   |   APPEND(Chain, u)
18. |   |   counter[u]++
19. |   |   if adj[u] ≠ ϕ then
20. |   |   |   for each v ∈adj[u] do
21. |   |   |   |   APPEND(NodeSuccessorsPairList, (u, v))
22. |   |   endif
23. |   |   else
24. |   |   |   APPEND(ChainList, Chain)
25. |   |   |   REMOVE-LAST-ELEMENT(Chain)
26. |   |   |   counter[u]--
27. |   |   endelse
28. |   endif
29. |   else
30. |   |   APPEND(Chain, u)
31. |   |   APPEND(ChainList, Chain)
32. |   |   REMOVE-LAST-ELEMENT(Chain)
33. |   endelse
34. endwhile
35. return ChainList
36. end
```

Output: A list of method-call chains (M_c^{CG})

Figure 3.9. Call graph to method-call chains finder algorithm

The current approach for finding method-call chains from the call graph restricts the graph traversal to stop as soon as a vertex is visited for the second time. It is done intentionally to avoid explosion during the generation of method-call chains. This is true that it avoids some of interested method-call chains from the analysis. But, such recursive calls/cycles can be easily handled during tracing. With reference to Figure 3.8, consider a case where method_C() calls method_E(), which then calls method_C() (i.e., loops back in call graph), then method_C() calls method_D(); resulting in a method-call chain with substring "…, method_C(), method_E(), method_C(), method_D(), …". Such cases are possible to be extracted during program tracing, resulting in a method-call chain: "main(), method_C(), method_E(), method_C(), method_D()".

3.2.3. Step 3: Collaborative Pattern Detector

This component detects collaborative patterns from the subject program using the method clones and the method-call chains found so far.

The proposed algorithm for detecting collaborative patterns is based on the underlying concept that a set of method-call chains form a collaborative pattern if the corresponding methods in the method-call chains are either same or belong to a clone class. We are using token-based string pattern matching for detecting such sets of method-call chains. In the first step, we represent each of the method-call chains as a string of tokens. We use same token-ID for all the methods that form a clone class. These token strings are then concatenated into a single token-sequence. This arrangement allows us to use a straight forward variation of existing Non-Extendible Repeat Finder (NERF)

algorithm [29] to find repeated substrings of tokens in the concatenated token-sequence. NERF computes all non-extendible repeats in the concatenated token-sequence. These non-extendible repeats are then used to find out collaborative patterns. Multiple occurrences of a given repeat in the concatenated token-sequence point to different instances of the same collaborative pattern. Figure 3.10 shows algorithm for detecting collaborative patterns which is demonstrated using an example in Figure 3.11.

Algorithm 3: Collaborative Pattern Detection Algorithm

Input: Method-Clone Classes, *MCLONE*
Method-call Chains, $M_c = M_c^T \cup M_c^{CG}$

DETECT-CP(*MCLONE*, M_c)

1. ▷ Initialize data structures needed for detecting collaborative patterns
2. *methodID* ← ϕ
3. *tokenString* ← ϕ
4. *concatenatedString* ← ϕ
5. *suffixArray* ← ϕ
6. *lcpArray* ← ϕ
7. *repeats* ← ϕ
8. *ccclasses* ← ϕ
9. ▷ Assign Unique Method-IDs to all methods
10. *methodID* ← ASSIGN-METHOD-ID(M_c, *MCLONE*)
11. ▷ Tokenize and Assign unique Chain-ID to each method-call chain
12. *tokenString* ← TOKENIZE-AND-ASSIGN-CHAIN-ID(M_c, *methodID*)
13. ▷ Concatenate *tokenString* to form a single token-sequence
14. *concatenatedString* ← CONCATENATE(*tokenString*)
15. ▷ Create Suffix Array from the *concatenatedString* using KS Algorithm
16. *suffixArray* ← CREATE-SUFFIX-ARRAY(*concatenatedString*)
17. ▷ Create LCP Array from the *concatenatedString* and *suffixArray* using GetHeight algorithm
18. *lcpArray* ← CREATE-LCP-ARRAY(*concatenatedString*, *suffixArray*)
19. ▷ Compute Repeats in the *concatenatedString* using NERF algorithm
20. *repeats* ← COMPUTE-REPEATS(*concatenatedString*, *suffixArray*, *lcpArray*)
21. ▷ Reverse map the token Strings and method IDs on repeats using method-call chains to compute collaborative clone classes.
22. *ccclasses* ← GET-COLLABORATIVE-CLONE-CLASSES(*repeats*, *tokenString*, *methodID*, M_c)
23. **end**

Output: A set of collaborative clone classes (*ccclasses*).

Figure 3.10. Collaborative pattern detection algorithm

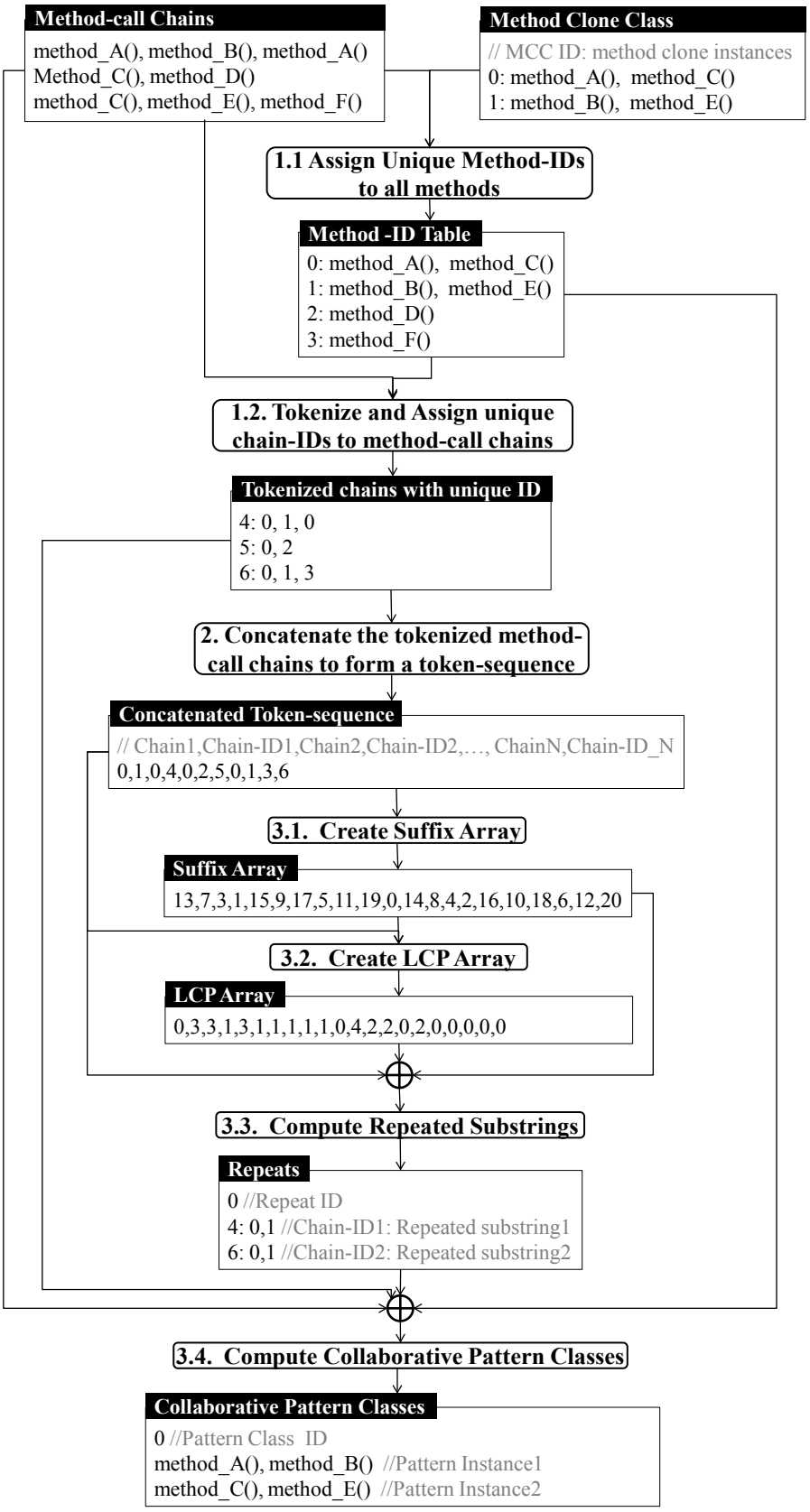


Figure 3.11. Collaborative pattern detection algorithm: illustrative example

Following steps describe the detection algorithm in detail:

1. *Tokenize and assign a unique chain-ID to each of the method-call chain*: This step consists of two tasks:
    1.1. Assign a unique ID to each method in such as way that all the members of a clone class have the same method-ID. Store method-ID and method-name information in a symbol table *methodID*.
    1.2. Represent each of the method-call chains as a string of tokens and assign a unique chain-ID to each of the tokenized method-call chain. To easily distinguish chain-IDs from method-IDs, keep the respective set of IDs to be disjoint of each other. Store the tokenized method-call chains in *tokenSting*.
2. *Concatenate the tokenized method-call chains to form a token-sequence:* Concatenate all the tokenized method-call chains into a single token-sequence (*concatenatedString*). Use a unique sentinel token, which is same as the unique chain-ID assigned to each of the tokenized method-call chain, to distinguish two tokenized method-call chains in the concatenated token-sequence.
3. *Compute repeated substrings of tokens in the concatenated token-sequence*: This step consists of four tasks:
    3.1. Initially, generate a suffix array (*suffixArray*) from the concatenated token-sequence using KS Algorithm [123].
    3.2. Compute longest common prefix (LCP) information (*lcpArray*) from the concatenated token-sequence and suffix array using a

linear-time algorithm, GetHeight (proposed by Kasai et al. [124]).

3.3. Use the concatenated token-sequence, suffix array, and LCP information to compute repeated substrings of tokens (*repeats*) in the concatenated token-sequence using existing Non-Extendible Repeat Finder (NERF) [29] algorithm. NERF gives different sets of repeated substrings as output.

3.4. Each set of repeated substrings corresponds to a collaborative clone class. Multiple occurrences of a repeated substring in the concatenated token-sequence indicate various instances of the same collaborative pattern. Reverse map the token Strings (*tokenString*) and method-IDs information (*methodID*) on the repeated substrings using Method-call chains (M_c) to get corresponding method names. Store this information in *ccclasses*.

3.3. Tool Implementation

The proposed collaborative pattern detection approach is implemented as a prototype tool called CoPaD (Collaborative Patterns Detector). CoPaD is implemented in Java with the extensive use of Apache Commons APIs [125]. We implemented AOP functionalities using AspectJ [126]. The whole system has three components:

1. First component, CMTOOL, detects method clones. We ran Clone Miner as a black box to get the method clones.

2. Second Component, TRACER, finds method-call chains from the program execution trace of the subject program generated at runtime. We implemented this component in Java using AspectJ.
3. Third component, CPDTOOL, is implemented in Java. It uses the outputs of above components to detect collaborative patterns.

In current form, CoPaD does not come with a graphical user interface but lists all the detected collaborative patterns in text file for easy navigation. In our list of future works, in line with Clone Analyzer [127], we plan to develop a rudimentary graphical user interface for visualization and analysis of collaborative patterns.

3.4. Conclusions

In this chapter, we presented the detailed approach for detecting collaborative patterns. The proposed approach first finds method clones and calling-relationship information from the subject program, and then uses this information for detecting collaborative patterns. We implemented the proposed approach as a prototype tool, called CoPaD. In the next chapter, we present experimentation related to the proposed approach.

Chapter 4.

EXPERIMENTATION

In the previous chapter, we presented the detailed approach for detecting collaborative patterns. In this chapter, we present experimentation results pertaining to the proposed approach. The chapter is organized as follows: We present brief overview of experimentation goals in Section 4.1. Detection results are highlighted in Section 4.2. Section 4.3 presents experimentation pertaining to the analysis of detected collaborative patterns. The benefits and applications of detected collaborative patterns are outlined in Section 4.4. Finally, the chapter is concluded in Section 4.5.

4.1. Goals of Experimentation

We performed experimentation keeping the following goals:

- *Detection—to detect collaborative patterns in software systems:* We performed experimentation to quantitatively assess the presence of collaborative patterns in software systems.
- *Analysis—to analyze the detected collaborative patterns:* This part deals with qualitatively and quantitatively analyzing the detected

collaborative patterns to evaluate their usefulness and benefits. Besides high-level clones of large granularity, collaborative patterns help in tracing important calling-relationship information from the source code of the subject program. Hence, we further analyze the detected collaborative patterns to find the method-calls that lead to or emerge from these collaborative patterns (further details to follow in Section 4.3). Such information proves to be very useful in finding similar process flows in the software.

The rest of the chapter discusses the experimentation in detail.

4.2. Detection Overview

We performed collaborative pattern detection and analysis on the source code of the JHotDraw 7 v.7.6.0 [128] and Clone Analyzer v.2.0 [127] using the proposed approach and the tool implemented using it, i.e., CoPAD. JHotDraw 7 is a two-dimensional graphical user interface framework for structured drawing editors written in Java. Clone Analyzer is a clone visualization and analysis tool. It allows the user to filter clones that are of interest to him/her. Table 1 shows features of these programs.

Table 1. Features of subject programs considered for evaluating collaborative pattern detection approach

Subject Program	JHotDraw 7	Clone Analyzer
Version Number	7.6.0	2.0
Language	Java	Java
Input size in terms of token count	3,49,399 Tokens	86,064 Tokens
Numbers of Source Files	514	40
Lines of Code	99,990	15,142
Average Statements per Methods	5.35	8.276

4.2.1. Detection Results

In JHotDraw 7, by using minimum clone size of 30 tokens, we found total of 1,001 small-granular code-clone classes (i.e., groups of fragments of duplicated contiguous code). By using method percentage coverage (MPC—percentage of a method covered by code clones) = 30% and method token coverage (MTC—number of tokens in a method covered by code clones) = 30, total of 413 method-clone classes are detected in the JHotDraw 7. We ran JHotDraw 7 with sample inputs such as Draw, PERT, and Teddy provided with the JHotDraw 7 package. We found total of 2,924 unique method-call chains in this subject program. It is to mention that the numbers of unique method-call chains reported is 2,924, which may seem to be low. A method-call chain consists of a list of methods that are called one after another. We found the length of method-call chains reported in an execution to be upto 24. However, it is true that same method is included in more than one method-call chain. Yet, 2,924 method call-chains with length upto 24 cover significant part of the source code during execution. In overall, we detected 248 collaborative patterns in the JHotDraw 7 source code. The number of instances in the collaborative patterns range from two to six. Table 2 shows summary of the collaborative pattern detection results.

Table 2. Summary of collaborative pattern detection results

Subject Program	JHotDraw 7	Clone Analyzer
Total Small-granular Code-Clone Classes (with minimum clone size = 30)	1,001	407
Total Method-Clone Classes (with MTC = 30, MPC = 30%)	413	63
Total Methods in Method-Clone Classes	1,078	337
Total Unique Method-call Chains	2,924	185
Total Collaborative Patterns	248	27
Minimum and Maximum number of Instances (*I*) for a Collaborative Pattern	2 to 6	2 to 17

By using the same pattern detection settings, we ran Clone Analyzer with its own source code as input. We found total of 185 unique method-call chains in the Clone Analyzer v2.0 source code. In total, 27 collaborative patterns are found in the source code. The number of instances in the collaborative patterns range from 2 to 17.

4.3. Analysis Overview

We observed different variations in the structure of the collaborative patterns, the detection of which may be useful for the analyst, designer, or implementer. Since collaborative patterns help in tracing important calling-relationship information from the source code, we analyzed the detected collaborative patterns manually based on the method-calls that lead to or emerge from these collaborative patterns. This manual analysis is based on the following three factors:

1. *The methods participating in the instances of the collaborative pattern*. The first source of information we analyzed is the methods participating in the pattern-instances. For example, methods participating in the instances of a collaborative pattern shown in the Figure 4.1 are A_1, A_2, A_n, A_1', A_2', A_n', A_1'', A_2'', and A_n''. Boxes with shade show such methods. The number of methods participating in an instance of the collaborative pattern can be termed as the length of the collaborative pattern. For the example shown in Figure 4.1, the length of the collaborative pattern is three.
2. *The method-calls which lead to the collaborative pattern.* We analyzed the method-calls and the corresponding method(s) that call(s) the first-

methods of the respective instances of the collaborative pattern. In Figure 4.1, A_1, A_1', and A_1" are first-methods in the respective instances of the given collaborative pattern. P and P' are the methods which call these methods.

3. *The method-calls which emerge from the collaborative pattern.* We also analyzed the method-calls and the corresponding method(s) which is/are called by the end-methods of the respective instances of the collaborative pattern. As shown in Figure 4.1, A_n, A_n', and A_n" are end-methods in the respective instances of the given collaborative pattern. Q is the common method which is called by all these end-methods.

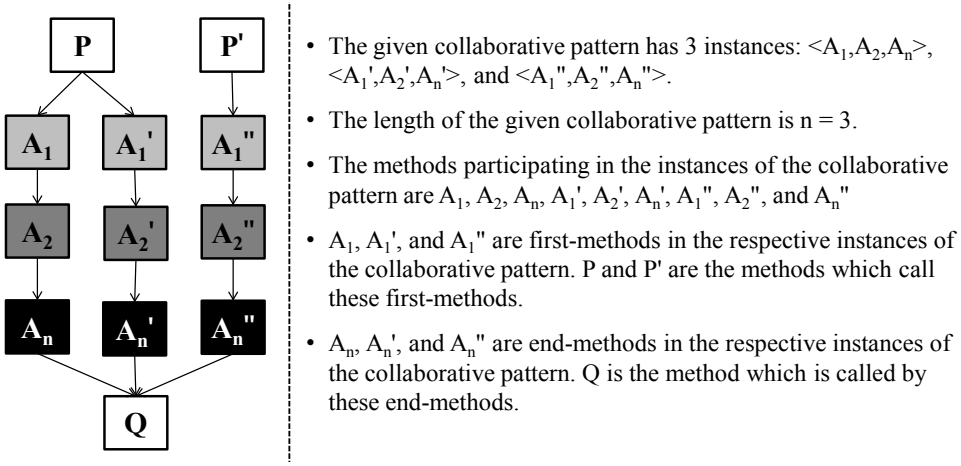


Figure 4.1. An example of a collaborative pattern with three instances

Figure 4.2 shows different cases of collaborative patterns emerged from the analysis of case studies discussed in the previous section. Assume that for $1 \leq i \leq n$, <A_i, A_i', A_i",…> represents a method-clone class as shown by the same shade in the figure. P, Q, R, and S represent methods which call the methods or are called by the methods forming the collaborative pattern.

Figure 4.2(a)–(d) show the cases when each of the corresponding methods (e.g., A_1, A_1', A_1"; similarly others) in the instances of the pattern is unique

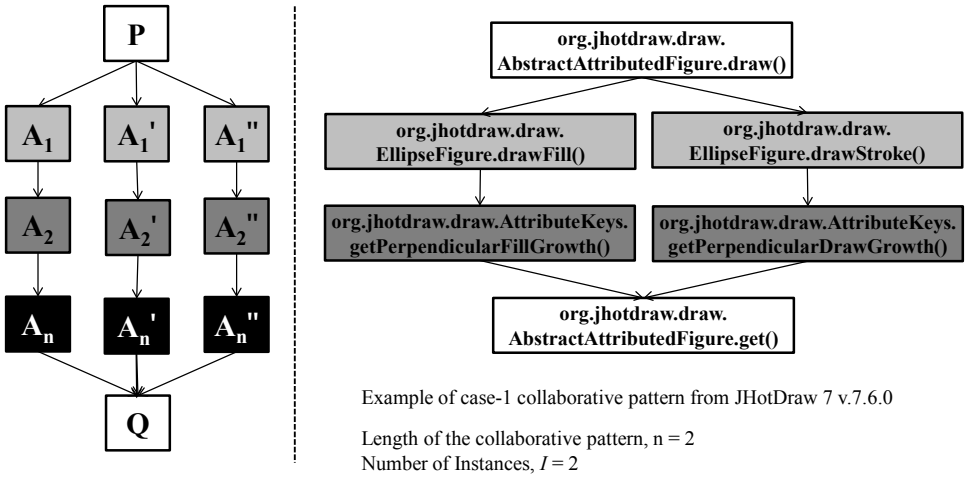
(i.e., $A_1 \neq A_1' \neq A_1''$). Such cases are divided into four classes based on whether there is a common method which calls or is called by the first-methods or the end-methods of the collaborative pattern, respectively.

As shown in Figure 4.2(a), there is a common method (P) which calls each of the first-methods (i.e., A_1, A_1', and A_1'') of the collaborative pattern. Similarly, there is a unique method (Q) which is called by all the end-methods (i.e., A_n, A_n', and A_n'') of the collaborative pattern. The right side of Figure 4.2(a) shows an example of such collaborative pattern from the JHotDraw 7 project. The given collaborative pattern has two instances. The length of the collaborative pattern is also two. There is a method 'draw()' in the 'AbstractAttributedFigure' class of 'org.jhotdraw.draw' package which calls two cloned methods—'drwaFill()' and 'drawStroke()'—both of which are the first-methods of the given collaborative pattern. The end-methods of the collaborative pattern, 'getPerpendicularFillGrowth()' and 'getPerpendicularDrawGrowth()' further call a common method 'get()' from the 'AbstractAttributedFigure' class.

Figure 4.2(b)–(d) show other three cases based on factors 2 and 3. For example, Figure 4.2(d) shows a collaborative pattern of length three detected from the Clone Analyzer v2.0. The detected collaborative pattern has three instances. For this collaborative pattern, there is neither a common method which calls each of the first-methods (i.e., SecondaryNavigator.getJInternalFrame(), PrimaryNavigator.getJInternalFrame(), and UserMinerSettings.getJInternalFrame()) nor a common method which is called by all the end-methods (i.e., SecondaryNavigator.getJScrollPane(),

PrimaryNavigator.getJScrollPane(), and UserMinerSettings.getScrollPane()) of the collaborative pattern.

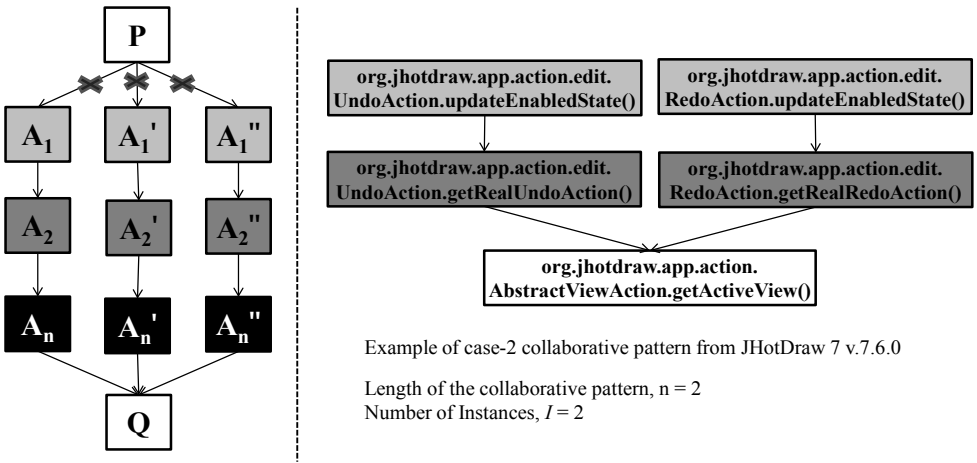
Similarly, when each of the corresponding methods in the instances of the pattern is not unique (but the corresponding methods are from the same method-clone class), we have four cases as shown in Figure 4.2(e)–(h). Figure 4.2(i) shows a special case of collaborative pattern in which the methods forming the instances of a collaborative pattern (<A_1, A_2,…, A_n>, for example) need not to be called successively one after the other.



Example of case-1 collaborative pattern from JHotDraw 7 v.7.6.0

Length of the collaborative pattern, n = 2
Number of Instances, *I* = 2

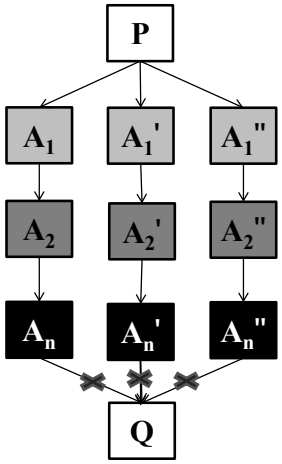
(a) Example of Case-1 Collaborative Pattern

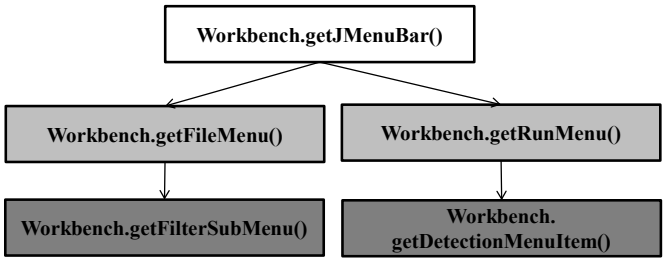


Example of case-2 collaborative pattern from JHotDraw 7 v.7.6.0

Length of the collaborative pattern, n = 2
Number of Instances, *I* = 2

(b) Example of Case-2 Collaborative Pattern

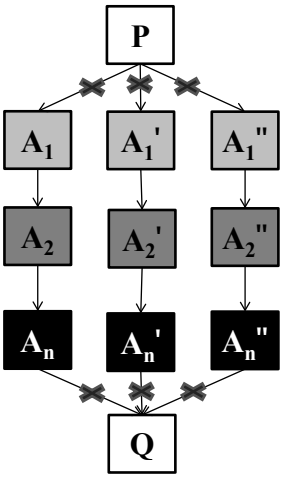


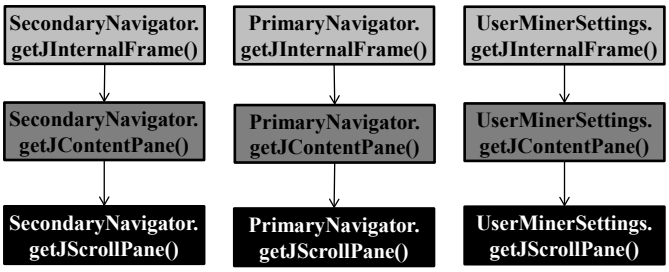


Example of case-3 collaborative pattern from Clone Analyzer v2.0

Length of the collaborative pattern, n = 2
Number of Instances, *I* = 2

(c) Example of Case-3 Collaborative Pattern

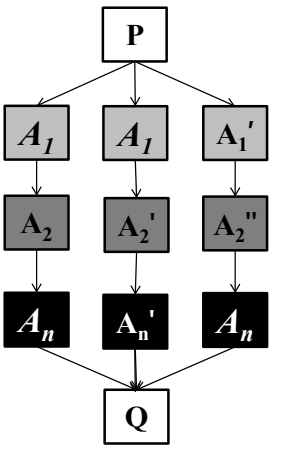


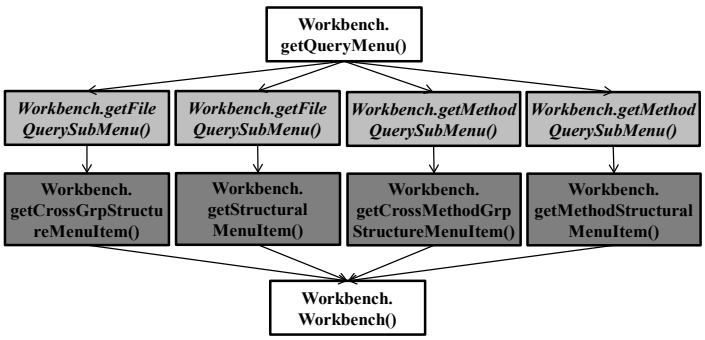


Example of case-4 collaborative pattern from Clone Analyzer v2.0

Length of the collaborative pattern, n = 3
Number of Instances, *I* = 3

(d) Example of Case-4 Collaborative Pattern

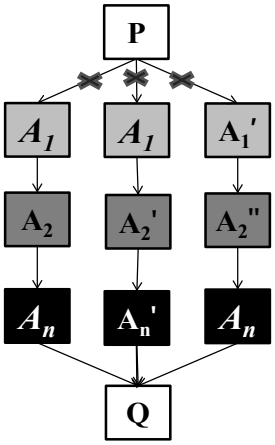


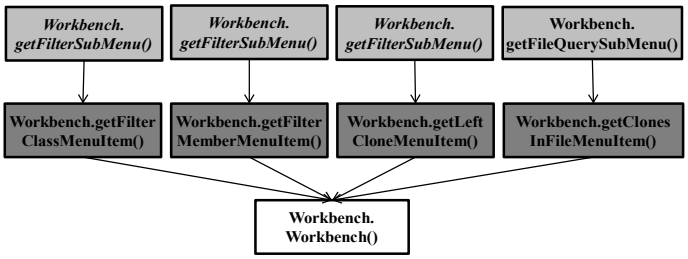


Example of case-5 collaborative pattern from Clone Analyzer v.2.0

Length of the collaborative pattern, n = 2
Number of Instances, *I* = 4

(e) Example of Case-5 Collaborative Pattern

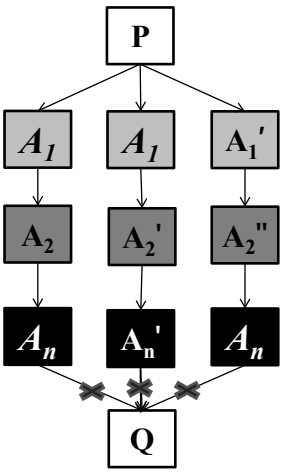


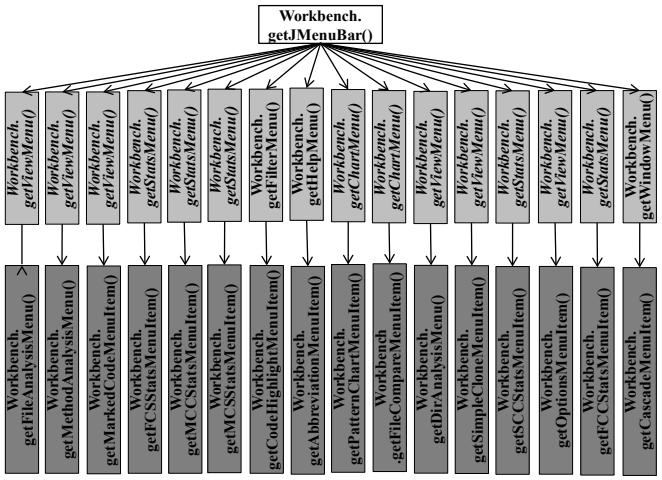


Example of case-6 collaborative pattern from Clone Analyzer v.2.0

Length of the collaborative pattern, n = 2
Number of Instances, *I* = 4

(f) Example of Case-6 Collaborative Pattern

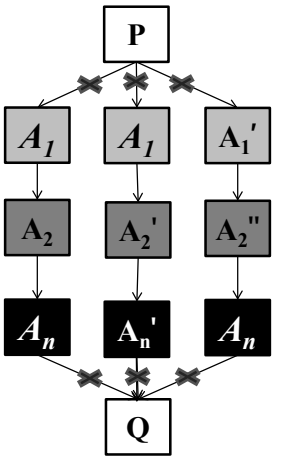


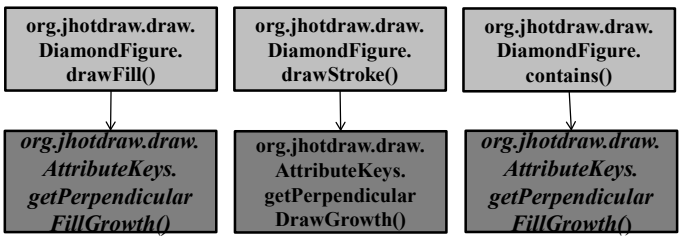


Example of case-7 collaborative pattern from Clone Analyzer v.2.0

Length of the collaborative pattern, n = 2
Number of Instances, *I = 16*

(g) Example of Case-7 Collaborative Pattern

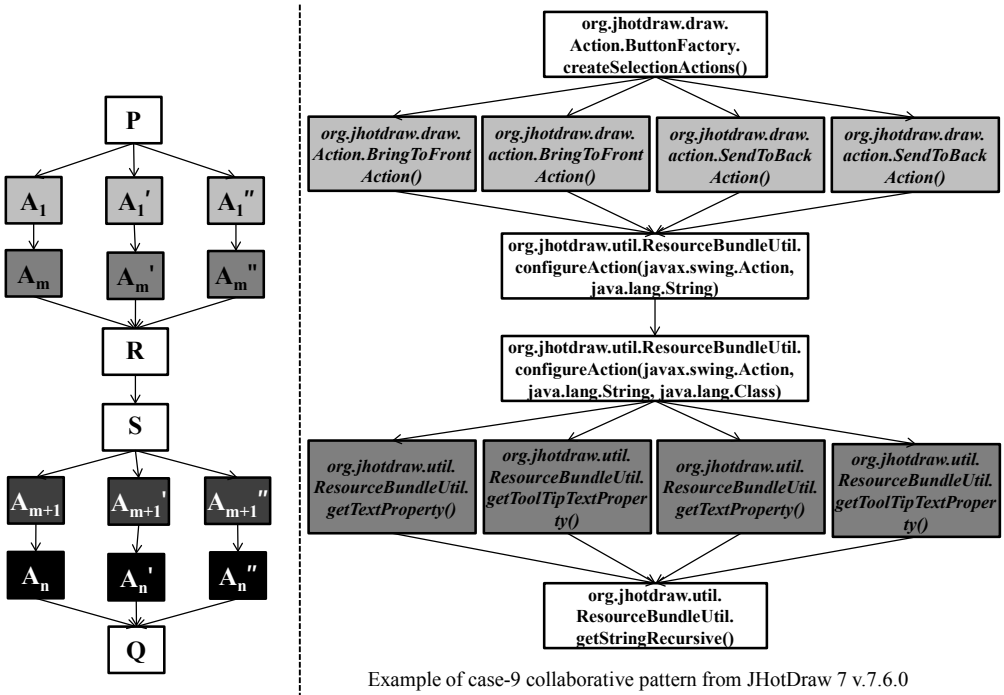




Example of case-8 collaborative pattern from JHotDraw 7 v.7.6.0

Length of the collaborative pattern, n = 2
Number of Instances, *I* = 3

(h) Example of Case-8 Collaborative Pattern



Example of case-9 collaborative pattern from JHotDraw 7 v.7.6.0

Length of the collaborative pattern, n = 2
Number of Instances, *I* = 4

(i) Example of Case-9 Collaborative Pattern

Figure 4.2. Different cases of collaborative patterns emerged from the analysis of case studies

4.3.1. Analysis Results

We analyzed the collaborative patterns detected during the case studies performed on JHotDraw 7 and Clone Analyzer (discussed in previous sections) based on the two criteria: pattern quantity and pattern quality.

Table 3 shows the number of patterns found in JHotDraw 7 and Clone Analyzer for various cases of collaborative patterns illustrated in Figure 4.2.

Due to the absence of any previous work that finds similar types of patterns and unavailability of ground truth data with which we can compare our work, recall cannot be calculated. Hence, due to the unavailability of any suitable reference set where all the collaborative patterns in the software are known, it is speculative to analyze the recall (completeness) of our approach.

Table 3. Summary of collaborative pattern analysis results

Case #	JHotDraw 7	Clone Analyzer
Case-1 Collaborative Patterns	181	7
Case-2 Collaborative Patterns	45	3
Case-3 Collaborative Patterns	15	7
Case-4 Collaborative Patterns	0	1
Case-5 Collaborative Patterns	0	6
Case-6 Collaborative Patterns	3	2
Case-7 Collaborative Patterns	0	1
Case-8 Collaborative Patterns	1	0
Case-9 Collaborative Patterns	3	0
Total Collaborative Patterns	**248**	**27**

The precision (correctness) of the proposed approach depends on the two factors. First is the precision of the technique used for detecting method clones, i.e., Clone Miner. Second is how accurately the method-call chains are generated from the software. In case of the proposed approach, each generated method-call chain always corresponds to a possible calling sequence in the program. Hence, the only factor that affects the precision of the proposed approach is the accuracy of the used Clone Miner. Assuming 100% precision for the Clone Miner, it is not possible for the proposed approach to report collaborative patterns that are actually no collaborative patterns. Hence, the proposed approach shows the same precision as the Clone Miner has. We further analyzed the detected collaborative patterns manually for the false positives and we found no false positives. To analyze the quality of detected collaborative patterns, we considered two factors:

1. Are the instances of a collaborative pattern significantly overlap with the instances of another collaborative pattern?
2. Are the collaborative patterns large enough?

Are the instances of a collaborative pattern significantly overlap with the instances of another collaborative pattern?

Since overlapped clones almost point to the same locations in the source code. Therefore, it is generally considered that overlapped clones are redundant and thus not so useful for developers as compared to non-overlapped clones [62]. Thus, it is useful to explicitly know the locations of overlapped clones. Hence, we analyzed the collaborative patterns found in the Clone Analyzer v2.0 for overlapping collaborative patterns. A collaborative pattern C_1 is overlapping with another collaborative pattern C_2 if all of the following conditions hold:

1. All the first-methods from both collaborative patterns C_1 and C_2 belong to same method-clone class.
2. All the end-methods from both collaborative patterns C_1 and C_2 belong to same method-clone class.
3. If there exists a method, say P, which calls all the first-methods of collaborative pattern C_1, then P must also call all the first-methods collaborative pattern C_2.
4. If there exists a method, say Q, which is called by all the end-methods of collaborative pattern C_1, then Q must also be called by all the end-methods of collaborative pattern C_2.

Out of 27 collaborative patterns detected in the Clone Analyzer v2.0, we found total of three such overlapping collaborative pattern classes, each having two collaborative patterns as members. The remaining 21 collaborative patterns are unique.

Are the collaborative patterns large enough?

Another metric we used for analyzing detected collaborative patterns is the length of the collaborative pattern. Table 4 shows the analysis results.

Table 4. Analysis of the length of patterns detected in clone analyzer v.2.0

Case #	Number of Patterns	Length of Collaborative Pattern (n)		
		n = 1	n = 2	n = 3
Case-1 Collaborative Patterns	7	6	1	0
Case-2 Collaborative Patterns	3	2	1	0
Case-3 Collaborative Patterns	7	6	1	0
Case-4 Collaborative Patterns	1	0	0	1
Case-5 Collaborative Patterns	6	0	6	0
Case-6 Collaborative Patterns	2	0	2	0
Case-7 Collaborative Patterns	1	0	1	0
Case-8 Collaborative Patterns	0	0	0	0
Case-9 Collaborative Patterns	0	0	0	0
Total Patterns	**27**	**14**	**12**	**1**

4.4. Benefits and Applications

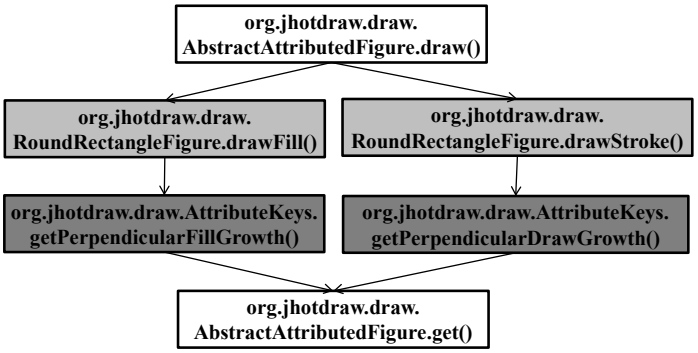
Detection of code clones, in general, helps in program understanding, error detection, refactoring, improving software quality, code compaction, etc. [2, 3, 8]. Collaborative patterns, being large-granular code clones facilitating a better context analysis, can prove to be more beneficial in the above scenarios. This section explores some of the benefits and applications of detecting collaborative patterns.

4.4.1. Better Program Understanding

Code clones reveal important design and implementation information about a software system. Hence, their detection is considered to be a good software engineering practice [3, 4, 129, 130]. Collaborative patterns can be even more beneficial in this regard.

According to [2, 131], if the functionality of a cloned fragment is comprehended, it is possible to have an overall idea on the other files containing similar copies of this fragment. Collaborative patterns are clones of larger granularity. Hence, it is easy to comprehend the functionality of an instance of the detected collaborative pattern. So, we can easily get an approximate idea of the functionality of other files containing the instances of this particular collaborative pattern.

Further, compared to other types of clones, collaborative patterns expose important calling-relationships information between the cloned program structures. For a given collaborative pattern, all the call-sequences start (similarly end) at the same method or at methods that belong to a method-clone class. For example, as shown in Figure 4.3, drawFill() and drawstroke() methods belong to same method-clone class which call other methods (getPerpendicularFillGrowth() and getPerpendicularDrawGrowth() respectively) that belong to another method-clone class. Such type of similar process flows is barely visible with other types of clones. The proposed collaborative pattern detection approach further improves program understanding by automatically tracing the method-calls that lead to or emerge from these collaborative patterns. For example, as shown in Figure 4.3, two instances of the given collaborative pattern are called by same method draw(). They also call the same method get(). By analyzing such process flows, much useful information can be discovered about the system design. Program understanding can be improved further by representing the detected collaborative patterns in the form of non-redundant templates using the ART.



Example of a collaborative pattern from JHotDraw 7 v.7.6.0
Length of the collaborative pattern, n = 2
Number of Instances, $I = 2$

Figure 4.3. Better program understanding: example of a collaborative pattern from JHotDraw7

4.4.2. Enhanced Reuse Opportunity

Collaborative patterns are large-granular program structures. They are large enough to form attractive candidate for reusable components. Also, collaborative patterns often manifest some of important concept or design decisions that were used during the development of the software system. Such design or concept level similarities exhibit opportunities for building reusable components [115]. In product line systems, clones spread across multiple systems. Cross-project clones can be used for reuse optimization [132]. Due to their large-granularity, in such cases, collaborative patterns further give useful indication of reuse opportunities. We can represent clone classes as non-redundant components and reuse them within or across software system. Later, Sections 6.1.5.2–6.1.5.4 illustrate this mechanism of reusing components within a software system or across a product line.

4.4.3. Efficient Refactoring

Among others, the simplest use of detected code clones is to remove them from the software by using refactoring. Refactoring allows improving the

design of the software without changing its functionality [133]. Refactoring clones helps in decreasing the complexity of the software, and reducing sources of errors emerging from these cloned program structures [2].

It is suggested that not every single occurrence of clones can be refactored. In fact, many of them are too complex or simply not refactorable [134]. It is found that there are types of clones where refactoring would not help [18]. However, research supports that there is still great potential for advancements in the area of software clone refactoring [135-137]. It is especially true when code clones are of large granularity representing high-level system concepts (for example, structural clones [5]).

Being high-level clones of large granularity, detection and analysis of collaborative patterns is useful in finding locations in software where large-granular duplications are present, and those can be refactored. After choosing these large-granular code duplications for refactoring, code clones (such as method clones) constituting the collaborative patterns can be relatively easily refactored because of the knowledge of the context. More specifically, one of the possible refactoring strategies is to move together several cloned methods linked by the calling-relations to the parent class or simply change the inheritance structure to remove the cloned methods.

4.4.4. Other Benefits

Improved Clone Detection: Small-granular code clone detectors generally detect code clones larger than a certain threshold value (e.g., 30–50 tokens or 4–6 lines of code [42, 100]). Large-granular code clone detectors group small-granular duplicated contiguous code fragments into larger cloned program

entities. For example, structural clones are cloned program entities that represent recurring patterns of duplicated contiguous code fragments occurring in a method, across methods, in a file, or across files [5]. The proposed approach improves the clone detection further by exploring the calling-relations among these cloned entities.

Good Candidates for Library: Davey et al. [40] claimed that if a program structure is cloned on several occasions in the software, the program structure has proved its usability. Hence, it can be incorporated in a library providing an effective set of reusable components. In the same fashion, we argue that collaborative patterns being large-granular program structures can prove to be even more effective and useful as a library candidate provided they also occur at several occasions. However, we have not found any example of collaborative pattern through our analysis that can be considered as a candidate for incorporating as a library.

4.5. Conclusions

In this chapter, we presented the experimentation pertaining to detection and analysis of collaborative patterns. We explored different possibilities and applications where detection of collaborative patterns can prove to be useful. In the next chapter, we discuss the mechanism of managing code clones of large granularity such as collaborative patterns, structural clones, and other types of software clones in detail.

Chapter 5.

MANAGING CODE CLONES USING THE ART

In the previous chapters, we presented the proposed approach for detecting collaborative patterns in detail. Once we found code clones (for example, collaborative patterns, simple clones, structural clones, or any other types of code clones) in the software—the questions arise how are we going to manage them, how can we benefit from their knowledge in terms of easier maintenance or better reuse. This chapter and the next chapter deal with all such aspects pertaining to clone management.

Code clones obstruct program understanding and increase maintenance costs [2, 3]. While we may not be able to eliminate all these clones in a software system, these can be dealt with significantly at the meta-level [138]. With this understanding, we propose a solution to manage big clones (i.e., code clones of large granularity such as collaborative patterns, structural clones, or other large-granular cloned program structures) by presenting a meta-programming technique and tool, the ART (Adaptive Reuse Technique), that can manage

families of redundant software systems by providing a common base of non-redundant, adaptable, and reusable meta-components. These meta-components are easier to reuse, maintain, and comprehend, and can track the program changes during evolution.

The chapter is organized as follows: after providing introduction and motivation in Section 5.1, Section 5.2 provides a detailed description of the ART. The detailed research methodology for managing code clones using the ART is presented in Section 5.3. Finally, Section 5.4 concludes the chapter.

5.1. Introduction and Motivation

There is a large body of research on various reasons why code clones arise—both across and within system versions—and whether code clones are good or bad [17, 137-140]. These studies show that designers may intentionally create certain clones to fulfill some design goals (e.g., for performance or readability) [17]. Other clones may result from careless design and can be refactored [137, 139], and yet others may not play any useful role but cannot be eliminated using conventional design techniques [140]. Nevertheless, cloning is a reality and there is need to deal with it [102]. It is beneficial to know the locations of code clones in the programs—how they are similar, and how the clone instances differ from one another. This is particularly true for big clones, which arise even if software evolution is systematically managed in a reuse-based manner with variability management techniques [141]. A study of industrial systems has shown that around 50% of small cloned code fragments tend to be contained in big clones [4]. While big clones are certainly intentional, they contribute to increased program size and complexity [4].

Clone management aims at organizing existing clones, minimizing their negative effects, controlling their growth and dispersal, and avoiding them altogether [102]. Giesecke [142] divided clone management into three categories: preventive, corrective, and compensatory. Preventive clone management aims at preventing the introduction of new clones into a software system. Corrective clone management deals with removing the existing clones from the software system. Compensatory clone management aims at minimizing negative effects of existing clones that cannot be removed from the software system [143].

As mentioned above, corrective clone management techniques aim at removing the existing clones from the software system. Most of the clone management techniques proposed in the literature fall under this category. It includes refactoring, macros, generics, higher order functions, etc. [41, 133-135, 137, 144-146]. However, sometimes this objective is not always feasible. Especially, as mentioned above, if clones are created intentionally for better performance or readability, it is not wise to remove them altogether [17]. Also, removing clones with techniques such as refactoring may result in the system design conflicting with other important design goals [22]. Therefore, there is a lot of literature on whether to refactor, what to refactor, or what not to refactor [18, 134-137]. Clones can be automatically transformed by replacing clones with macros (pre-processor commands) [41]. But, program instrumented with macros might significantly decrease the comprehensibility of the source code [147].

Although it may not always be possible to eliminate all the clones from a software system [138], these can be dealt with using a generic representation

of the software. With this understanding as motivation, we propose a meta-programming technique and tool, the ART (Adaptive Reuse Technique), that can be effectively used to manage big clones within or across versions of a software system at the meta-level. The ART is an enhanced, lightweight, and XML-free version of the XVCL (XML-based Variant Configuration Language) [118]. It manages big clones by representing them in the form of non-redundant, adaptable, and reusable templates, called ART templates. ART templates can be built for groups of similar program structures of different kind (e.g., methods, files, or directories) that differ in terms of the variety of ways typically found in the real systems.

Compared to corrective clone management techniques such as refactoring and macros, which aim at removing clones to handle them, the ART provides a compensatory clone management solution. It means that the ART aims at minimizing the negative impacts of clones (especially big clones) without actually removing them. The ART actually does not eliminate clones from runtime code, but effectively deals by unifying them at meta-program level. It offers enhanced software maintenance by providing two-fold view of the software system: one is a clone-free source code in the form of ART templates for easier maintenance, and another is an executable code with those useful clones that should be kept in the software system during runtime for performance reasons.

The benefits of the ART include simplification of SPL core assets due to non-redundancy, productivity gain due to concise template representation of programs, and easier comprehension and traceability of change impact during SPL evolution. In various similarity groups, depending on the cloning, the

proposed technique eliminates 25–75% of the code by unifying clones into non-redundant templates. Unification of clones further improves program understanding. Program relations that have to do with the impact of changes are important in program understanding, maintenance, and evolution, but remain mostly implicit in conventional programs. ART templates expose and explicate some of these program relations. For example, when maintaining duplicated code, we often must know where such duplicates are and how they are different, in order to decide if and how each of them should be modified. The ART makes such information more visible and tractable, reducing the risk of unexpected errors when changing programs.

This chapter provides details of the ART and the methodology of managing clones using it. Quantitative and qualitative evaluation of the strengths, weaknesses, and trade-offs involved in the application of the ART is explored in the next chapter.

5.2. An Overview of the ART

The ART is a meta-programming technique and tool that can be effectively used to manage clones within or across versions of a software system at the meta-level. It is an enhanced and lightweight version of the XVCL [118]. XVCL is a dialect of XML. So, it is necessary to know XML syntax and rules before understanding and working with the XVCL. The ART parts with XML syntax and processing. It uses a C Preprocessor (cpp) [148] based flexible and more readable syntax. The ART syntax is flexible in the sense that it offers the capability to redefine the syntax as and when needed by the users. This is particularly useful when reserved words from the ART syntax conflict with the

reserved words of the native language. In such cases, ART syntax can be easily changed and users can define their own syntax. Hence, the ART offers user-defined syntax. Comparison of the ART with the XVCL, preprocessors, and other related techniques is presented in the related works section in Chapter 6.

5.2.1. How Does the ART work?

The ART works on the principle of representing each clone class found in the software system in the form of non-redundant, adaptable, and reusable meta-components called ART templates. An ART template is a file with original program code (i.e., native language of the software) instrumented with ART commands (explained in detail in Section 5.2.2) for ease of customization. These ART templates can be converted back to the clone classes using the ART Processor. The ART Processor takes the ART templates as input and generates the instances of the clone classes as output. In this way, as mentioned in Section 5.1, the ART offers enhanced software maintenance by providing two-fold view of the software system: one is a clone-free source code in the form of ART templates, and another is an executable code with those useful clones that should be kept in the software system during runtime for performance reasons. Next subsection discusses the ART-template solution in detail.

5.2.1.1. An Overview of the ART-Template Solution

For each of the detected clone class, we distill common code into ART templates and mark the locations of variation points using ART commands. Figure 5.1 outlines the overall solution, which consists of an ART-template hierarchy in which templates at the lower-level serve as building blocks for the

higher-level templates. The ART templates are linked together by #adapt commands. The top-most template, called the specification file (SPC), specifies how to adapt other templates lower in the hierarchy to accommodate required variations. The ART Processor checks the templates for their conformance to the grammar definitions. It then traverses the template hierarchy in the depth-first order, starting from the SPC, and performs adaptations by executing the ART commands embedded in the SPC and other ART templates. During traversal, each ART template adapts other templates from its sub-hierarchy. At the end, the ART Processor produces the required cloned instances.

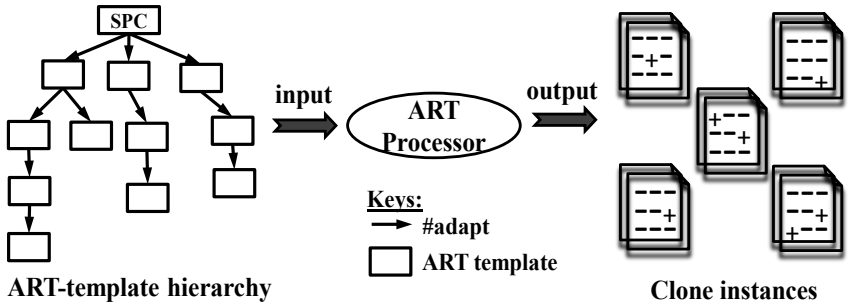


Figure 5.1. An overview of the ART-template solution

The flow diagram depicting the various steps of ART-template processing is shown in Figure 5.2. The Processor starts by reading the SPC (step-1). It fetches the ART commands step-by-step in the order in which they appear in the SPC (step-2). Whenever it hits #adapt command (step-3), the processing will switch immediately to the adapted template (step-4) and switch back when the adapted template finishes its processing. Within a template, each ART command is processed one after another, in the same way as in the SPC. For the other commands, the Processor executes the ART command and builds the output (step-4') incrementally. Once the Processor reaches the end of the

SPC (step-5), it generates the required source code files (step-6); if not, the ART Processor fetches the next ART command from the SPC (step-6').

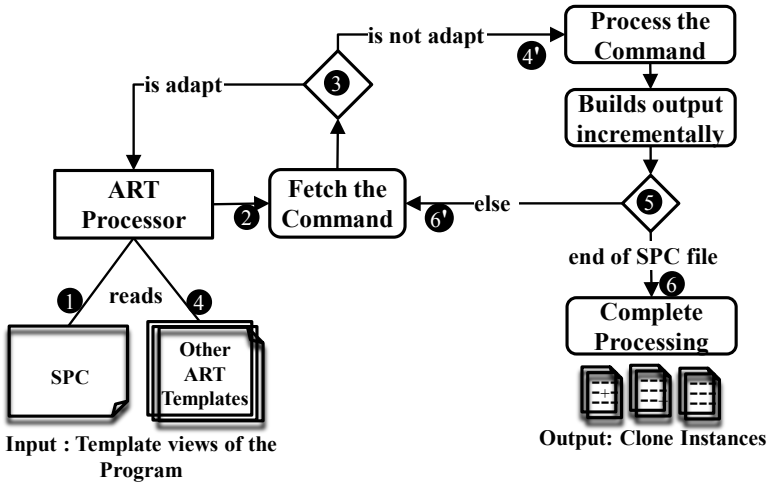


Figure 5.2. Traversal mechanism of the ART Processor

Figure 5.3 shows an example to illustrate the ART-Processor traversal mechanism.

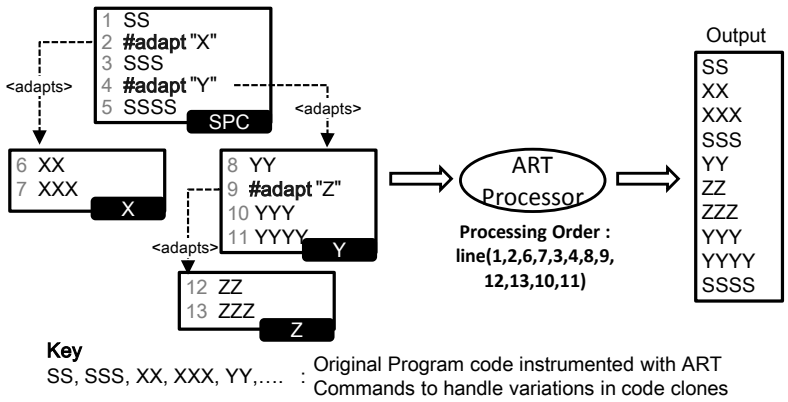


Figure 5.3. Example illustrating the ART Processor traversal mechanism

The ART Processor starts processing at line 1 (in the SPC). It emits the code to the output file, and then executes the command #adapt "X" (line 2). It suspends the processing of the SPC, and transfers processing to template X. The ART Processor emits code from lines 6 and 7 to the output file, and returns to the SPC (line 3). It then emits code (line 3) to the output file. Next,

due to the execution of the command #adapt "Y" (line 4), the execution of the SPC is suspended and processing transfers to template Y. While executing the template Y, it emits code (line 8) to the output file, and suspends execution of the template Y and jumps to template Z due to the execution of the command #adapt "Z" (line 9). The ART Processor continues processing this way until the end of the SPC (line 5).

A prominent feature of the ART is that it blends in a non-conflicting way with the underlying programming language. It is because the ART syntax is user-defined. It makes it easy to use without affecting already existing software solutions and the people who work with them.

5.2.2. ART Command Set

This section presents each of the ART commands in detail.

5.2.2.1. Comments in the ART

A single-line ART comment can be written by preceding the symbol %, for example:

```
% This is a single-line ART comment
```

Multi-line ART comments are written between %> and <%, for example:

```
%> This is a multi-line
 ART comment <%
```

5.2.2.2. #adapt Command

#adapt command inside an ART template (say template B) instructs the ART processor to:

- Suspend the processing of the current ART template (i.e., template B).
- Process the ART template specified by its attribute name, say template_name
- Process all the descendent ART templates in the hierarchy to the adapted ART template (i.e., specified by template_name)
- If applicable, perform all the customizations specified under the body of #adapt command to the visited ART templates.
- Once processing of adapted ART template and its descendent ART template finish, return control back to the current ART template (i.e., template B).

The ART does not support recursive adaptation. It means that an ART template is not allowed to adapt itself or any of its ancestors' ART templates.

Based on whether customizations have to be applied or not to the adapted templates, the ART has two variations for #adapt command. Without any specified customizations, #adapt has following format:

```
% simple adapt
#adapt template_name
```

We can specify customizations that should be applied to the adapted templates under extended #adapt command as follows:

```
% extended adapt
# adapt: template_name
    <customizations>
#endadapt
```

It is to note that colon character ":" is compulsory after the "#adapt" keyword. Also, any of the ART commands can be included under the "customizations" field. ART Processor applies the specified customizations to the designated templates and proceeds to process them.

ART command #adapt corresponds to #include directive in cpp that supports macro invocation. However, unlike #include directive, #adapt command allows the same source file to be customized differently (using extended adapt) in different scenarios in which it is reused.

5.2.2.3. ART Variables and Expressions

ART variables can be declared using #set command. Using #set command, we can declare both single-valued as well as multi-valued variables.

A single-valued variable can be an integer, expression, or string as below:

```
#set var1 = 2                % assigns integer value 2 to var1
#set var2 = var1 + 1         % assigns value 3 to var2
#set var3 = "Text"           % assigns string "Text" to var3
```

Note that string values must be in double quotes (").

A multi-valued variable can be declared using the same #set command, but the values are separated by commas as below:

```
#set var4 = 2, 3, 4
#set var5 = "Text1", "Text 2", "Text3"
```

ART expressions can appear anywhere in ART templates. An ART expression is enclosed between question mark "?" symbols. Value of an ART variable can be referred by placing "@" symbol in front of the variable. For example,

expression “?@var1?” refers to the value of variable var1. Each extra “@” in front of an ART variable indicates another level of indirection, for example:

```
#set y = "x"
#set x = "z"
#set z = "w"
?@y?          % @y = value of y = x
?@@y?         % @@y = value of (value of y) = value of x = z
?@@@y?        % @@@y = @@x = @z = w
```

#set command in the ART corresponds to #define directive in cpp. However, the ART has different scoping rules as compared to cpp. ART variables allow the variable values to be propagated along the adapted templates. The first declaration of an ART variable in a template overrides any subsequent declarations of the same variable in all the adapted templates, unless the same variable is redefined in the template again. For example, as shown in Figure 5.4, ART template SPC declares a variable “var” with value 4 (line 2). So, line 3 outputs “var” value to be 4 (line 10). The #set command in adapted template A.art (line 8) is ignored and line 9 outputs value to be 4 (line 11). Line 5 redefines the value of “var” in the same template, i.e., the SPC. So, line 6 outputs new value which is 5 (line 12). The adapted template A.art now outputs this new value (i.e., 5 in line 13) while processing line 9.

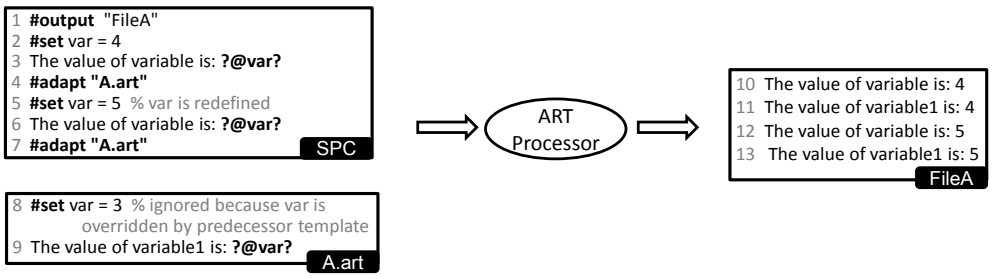


Figure 5.4. Example: #set command and variables in the ART

There are three types of expressions in the ART. These are name expression, string expressions, and arithmetic expression.

A simple name expression may contain just a variable reference, such as: ?@C? or ?@@C?

More complex (but more useful) name expressions can be written as: ?@A@B@C?. In this case, the value of the name expression is computed from right to left as follows:

```
value of (A <> value of (B <> value of (C) ))    %  symbol '<>' means string
                                                     concatenation.
```

Referenced variable names created at each intermediate evaluation step must represent variables that exist in processing flow. Otherwise, the ART processor reports an error. For example:

```
#set A = "B"
#set B = "C"
#set C = "D"
#set D = "F"
#set BD = "G"
#set AG = "H"
?@C?              % @C = value of C = D
?@A@B@C?          % @A@B@C = @A@BD =@AG = H
?@@@C?            % ERROR: @@@C = @@D = @F = ? (variable does not
                  exist in process flow)
```

A string expression is concatenations of name expressions and strings. In order to evaluate a string expression, ART Processor first evaluates the name

expressions from left to the right. It then replaces the name expressions with their respective values in the string expression.

For example, with reference to the above example, string expression ?@A@B@C?"Text"?@D? is evaluated as:

- Evaluate name expression ?@A@B@C?. It results in H. Concatenate this with string “Text”. It results in HText.
- Evaluate name expression ?@D?. It results in F. Concatenate this with HText. Final Output is HTextF.

Arithmetic expressions are well-formed expressions that can contain ‘+’, ‘-’, ‘*’, ‘/’ operators and nested parenthesis. It is not allowed to use arithmetic expressions intermixed with name expressions or string expressions.

In arithmetic expressions, ART variables can be referenced simply by referring to their names (instead of using “@” symbol). For example, ?a + (b + 2)? is valid arithmetic expression where a and b are ART variables as shown below:

```
#set a = 2
#set b = 4
Value of c is = ?a + (b + 2)?
```

In this case, output is: Value of c is = 8

5.2.2.4. #output Command

For an ART-template solution, ART Processor interprets the ART commands and emits any source code found in the processed ART templates to output file(s). Path of such output file(s) can be specified using #output command (Figure 5.5(a)). The path can be absolute or relative path. However, this

command is optional. In case this command is not used, the ART Processor emits the code to an automatically generated default file named defaultOutput.txt (Figure 5.5(b)). ART Processor creates defaultOutput.txt file in the main folder of its installation.

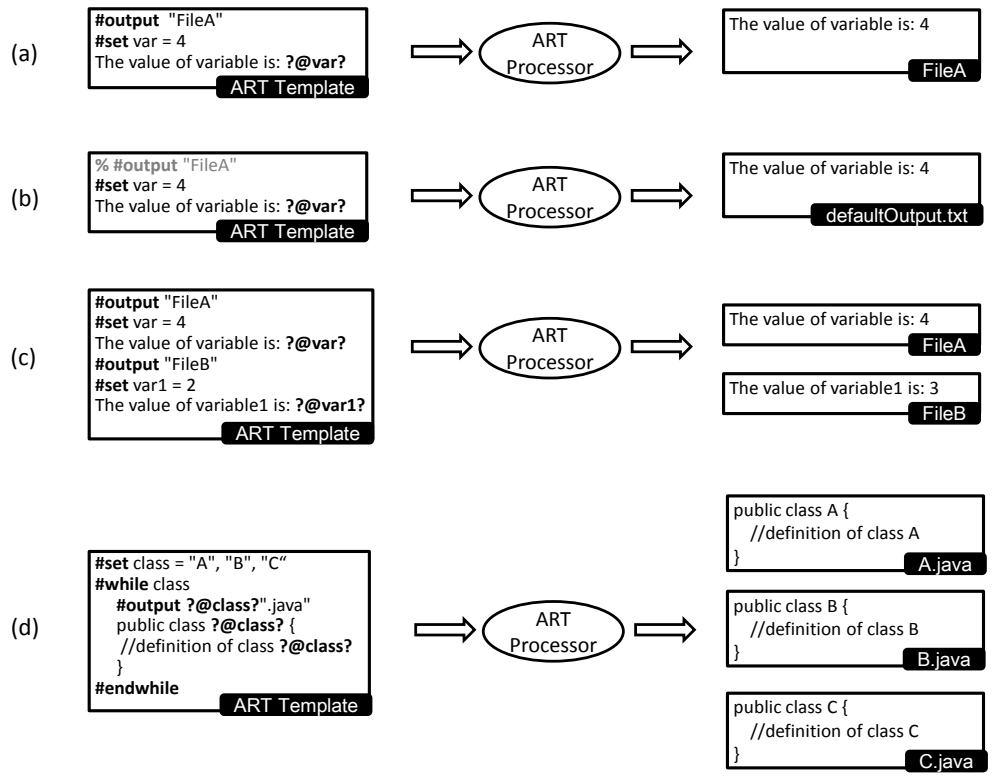


Figure 5.5. Example: #output command in the ART

The ART allows using multiple #output commands in a template or across template. Once ART Processor executes the "#output file_f", it emits subsequent output in the file file_f, until the next #output command overrides the file_f with another file name (Figure 5.5(c)). When ART Processor encounters the line "#output file_f" for the first time, it checks whether file file_f exists or not. If file file_f does not exist, ART Processor creates the file and emits the output to it. Otherwise, the content of the file is overridden by the new emitted content. In subsequent processing, if any other #output

command refers again to the same output file (i.e., file_f), the new emitted content is concatenated to the file.

Using ART variables, it is possible to emit source code to multiple output files using single #output command. For example, as shown in Figure 5.5(d), an ART variable "class" is declared with three values. In each iteration of the while loop, ART Processor creates a new file and emits the source code to the created file.

5.2.2.5. Loops and Selections

The ART implements loops and selections using #while and #select commands, respectively. #while command is a generation loop that iterates over its body and generates custom text in each iteration. #select command allows choosing one of many customization options.

A #while loop can be controlled by using one or more multi-value ART variables. It is to mention that all the multi-value variables listed as control variables must have the same number of values. Then, in i^{th} iteration of the loop, i^{th} value from each of the control variable is used. The ART Processor starts the loop with index-value of 1, increments the value of index by 1 in each iteration, and terminates by processing the last value of each of the multi-value variables. Further, it is also possible to specify the name of control variable in the #while loop using expressions (as shown in Figure 5.6, line 5).

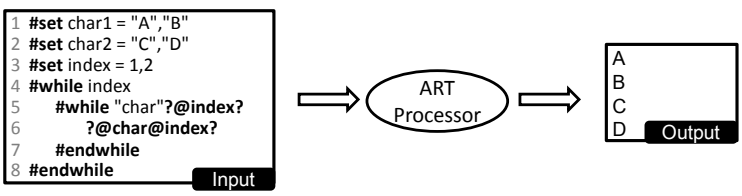


Figure 5.6. Example: #while command in the ART

Using #select command, depending on the value of a control variable, we can select one of many options. Options are selected based on the value of the control variable specified as attribute in #option clause. Figure 5.7 shows an illustrative example for #select command. As shown in the figure, besides #option clauses, #select command can include optional #option-undefined and #otherwise clauses. #option-undefined clause is processed if control variable is undefined. If none of the #option**s** are selected, then #otherwise clause is processed by the ART Processor. We can use "|" symbol to specify more values to a control variable. For example, "#option Second | Third" is processed if value of the control variable index is Second or Third.

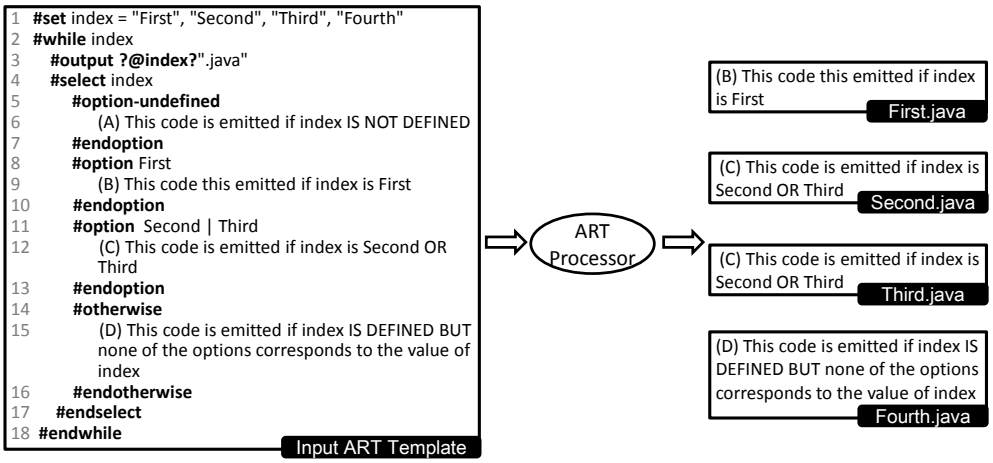


Figure 5.7. Example: #select command in the ART

5.2.2.6. Breakpoints (Insert-Break Mechanism)

The ART supports breakpoint mechanism. Breakpoints serve as anchors where additional code can be injected. It makes the ART capable of handling unexpected variations during evolution. Breakpoints can be marked using #break command. These breakpoints can be easily customized, i.e., additional code can be easily injected by using #insert, #insert-before, and #insert-after commands.

#break command has two variations as shown below. The content under #break is the default content. If there is no matching insert command, then the break's default content is processed. Matching is done based on the specified name (breakX in the example).

(1) % simple break command

```
#break breakX
```

(2) % extended break command

```
#break: breakX          %  note that colon (i.e, :) is compulsory
    default-content
#endbreak
```

There are three types of insert commands to modify the templates at the breakpoints identified by matching #break command. #insert command replaces the default-content of all the matching #break commands with its content. #insert-before command inserts its content before the matching #break command. Similarly, #insert-after command adds its content after the matching #break command. It is to mention that #insert-before and #insert-after commands do not replace the default-content inside its matching #break. Also, a single #break can be simultaneously extended by all three types of insert commands (i.e., #insert, #insert-before, and #insert-after commands).

Figure 5.8 shows illustrative example of insert-break mechanism. As shown in the figure, all the insert commands in lines 3–11 are processed with matching breakpoint breakABC. The ART Processor emits the output as shown by lines 1–3 of the output file FileA. In case, there is no #insert that matches a #break (e.g., breakDEF), then the break's default-content is processed. In this case, the ART Processor emits the output as shown by lines 4–6 of the output file.

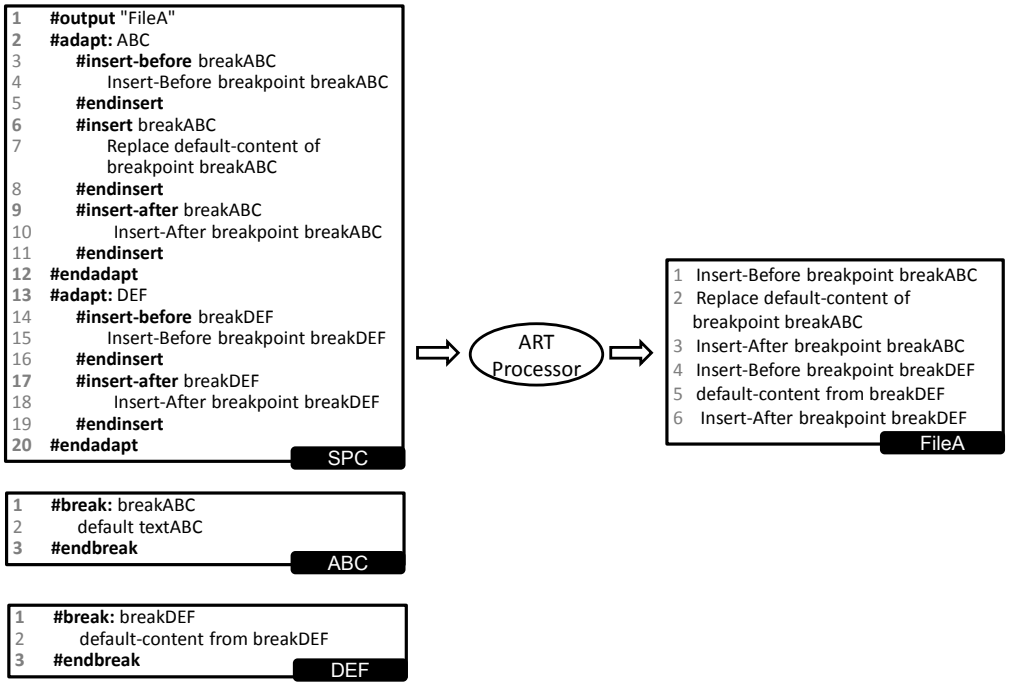


Figure 5.8. Example: breakpoints in the ART

5.2.2.7. Setloop Mechanism

Many multi-value ART variables can be used to control #while loops. Each iteration of the loop uses the i^{th} value of each of the control variables. But keeping track of the corresponding values becomes troublesome, especially when variables have many values that are often changed. Any mismatch of values may cause an annoying error. #setloop command alleviates this problem by allowing us to organize the values of the control variables to be used in a while loop in a more intuitive and less error prone way than multi-value variables do.

The basic usage scenarios for this command can be directly translated into #set commands that control #while in a usual way. Suppose we have:

```
#set x = "x1", "x2", "x3"

#set y = "y1", "y2", "y3"

#set z = "z1", "z2", "z3"
```

```
#while x, y, z
```

Then, instead of specifying a list of all the values one control variable will have over all iterations , #setloop provides a list of iterations and specifies the value of each variable per iteration as shown below:

```
#setloop loopA
    #iter  x = "x1", y = "y1", z = "z1"
    #iter  x = "x2", y = "y2", z = "z2"
    #iter  x = "x3", y = "y3", z = "z3"
#endsetloop
#while loopA
```

#setloop command contains one or more #iter elements. Each #iter element specifies values of control variables to be used in an iteration of a while loop. Values specified in i^{th} #iter element are used in i^{th} iteration of the loop.

“loopA” in the above example serves as an id of the #setlloop. Loop control variables declared inside a #setloop (using #iter) can be referred to (read-only) from outside as if they were multi-value variables declared at the location of the #setloop command, e.g.:

```
loopA.x -- where loopA is a loop-name and x is one of its control variables.
```

It follows that we can also have a loop that uses any selected control variables defined in some #setloop, e.g.:

```
#while loopA.x
```

In basic usage scenario, each #iter element contains one or more “variable=value” pairs. But, it is also possible to set default values for control variables in case a value of a given variable is not explicitly stated in the #iter

element. It is feasible by using an optional #vars clause in the #setloop that allows us to specify control variables with their optional default values. Only one default value per control variable can be specified. However, in the absence of #vars clause, each #iter must specify values for all control variables. In case we have #vars, then all the control variables defined in #iter elements must be also listed in #vars, whether or not they have default values.

In case the same value of a given control variable need to be used in a number of iterations, an optional #vars clause can simplify loop specifications by providing default values. Whenever the value of a given variable is not specified in a #iter, the default value is used. For example,

```
#setloop loopA
    #vars
        #var x = "x-dafault"
        #var y = "y-default"
    #endvars
    #iter x = "x1", y = "y1"
    #iter x = "x2"              % this iteration uses default value of y
    #iter y = "y3"              % this iteration uses default value of x
#endsetloop
```

In this case, iterations 2 and 3 use default values of y and x, respectively.

It is true that only one default value per control variable can be specified. But, a #setloop can be modified using the insert-break mechanism. This approach allows us to modify loop iterations as shown by the example of Figure 5.9. In this example, the #setloop defines following values:

Iteration 1 (line 7 in setloop template): x = x1, y = y1, z = z-default

Iteration 2 (line 8 in setloop template): x = x2, y = y-default, z = z-default

Iteration 3 (line 9 in setloop template): x = x-insert, y = y-insert, z = z-default

Iteration 4 (line 10 in setloop template): x = x-insert, y = y-default, z = z-insert

Such cases are very difficult to handle with #set commands that control #while in a usual way.

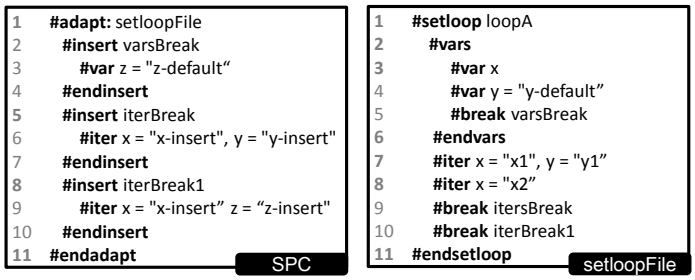


Figure 5.9. Example: setloop mechanism in the ART

Table 5 gives summary of selected ART commands.

Table 5. Summary of selected ART commands

Syntax	Command Definition
#adapt* template_name** **or:** ***#adapt*: template_name** **<customizations>** ***#endadapt	*#adapt* command instructs the ART processor to adapt the named template and its descendants. *#adapt* may also allows to specify customizations that should be applied to the adapted template. Customizations may include any ART commands.
***#output* pathname**	*#output* command specifies the path of the output file where the source code should be placed. The pathname can be absolute or relative path. If the output file is not specified, then the ART Processor emits the code to an automatically generated default file named *defaultOutput.txt* in the main folder of the installed ART processor.
***#set* var_name = val1[,val2,val3, …]**	*#set* command declares an ART variable "var_name" and sets its value to a single or multi-values.
?@var_name?	A direct reference to the value of variable "var_name". Each extra '@' symbol in the front of a variable name indicates an extra level of indirection.
***#break* breakX** **or:** ***#break*: breakX** **default content**	*#break* marks a breakpoint at which changes can be made by ancestor template via *#insert*, *#insert_before*, *#insert_after* commands. The content under *#break* is the default content. If no *#insert* matches a *#break*, then the break's default content

Syntax	Command Definition
#endbreak	is processed.
#insert **breakX** **content_body** ***#endinsert***	*#insert* command replaces all matching *#break*s with its content. Matching is done by a name (breakX in the example).
#insert-before **breakX** **content_body** ***#endinsert***	*#insert-before* and *#insert-after* add their content before or after the matching *#break*s, without deleting their content.
#insert-after **breakX** **content_body** ***#endinsert***	A single *#break* may be simultaneously extended by *#insert*, *#insert-before* and *#insert-after* commands.
#while **var1[,…,varN]** **content_body** ***#endwhile***	*#while* is a generation loop that iterates over its body and generates custom text at each iteration.
#select **control_var** ***#option*** **option** **option_body** ***#endselect***	*#select* allows us to choose one of the many customization options.
% comment **%> comments <%**	Single line comment Multiple lines comments

5.2.3. ART Syntax

In this section, we describe syntactical structure for each of the ART commands. We use following notations to specify the syntax of ART commands:

- Definition symbol is ::=, e.g., A ::= B
- Alternate symbol is |, e.g., A ::= B | C
- 0 or more times repetition symbol is *, e.g., A ::= B*
- 1 or more times repetition symbol is +, e.g., A ::= B+
- Optional part symbol is square bracket […], e.g., A ::= [B] C
- Grouping is symbolized by round brackets (…), e.g., A ::= (BC)*
- Non-terminal symbols are written with a mixture of uppercase letter, lowercase letter, digits and a special symbol -.
- Terminal symbols are keywords, special symbols etc.

- Special sequence is symbolized using ?...?, e.g., STRING ::= ? Mixture of any characters ?

Comments in the ART

comment ::= '%' SINGLELINE-TEXT |
'%>' (SINGLELINE-TEXT | MULTILINE-TEXT) '<%'

SINGLELINE-TEXT ::= ? Mixture of any characters in a single line ?

MULTILINE-TEXT ::= ?Mixture of any characters that may spread over many lines?

#adapt Command

adapt ::= as-is-adapt | extended-adapt

as-is-adapt ::= '#adapt' path

extended-adapt ::= '#adapt:' path
adapt-body
'#endadapt'

path ::= Expression | VAR-NAME | STRING

adapt-body ::= (command)*

#set Command

set ::= '#set' VAR-NAME = value (, value)*

value ::= Expression | VAR-NAME | STRING | INTEGER

#output Command

output ::= '#output' path

path ::= Expression | VAR-NAME | STRING

#while Command

while ::= '#while' control-var (, control-var)*

while-body

#endwhile

control-var ::= Expression | VAR-NAME

while-body ::= (textual-content | command)*

#select Command

select ::= '#select' control-var

['#option-undefined'

option-body

'#endoption-undefined']

('#option' value (| value)*

option-body

'#endoption')*

['#otherwise'

option-body

'#endotherwise']

#endselect

control-var ::= Expression | VAR-NAME

value ::= Expression

option-body ::= (textual-content | command)*

#insert and #break Commands

insert ::= '#insert'['-before' | '-after'] break-name

insert-content

#endinsert

break-name ::= Expression

insert-content ::= (textual-content | command)*

break ::= ‘#break’ break-name

break-content

#endbreak

break-content ::= (textual-content | commad)*

#setloop Command

setloop ::= ‘#setloop’ setloop-name

[setloop-vars]

(‘#iter’ iter-desc)+

[break]

‘#endsetloop’

setloop-name ::= VAR-NAME

setloop-vars ::= ‘#vars’

(‘#var’ VAR-NAME [= value])+

[break]

‘#endvars’

iter-desc ::= VAR-NAME = value (, VAR-NAME = value)*

value ::= Expression | VAR-NAME | STRING | INTEGER

Expression

Expression ::= Arithmetic-Expression | Name-Expression |

String-Expression

Name-Expression ::= ‘?’ ‘@’ (VAR-NAME | ‘@’)* VAR-NAME ‘?’

String-Expression ::= (STRING* Name-Expression+ STRING*)+

Arithmetic-Expression ::= <syntax for arithmetic expressions is same as in C preprocessor (‘+’, ‘-‘, ‘*’, ‘/’, nested parenthesis)

Command

command ::= Any one of the ART commands

VAR-NAME

VAR-NAME ::= (any letter | '_') (any letter | number | '_')*

ART Processor is case-sensitive.

STRING

STRING ::= ? Mixture of any characters ?

5.2.4. Architecture and Implementation Details

Figure 5.10 shows architecture of the ART Processor. It takes the specification file (SPC) as input from the user through the ART User-Interface module. In the next step, the ART Lexer module, which is a lexical analysis tool in the ART, converts the SPC and the other adapted ART templates into sequences of tokens. These token sequences are then parsed by the ART Parser in accordance with the ART grammar rules, and an abstract syntax tree is generated as output. The ART Parser recognizes the ART commands only and skips any other code or text. It helps in integrating the ART code with other programming languages in an unrestricted form. At the end, the generated abstract syntax tree is evaluated using the ART Evaluator module to get the required clone instances.

The ART Processor is implemented in Java and is available in a ready-to-use form (available at: http://art.comp.nus.edu.sg/). The lexical analyzer for the ART Processor is built by adapting ANTLR [149]. The ART Processor can be run from command-line mode as well as using graphical user interface mode.

It is also supported by editor plug-ins for Notepad++ and Microsoft Visual Studio.

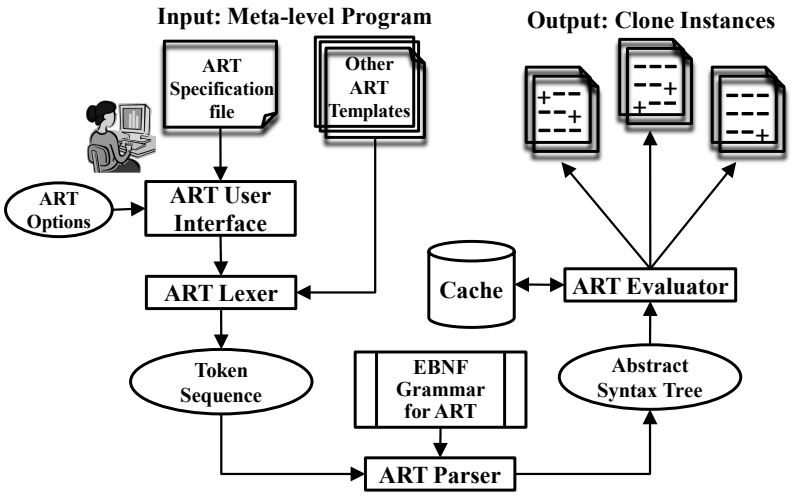


Figure 5.10. Architectural overview of the ART Processor

5.3. Detailed Methodology

The research methodology for managing code clones using the ART consists of four major steps as shown in Figure 5.11.

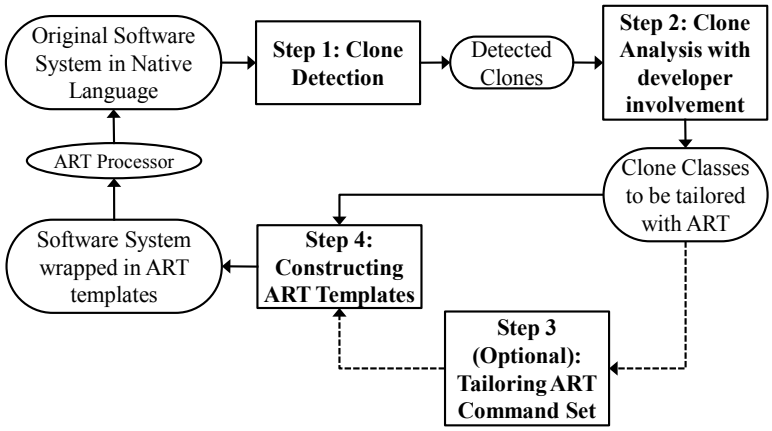


Figure 5.11. Detailed research methodology for managing code clones

5.3.1. Step 1: Clone Detection

The first step deals with the detection of code clones (small cloned code fragments as well as big clones) from the software system. Developers can use

any clone detection techniques and tools of their choice for detecting code clones. However, the usefulness of the ART directly depends on the accuracy of the used clone detector. The better the recall and precision of the clone detector, the higher the likelihood of finding the best clone classes whose template views can benefit developers.

In addition to the selection of a proper clone detector, another factor that affects the usefulness of the ART is the choice of setting the minimum size of a clone that should be detected by the clone detector. For token-based clone detectors, Kamiya et al. [42] suggests using the minimum value of 30 tokens to obtain meaningful results. We also consider it to be a suitable value for our experimentation, as it corresponds to approximately 4–6 lines of codes (LOC).

5.3.2. Step 2: Clone Analysis with Developer Involvement

With the large number of clones reported by the clone detector, developers should pay most attention to recurring structures of substantial size that form meaningful clone classes. ART templates of such structures are likely to be beneficial to developers. This section shows such types of clone classes with examples.

5.3.2.1. Types of Clones that can be handled using the ART

Based on the clone granularity, candidate clone-classes can be grouped into different categories as discussed below:

Similar Directories

Figure 5.12 gives an example of cloned directories—/jbd and /jbd2 found in the Linux kernel-3.10. In the Linux kernel, the Journaling Block Device (JBD)

provides a file-system independent interface for file system journaling. There are two directories, namely /jbd and /jbd2, implementing this functionality, with /jbd2 being an evolutionary branch of /jbd. /jbd2 compatibly extends /jbd with new features such as support for 64-bit computers, check-summing of journal transactions, and asynchronous transaction commit block write.

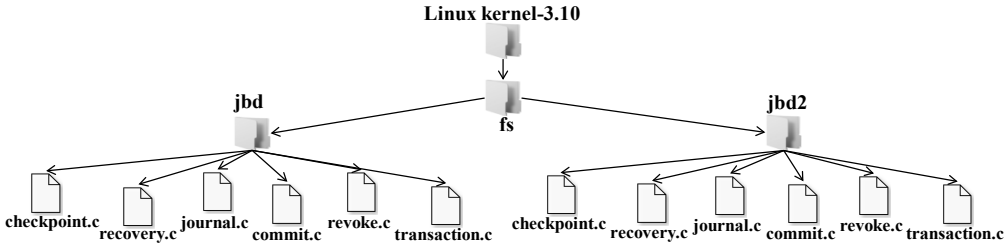


Figure 5.12. Cloned directories /jbd and /jbd2

Table 6 shows similarities and differences (in terms of LOC) among files in /jbd2 with respect to their counterparts in /jbd. The considerable similarity in functionality and code between the files corresponding by name in the two directories suggests that /jbd2 files were created by copying and modifying /jbd files.

Table 6. Comparison of /jbd2 with respect to /jbd

File Name	Total LOC in Corresponding jbd/jbd2 files	Identical LOC	LOC with Parametric Differences	Modified LOC	Inserted LOC	Deleted LOC
checkpoint.c	782/705	554	47	12	29	95
commit.c	1002/1192	523	93	35	364	218
journal.c	2122/2146	1266	287	29	690	229
recovery.c	594/862	420	52	12	234	0
revoke.c	740/769	544	94	3	25	0
transaction.c	2229/2348	1346	130	56	516	399

Figure 5.13 sketches code snippets highlighting the code similarity and differences between the two checkpoint.c files. The directories /jbd and /jbd2 exemplify the situations that can benefit from ART-template views of the program.

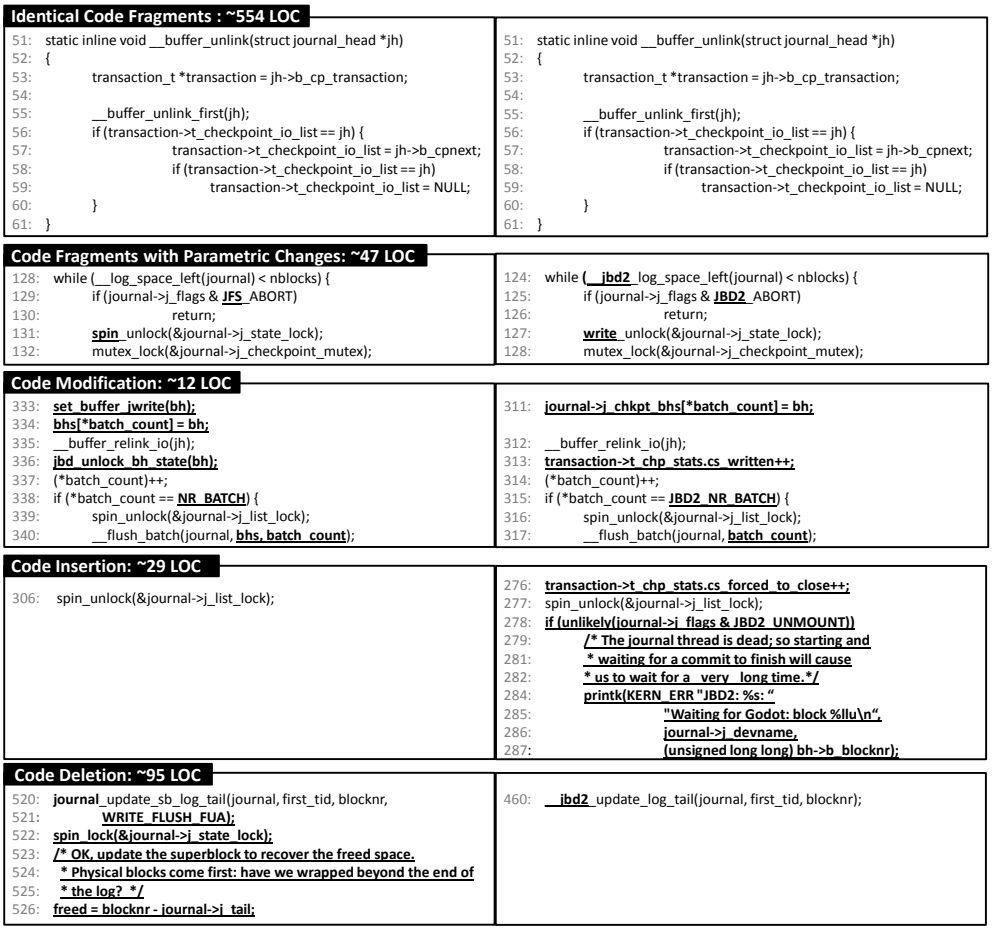


Figure 5.13. Code snippets of cloned file /jbd/checkpoint.c (left) and /jbd2/checkpoint.c (right)

In the Linux kernel and other software systems that we considered for case studies (Java Buffer Library, for example), we found many other cases following the pattern of /jbd and /jbd2. However, in some cases, a directory contains one or more files that do not have similar counterparts in the cloned directory. The reason we find such types of big clones in the Linux kernel—and, we believe, in many other evolving systems—is the limitation of underlying variability management techniques to tackle such duplicated program structures in a non-redundant way, due to functional similarities among different subsystems, extensions to the existing functionalities, adaptation of the existing subsystem code for the new one (incremental

development), and decentralized and voluntary basis development efforts [150-152].

Similar Files

From the large number of code clones reported by a clone detector, despite similar directories, developers should also pay most attention to other recurring structures of substantial size that form meaningful clone classes. One of such cases is similar files. Due to large size, similar files (i.e., file clones) are one of the clone-candidates whose ART templates are likely to be beneficial to the developers.

Many cases of similar files within the same directory, as well as across directories, occur in software systems.

A common reason for replicating a file in the same directory is to make a certain existing functionality available for another computer architecture, device, or tool. An example from the Linux kernel-3.10 is the drivers for different brands of touchscreen devices—in directory /drivers/input/touchscreen, 10 files share the same structure and much code. Similarly, in the Java Buffer Library, a group of seven source files—ByteBuffer.java, CharBuffer.java, IntBuffer.java, DoubleBuffer.java, FloatBuffer.java, LongBuffer.java, and ShortBuffer.java—have almost 90% of the cloned code (either exact or with parametric differences). It makes these files a good candidate for the ART-template representation.

Two directories having almost similar purposes (vide Figure 5.12) may contain similar files. Sometimes, the same or similar file may be required in two or more directories, even if the corresponding directories do not have enough

code similarity. For example, in the Linux kernel-3.10, functionality for handling extended user attributes is needed in directories /fs/ext2, /fs/ext3, and /fs/ext4, and therefore file “xattr_user.c” that defines this functionality appears in all three directories.

Collaborative Patterns

Collaborative patterns are useful candidates to consider, especially when ART template representation of them proves to be beneficial to developers. For example, in the Clone Analyzer-2.0, there are three collaborating methods: getJInternalFrame(), getJContentPane(), and getJScrollPane(). These three methods occur in each of following three files: “SecondaryNavigator.java”, “PrimaryNavigator.java”, and “UserMinerSettings.java”. These three files do not have enough code similarity, and hence cannot be considered as file clones of each others. But, these three collaborating methods can be represented as an ART template that can be shared across these three files.

Duplicated Code Fragments and Methods

At times, template views of duplicated code fragments can also be useful, particularly so if these code fragments are large enough (at least six LOC, for example), play some specific role (e.g., represent some meaningful function), and/or recur in many places in programs. For example, in the Linux kernel-3.10, the code fragments in Figure 5.14 implement a device-specific queue handling procedure for different wireless network adapters. An instance of this code fragment occurs once in each of the files “rt2400pci.c”, “rt2500pci.c”, “rc2800pci.c”, and “rt61pci.c”, and twice in each of the files “rt2500usb.c”, “rc2800usb.c”, and “rt73usb.c”.

rt73usb.c

```
static void rt73usb_start_queue(struct data_queue *queue)  {
    struct rt2x00_dev *rt2x00dev = queue->rt2x00dev;
    u32 reg;
    switch (queue->qid) {
    case QID_RX:
        rt2x00usb_register_read(rt2x00dev, TXRX_CSR0, &reg);
        rt2x00_set_field32(&reg, TXRX_CSR0_DISABLE_RX, 0);
        rt2x00usb_register_write(rt2x00dev, TXRX_CSR0, reg);
        break;
    case QID_BEACON:
        rt2x00usb_register_read(rt2x00dev, TXRX_CSR9, &reg);
        rt2x00_set_field32(&reg, TXRX_CSR9_TSF_TICKING, 1);
        rt2x00_set_field32(&reg, TXRX_CSR9_TBTT_ENABLE, 1);
        rt2x00_set_field32(&reg, TXRX_CSR9_BEACON_GEN, 1);
        rt2x00usb_register_write(rt2x00dev, TXRX_CSR9, reg);
        break;
    default:
        break;
    }
}
```

rc2800usb.c

```
static void rt2800usb_start_queue(struct data_queue *queue) {
    struct rt2x00_dev *rt2x00dev = queue->rt2x00dev;
    u32 reg;
    switch (queue->qid) {
    case QID_RX:
        rt2x00usb_register_read(rt2x00dev, MAC_SYS_CTRL, &reg);
        rt2x00_set_field32(&reg, MAC_SYS_CTRL_ENABLE_RX, 1);
        rt2x00usb_register_write(rt2x00dev, MAC_SYS_CTRL, reg);
        break;
    case QID_BEACON:
        rt2x00usb_register_read(rt2x00dev, BCN_TIME_CFG, &reg);
        rt2x00_set_field32(&reg, BCN_TIME_CFG_TSF_TICKING, 1);
        rt2x00_set_field32(&reg, BCN_TIME_CFG_TBTT_ENABLE, 1);
        rt2x00_set_field32(&reg, BCN_TIME_CFG_BEACON_GEN, 1);
        rt2x00usb_register_write(rt2x00dev, BCN_TIME_CFG, reg);
        break;
    default:
        break;
    }
}
```

Figure 5.14. Sample code fragments from rt73usb.c and rc2800usb.c from the Linux kernel-3.10 (differences highlighted)

5.3.3. Step 3: Tailoring ART Command Set (optional step)

The ART syntax is cpp based. However, users can easily change the ART syntax to suit their requirements. This would be helpful if any of the ART commands conflicts with the reserved words of the native language of the software system under consideration, or if the ART user feels uncomfortable with any of the ART command syntax. We have used the default ART implementation, which is a cpp compatible version, i.e., does not conflict syntactically with cpp directives. A brief description of the default ART command set is given in Table 5 and is explained in detail in Section 5.2.2.

5.3.4. Step 4: Constructing ART Templates

This step deals with representing each clone class, found after clone analysis, in the form of non-redundant ART-template views of the program. This subsection explains how to systematically use the ART to represent clones in the form of generic, adaptable, and reusable ART templates.

5.3.4.1. ART Template Construction Mechanism

In ART-template based view of the program, each clone class (i.e., similar directories, similar files, etc.) is represented using ART templates. These ART templates record the locations of variation points where different instances of the clone class differ.

Despite a large fraction of code common to all the clone instances (i.e., identical code fragments in the corresponding clone instances) of a clone class, as shown in Figure 5.13, there are mainly following three types of differences among corresponding clone instances:

1. Parametric differences (code with parametric changes)
2. Alternatives (code modifications), and
3. Extras (code insertions and deletions).

The first task during ART-template construction is to identify these similarities and differences among corresponding clone instances of the clone class. For example, with reference to Figure 5.15:

- Code fragments A, D, and F correspond to identical code fragments in all the three instances of the given clone class.
- Code fragments B1, B2, and B3 have parametric differences among them. Similarly, code fragments H1, H2, and H3 also have parametric differences among them.
- Code fragments E1, E2, and E3 correspond to alternative code in the three clone instances.
- Remaining code fragments are extras, i.e., code insertion or deletions.

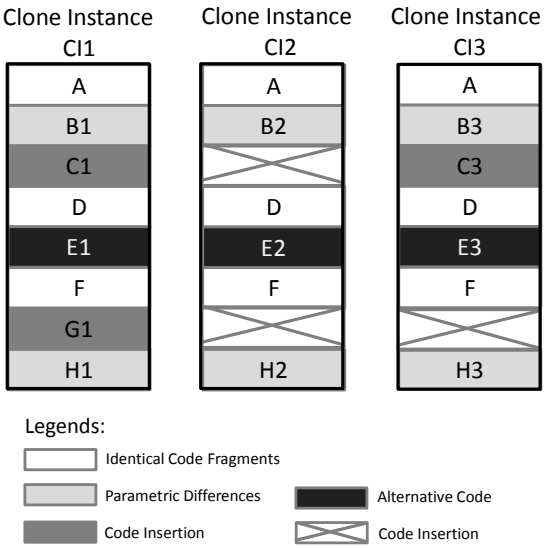


Figure 5.15. Illustrative example to show similarities and differences among clone instances of a clone class

Once the corresponding similarities and differences are identified, ART templates record exact locations of these variation points at which the clone instances differ. ART commands can be used systematically to mark these variation points as discussed below:

- *Handling Identical Code Fragments:* Identical code fragments can be used directly as-it-is in the corresponding ART templates. For example, with reference to Figure 5.15, identical code fragments A, D, and F can be used directly in the ART-template solution without any modification.
- *Handling Parametric Differences:* Parametric differences such as variations in user-defined identifiers, literals, layout, types, etc. can be systematically dealt with the ART. Such differences can be easily handled using ART multi-value variables. Such multi-value variables can be declared using #set command. Each value of a multi-value variable corresponds to the parametric variations in the corresponding

clone instances. For example, suppose B1 uses a parameter “b1”, while the same parameter is named as “b2” and “b3” in B2 and B3, respectively. These parametric differences can be unified by defining an ART variable say “var1”, and assigning these values to that variable as follows:

```
#set var1 = "b1", "b2", "b3"
```

The three parametric differences, i.e., “b1”, “b2”, and “b3”, can be referenced using “@” operator (as discussed in detail in Section 5.2.2).

- *Handling Alternatives:* ART command #select allows choosing one among alternatives. Each of the alternatives is represented by a #option clause under #select. For example, with reference to Figure 5.15, we can unify alternative code fragments E1, E2, and E3 using #select as:

```
#select <clone-instance-id>
    #option <clone-instance-1>
        E1      % alternative code from Clone Instance 1
    #endoption
    #option <clone-instance-2>
        E2      % alternative code from Clone Instance 2
    #endoption
    #option <clone-instance-3>
        E3      % alternative code from Clone Instance 3
    #endoption
#endselect
```

- *Handing Extras:* #insert and #break commands together handle additions and deletions of extra code. #break command marks the location in the template where the additional code needs to be inserted.

Such additional code fragments can be then injected at the marked point using #insert, #insert-before, and #insert-after commands. The insert-break mechanism is discussed in detail in Section 5.2.2.

It is to mention that the actual construction of ART templates is a manual process that can be performed systematically using the ART commands. Just like program design, ART template design requires expert judgment that cannot be easily replaced by automated decision making process. There is a choice of ART mechanisms such as parameterization, selection, or insertions of program structures at designated points in templates that can be used to tackle various redundancy situations. These ART template design choices have various desirable and undesirable outcomes just like a decision to use a certain design pattern during conventional program design may have positive and negative implications. However, the process of generation of code from the ART templates has been automated using the ART Processor. It is a challenge for future research to identify design heuristics that could allow us to automate some of the ART template design decisions.

In the rest of this subsection, we discuss how these ART commands can be used systematically during the ART-template construction for different types of clones using examples.

5.3.4.2. Constructing ART Templates for Similar Directories

We can represent each set of similar directories using an ART-template hierarchy. In the template-hierarchy, ART templates are linked via #adapt commands. The topmost template, called specification file (SPC), implements

the overall process of generating clone instances from the ART-template hierarchy.

With reference to the left side of Figure 5.16, assume that there are three directories “DirX”, “DirY”, and “DirZ”. It is mentioned in the Section 5.3.2.1 that similar directories may follow a regular similarity pattern as in Figure 5.12. Based on this, assume that “DirX” and “DirY” follow a regular similarity pattern. On the other hand, in some cases, a directory may contain one or more files that do not have similar counterparts in the cloned directory. Assume that “FileZ” in “DirZ” represents such cases. For a given clone class, ART template-solution follows a hierarchical structure as shown in the bottom part of Figure 5.16:

1. The topmost ART template at level 1, called SPC, handles parametric differences. Also, it is the topmost template that implements the overall process of generating clone instances from the ART-template solution.
2. ART templates at level 2 handle code differences such as alternative and extras among the similar files.
3. Lower-level templates at level 3 handle code similarities in the clone instances and serve as building blocks for the corresponding similar files. These ART templates are customized using ART commands to eliminate redundancies. Further, if required, as shown in Figure 5.16, these templates may be interlinked by #adapt commands to form a hierarchy (further details to follow when different examples are provided in the thesis).
4. Remaining files that do not have counterparts in the cloned directories can be used as-it-is in the template solution. It is to mention that it is

one of the enhancements that the ART offers as compared to the XVCL. The XVCL does not allow adapting non-XVCL files.

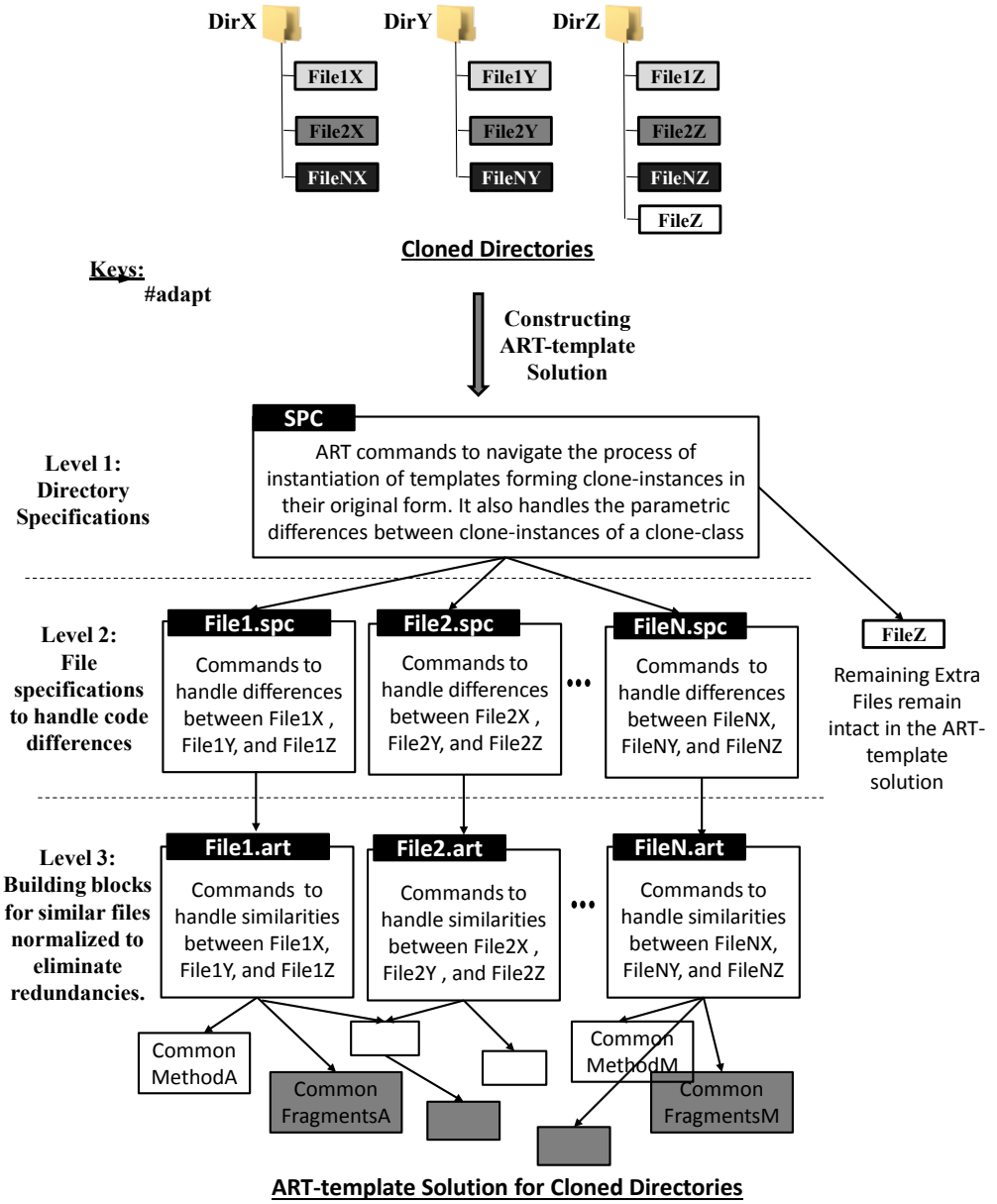


Figure 5.16. Constructing ART-template hierarchy

We use JBD file system of the Linux kernel-3.10 as an example to illustrate the template construction process. Figure 5.17 shows a sketch of the ART templates for the JBD file system of the Linux kernel-3.10. Each pair of clones in the two source files (e.g., checkpoint.c in /jbd and /jbd2) is represented by a template (e.g., checkpoint.art). The associated template checkpoint.spc

specifies the differences between the two source files as deltas from checkpoint.art. The top-most template jbdX.spc navigates the process of instantiating the templates to form the Linux source files in their original form.

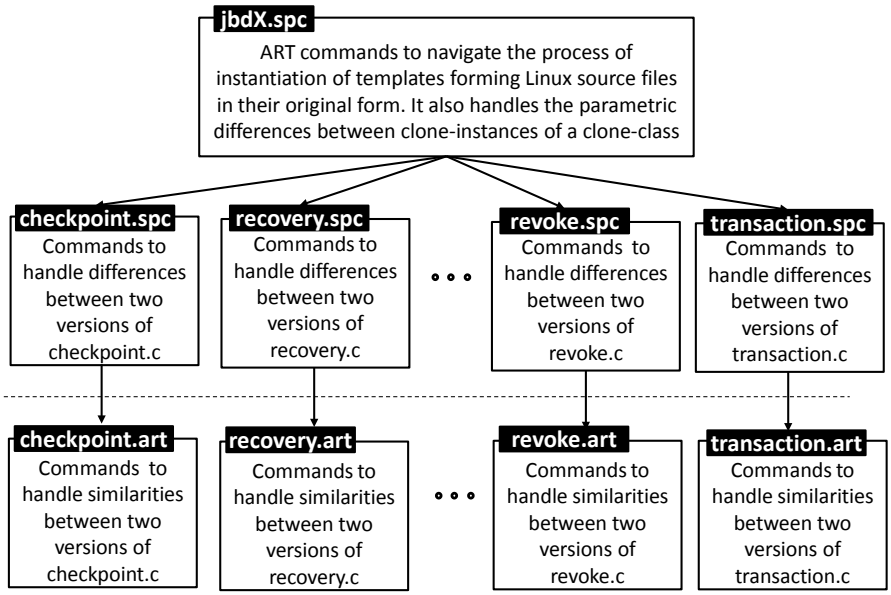


Figure 5.17. Constructing ART templates: JBD example

Figure 5.18 shows the expansion of some of the ART templates to highlight the solution. As shown in jbdX.spc, ART variables are declared using #set commands (lines 1–6). Variable "dirName" is assigned two values, "jbd" and "jbd2" (line 2) that control the #while loop (line 7). The loop executes twice, with the value of "dirName = jbd" in the first iteration, and the value of "dirName = jbd2" in the second iteration. The variable "fileName" is set to six values, each representing a file name (line 3).

The ART variable "action" helps represent lines:

```
spin_unlock(&journal->j_state_lock);       //in jbd/checkpoint.c
write_unlock(&journal->j_state_lock);      //in jbd2/checkpoint.c
```

in a single line in checkpoint.art (line 4):

```
?@action?_unlock(&journal->j_state_lock);
```

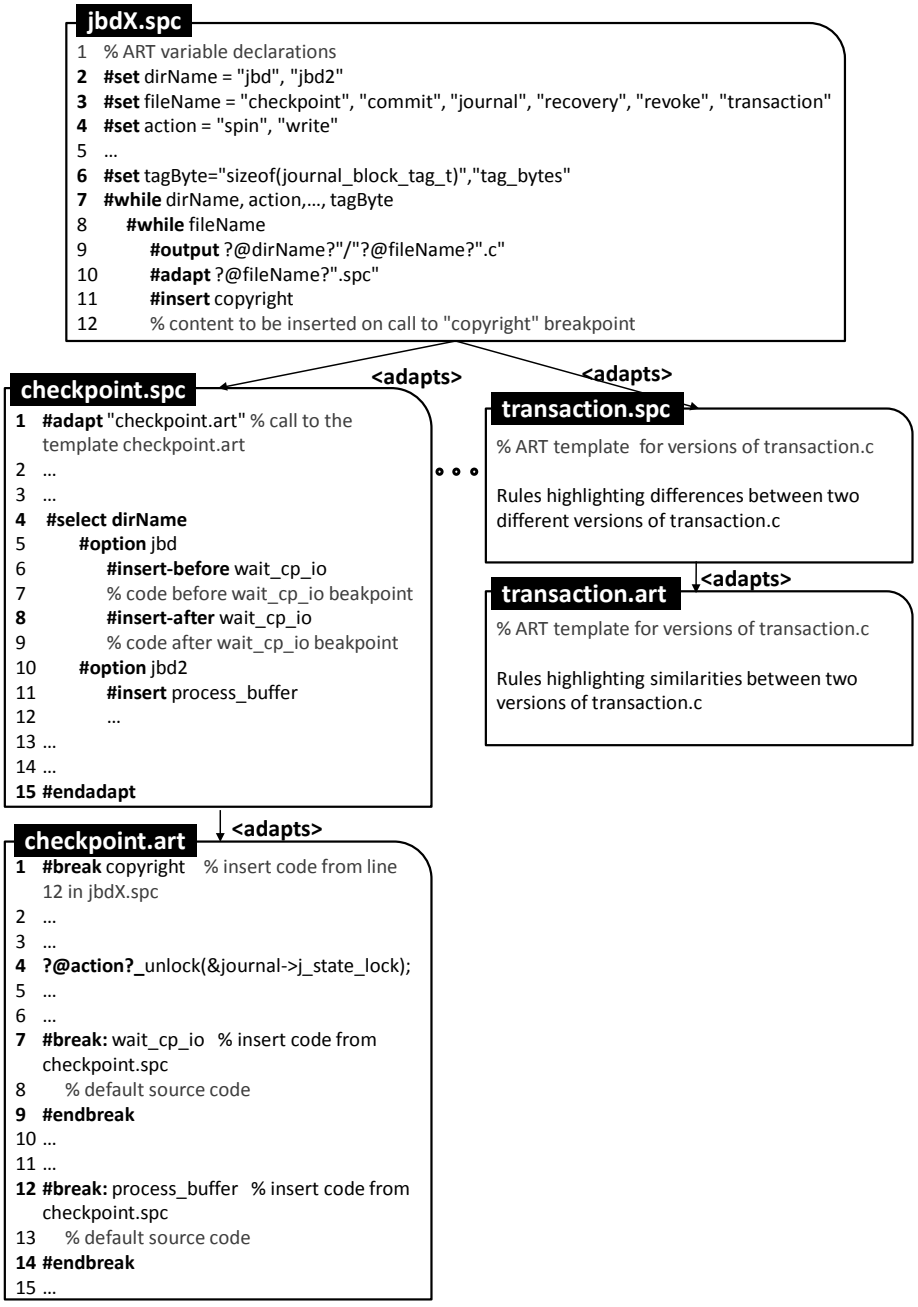


Figure 5.18. Code snippet of ART templates for the JBD example

The two values of "action" are defined by:

#set action = "spin", "write" // line 4 in jbdX.spc

The generation loop defined in line 7:

#while dirName, action,…, tagByte

is controlled by a list of variables. In this way, any parametric differences between the two checkpoint.c files are catered to. The command #output (line

9) instructs the ART Processor to create a directory and to place any further output into this directory. Expression "?@fileName?" is used to fetch the value of an ART variable fileName (line 9). The #adapt command in line 10 instructs the ART Processor to include the designated template to the output.

Variation points at which the two corresponding files (e.g., checkpoint.c) in /jbd and in /jbd2 directories differ are marked using ART commands—references to ART variables, #select, #break, and other commands. ART variables control selection of the code in case of alternative differences. This is illustrated as "#select dirName" in the template checkpoint.spc (line 4). #option (lines 5 and 10 in checkpoint.spc) controls the variable values.

File checkpoint.c in one directory contains some extra lines compared with the checkpoint.c in another directory. These extra lines are specified using #insert commands in various "#select dirName" options. "#insert process_buffer" (line 11 in checkpoint.spc) propagates the code to "#break: process_buffer" in the checkpoint.art (line 12). #insert-before and #insert-after (lines 6–9 in checkpoint.spc) add their code before or after the code contained in the matching #break (line 7 in checkpoint.art). While #select instruments a template with known variations, #break allows for extensions to a template in unexpected ways in the specific context of adaptation, without affecting others. These provisions for unexpected evolutionary changes give ART templates flexibility and stability.

Other Cases of Clones at the Directory Level

Other cases of cloned directories may not follow such a regular similarity pattern as in /jbd and /jbd2. For example, in the Linux kernel-3.10, in the

directories /drivers/infiniband/hw/qib and /drivers/infiniband/hw/ipath, in addition to similar files, /drivers/infiniband/hw/qib contains some extra files that do not have a counterpart in /drivers/infiniband/hw/ipath. Still, there is enough similarity in the concept and the code between /drivers/infiniband/hw/ipath and /drivers/infiniband/hw/qib to build ART templates for these two directories. The scheme used for building ART templates for /jbd and /jbd2 is also applicable in these situations, as ART templates manage pairs of similar files only and the remaining other files remain intact in the directories.

5.3.4.3. Constructing ART Templates for Similar Files

In this case, we deal with similar files found in the same directory and similar files in different directories, bearing in mind that directories as a whole are not considered good candidates for representing them as templates. For each such situation, we can create ART templates for similar files if we think that exposition of similarities and differences among these files can aid developers in reuse, program understanding, maintenance, and evolution of the software system.

The ART-template solution for similar files follows a similar scheme to that shown in Figure 5.17 and Figure 5.18. The topmost template, called SPC, specifies the parametric differences between similar files. Similar to previous cases, SPC also contains the code to initialize the ART-template solution to generate the similar files into their original form. Lower level templates handle differences and similarities between the corresponding similar files.

5.3.4.4. Constructing ART Templates for Collaborative Patterns

This case deals with constructing ART-template solution for recurring configuration of collaborating methods, where corresponding methods in the instances of the configuration are clones of each other and the files containing the configuration as a whole are not good candidates for template representation. Each of such situations can be handled by creating an ART template (as compared to the template hierarchy for similar files and similar directories) for the configuration and adapting it in the required files. However, similar to other cases, parametric differences are handled using multi-valued ART variables. Other differences are handled using loops, selection, and insert-break mechanisms.

Figure 5.19 shows an example of ART-template solution for one of the collaborative patterns described in the Section 5.3.2.1. Variable “className” is assigned three values each representing the filename containing the instance of three collaborating methods (line 1). The #while loop in line 2 controls the generation of three instances of the given collaborative pattern. Since the corresponding methods in the instances of the pattern are clones of each other, the additions and deletions of extra code in the corresponding methods are limited to few lines of code only. Hence, they can be easily catered to using #select command. For example, method getJContentPane() in UserMinerSettings.java contains a few extra lines than its two other instances in SecondaryNavigator.java and PrimaryNavigator.java. This can be easily handled using #select command (line 5). It improves the readability of the constructed template too. As usual, alternative code among cloned methods are handled using #select commands (line 13). We further converted a few of

collaborative patterns found in JHotDraw 7 and Clone Analyzer into ART templates. In these cases, we eliminated 20–40% of the redundant code.

```
#set className = "SecondaryNavigator", "PrimaryNavigator", "UserMinerSettings"

#while className
  private JPanel getJContentPane() {
    % common code to all the instances here
    #select className
      #option UserMinerSettings
        % extra lines of code specific to UserMinerSettings. getJContentPane() method here
      #endoption
  #endselect
  }

  private JInternalFrame getJInternalFrame() {
    %common code to all the instances here
    #select className
      #option SecondaryNavigator
        % alternative code specific to SecondaryNavigator.getJInternalFrame() method here
      #endoption
      #option PrimaryNavigator
        % alternative code specific to PrimaryNavigator.getJInternalFrame() method here
      #endoption
      #option UserMinerSettings
        % alternative code specific to UserMinerSettings.getJInternalFrame() method here
      #endoption
    #endselect
  }

  private JScrollPane getJScrollPane() {
     %> common code to all the instance is copied exactly.
        Alternatives and differences are handled using ART commands  <%
  }
#endwhile
```

Figure 5.19. Code snippet of an ART template for a collaborative pattern

5.3.4.5. Constructing ART Templates for Duplicated Code Fragments and Methods

Situations where the template views of duplicated code fragments and methods can also be useful are handled by creating an ART template and adapting it at the required variation points. The solution follows a similar scheme to that shown in Figure 5.19.

5.3.5. ART Templates to Original Clone-Instances

ART templates can be converted back to the original native code automatically by using the ART Processor. The ART Processor traverses the template hierarchy and generates the required clone instances as output.

As an example, for the templates as shown in Figure 5.18, the ART Processor generates the original native code traversing the template hierarchy and emitting the code for the six files in the /jbd and /jbd2 directories from their respective templates. Template views expose the fact that the two directories and corresponding files in them are similar to each other, and also explicate every detail of the similarities and differences among them. This information is implicit in the original native code. Explicating it using the ART can be useful in the future evolution of the software.

5.4. Conclusions

In this chapter, we presented a meta-programming technique and tool, the ART, that can be used for managing big clones. The ART represents clones in the form of non-redundant, adaptable, and reusable templates, called ART templates. A prominent feature of the ART is that due to user-defined syntax, it blends in a non-conflicting way with the underlying programming language. In this chapter, we first described each of the ART commands in detail. Then, a systematic mechanism for constructing ART templates has been elaborated. The ART has been properly implemented and is available in a ready-to-use form. In the next chapter, we discuss the experimental results evaluating the effectiveness, usefulness, and benefits of managing code clones using the ART. We also discuss related works in detail in the next chapter.

Chapter 6.

EVALUATION AND BENEFITS OF MANAGING CLONES USING THE ART

In the previous chapter, we presented a meta-programming technique and tool, the ART, that can manage families of redundant software systems by providing a common base of non-redundant, adaptable, and reusable meta-components—called ART templates. This chapter quantitatively and qualitatively evaluates the strengths and weaknesses of the ART (Section 6.1). Having discussed the related works in Section 6.2, Section 6.3 concludes the chapter.

6.1. Evaluation

We have created ART-template solutions for the Java Buffer library, Notepad system, and a part of the Linux kernel to quantitatively and qualitatively access the strengths, weaknesses, and trade-offs involved in the application of the ART. It is to mention that the predecessor of the ART, i.e., the XVCL, has already been used in the case studies for Java Buffer library [138] and Notepad system [153]. So, performing case studies on Java Buffer Library and Notepad system help us to compare the results from two systems.

The section is organized as: Initially the case studies using the ART for Java Buffer library (Section 6.1.1), Notepad system (Section 6.1.2), and a part of the Linux kernel (Section 6.1.3) are presented. Next, the learnings from these case studies have been used to perform the quantitative evaluation (Section 6.1.4) and qualitative evaluation (Section 6.1.5) of the ART. The section ends with a discussion on trade-offs of the technique (Section 6.1.6).

6.1.1. Java Buffer Library Example

The Java Buffer library has been a part of the java.nio.* package in JDK since version 1.4.1. It implements containers for reading and writing data in a linear sequence. It consists of buffer classes differing from each others with respect to buffer element type, memory allocation scheme, byte ordering, and access mode. Figure 6.1 shows a feature diagram [154] with five feature dimensions, with specific variant features listed below a corresponding feature dimension box. Class names in the Buffer library reflect combination of these specific features implemented into a given class. For example, DirectIntBufferR is a Read-Only buffer of integers, implemented using direct memory scheme. Classes whose names do not include ‘R’, are ‘W’—Writable by default. The Buffer library contains classes whose names are derived from a template: [MS][T]Buffer[AM][BO], where MS—Memory Allocation Scheme: Heap or Direct; T—Element Type: Int, Double, Float, Long, Short, Byte, or Char; AM—Access Mode: W (Writable, default) or R (Read-Only); BO—Byte Ordering: S (non-native) or U (native), B (Big-Endian) or L (Little-Endian). For simplicity, we can ignore VB—View Buffer, which is, in fact, yet another feature that allows us to interpret byte buffer as Char, Int, Double, Float, Long, or Short. Each legal combination of variant features yields a unique buffer

class and it ends up having 74 buffer classes with 68% similarity between them [138].

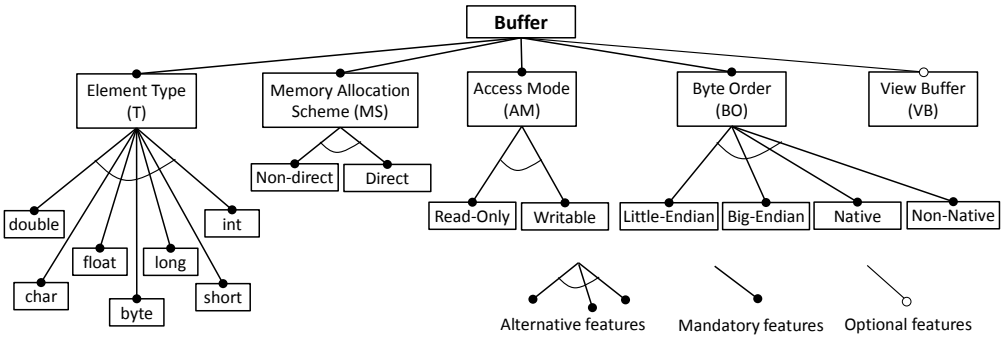


Figure 6.1. Features in the Java Buffer library

Representing code clones using ART templates makes the code easier to understand and debug. Based on the clone analysis of the library, 71 of the buffer classes can be grouped into seven similarity groups, while the remaining three buffer classes (Buffer.java, MappedByteBuffer.java, and StringCharBuffer.java) remain intact [138]:

1. [T]Buffer: seven classes that differ in buffer element type, T: Byte, Char, Int, Double, Float, Long, Short
2. Heap[T]Buffer: seven classes, with memory allocation scheme ‘Heap’, that differ in buffer element type, T
3. Heap[T]BufferR: seven ‘Read-Only’ and ‘Heap’ classes that differ in buffer element type, T
4. Direct[T]Buffer[S|U]: 13 ‘Direct’ classes for combinations of buffer element type, T, with byte orderings: S—non-native or U—native byte ordering (it is to mention that byte ordering is not relevant to buffer element type ‘Byte’)

5. Direct[T]BufferR[S|U]: 13 ‘Read-Only’ and ‘Direct’ classes for combinations of parameters T, S and U (byte ordering is not relevant to buffer element type ‘Byte’)
6. ByteBufferAs[T]Buffer[B|L]: 12 ‘ByteBufferAs’ classes for combinations of buffer element type, T, with byte orderings: B—Big-Endian or L—Little-Endian. T here denotes all seven buffer element types except ‘Byte’ (i.e., equivalent to VB)
7. ByteBufferAs[T]BufferR[B|L]: 12 ‘Read-Only’ ‘ByteBufferAs’ classes for combinations of parameters T (except ‘Byte’), B and L.

The ART-template solution of the Java Buffer library consists of template representations for each of these seven similarity groups (as shown by T1 to T7 in Figure 6.2) bonded together with a specification file SPC. Each of the similarity groups (e.g., T4) is represented by a template hierarchy, in which an ART template is either unique to one class, or is common to some/all of the buffer classes.

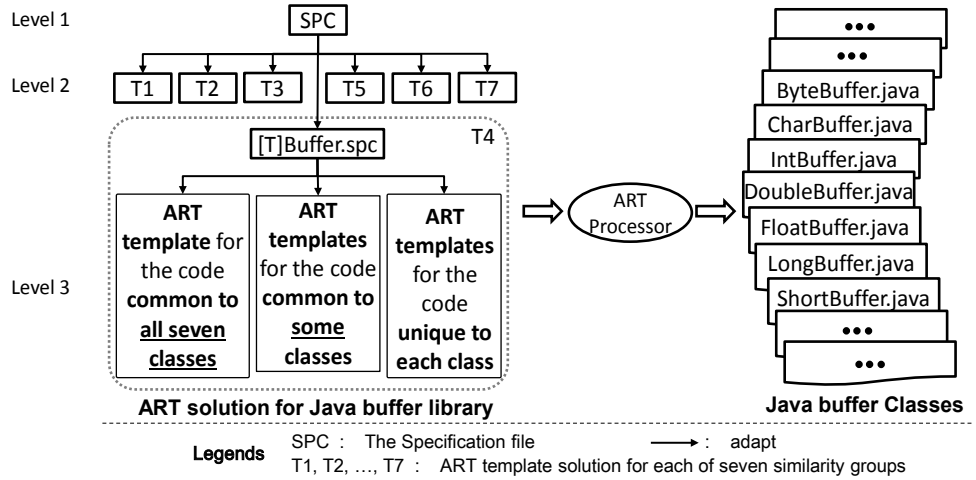


Figure 6.2. ART-template solution for the Java buffer library

Figure 6.3 shows the details of a fragment of the ART-template solution shown on the left side of Figure 6.2. As explained in Section 5.3.4.1 in Chapter 5,

level 3 ART templates play role of the templates defining common parts for all the classes in the respective similarity groups. For example, seven classes in the group [T]Buffer are derived using [T]Buffer.art. ART template [T]Buffer.spc contains specifications instructing the ART Processor on how to adapt [T]Buffer.art and other ART templates at levels below it to derive buffer classes in the [T]Buffer group. We have analogical solutions in parts of the buffer ART-template solution for other six groups of similar classes.

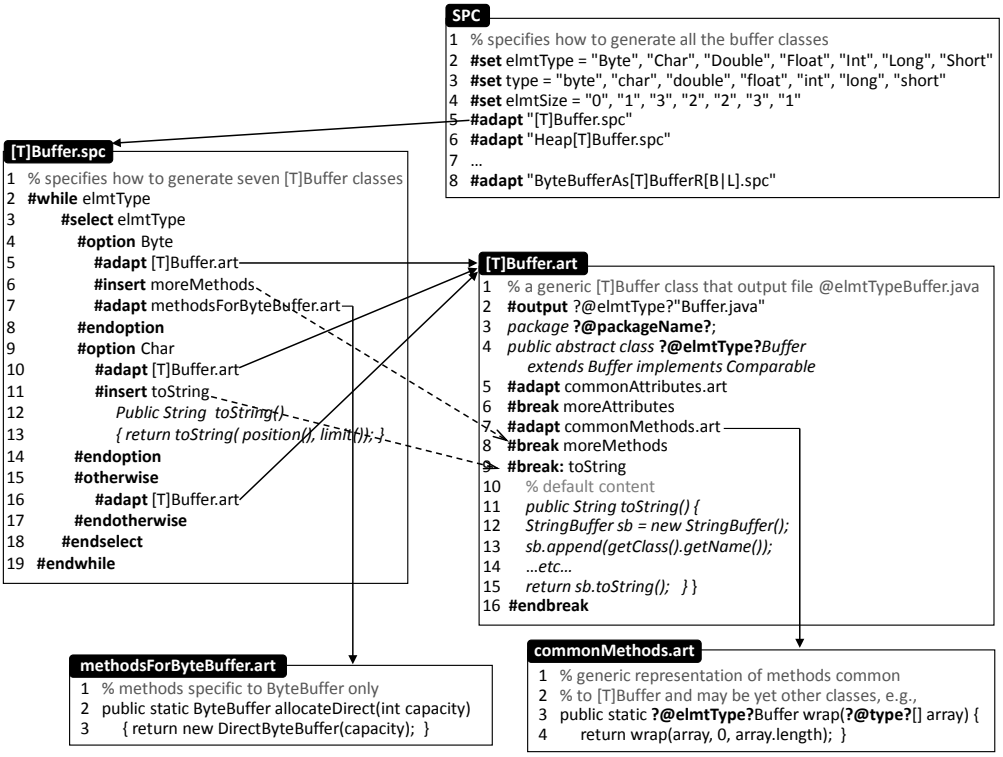


Figure 6.3. ART-template solution for seven [T]Buffer classes (partial)

#set command in line 2 of the SPC assigns values listed on the right side to a variable named elmtType. Expression ?@elmtType? refers to one of such values (line 4 in [T]Buffer.art, for example), which is replaced by the variable's value during processing. Having set values for the ART variables, the SPC initiates generation of classes in each of the seven groups of similar classes via suitable #adapt commands (lines 5–8).

The #while loop in [T]Buffer.spc (lines 2–19) is controlled by a multi-value variable, namely elmtType. In each iteration, the #select command uses the current value of elmtType to choose a proper #option for processing.

#output command in [T]Buffer.art (line 2) defines the name of a file where ART Processor will emit the code for a given class. ART template [T]Buffer.art further defines common elements found in all seven classes in the group. Five of those classes, namely DoubleBuffer, IntBuffer, FloatBuffer, ShortBuffer, and LongBuffer, differ only in type parameters (as in the sample method wrap() shown in ART template commonMethods.art). These differences are unified by ART variables, and no further customizations are required to generate these five classes from [T]Buffer.art. These five classes are catered for in #otherwise clause under #select (lines 15–17 in [T]Buffer.spc). However, classes ByteBuffer and CharBuffer have some extra methods and/or attribute declarations. In addition, method toString() has different implementation in CharBuffer than in the remaining six classes. Customizations specific to classes ByteBuffer and CharBuffer are listed in the #adapt commands, under #option Byte and #option Char, respectively.

The ART-template representation of the Buffer library explicates every detail of the similarities and differences among buffer classes. This information is implicit in the original Buffer library. Knowing the similarities and differences among the buffer classes helps the programmer to easily comprehend and understand the code. The original Buffer library consists of 16,299 LOC (including java code and comments), which are reduced to just 3,771 LOC (including java code, comments, and ART commands) in the non-redundant ART-template solution.

6.1.2. Notepad Example

Besides achieving non-redundancy in the software systems, this case study (taken from [153]) also exemplifies the capability of the ART in managing multiple versions of a software system from a common code-base. The Notepad example discussed is for illustration purposes, but the technique can be applied to any large software system that is a member of the product line. A Notepad is a typical text editor with drop-menus, a toolbar, and an editing panel. Our objective is to use the ART to develop a generic solution so that:

- It can cater to any changes arising during software maintenance and evolution, such as the addition of more menus, menu items, toolbar buttons, or functionality.
- It can be used in developing other similar systems (i.e., members of the Notepad product line).

Custom requirements of different customers/users may lead to multiple versions of the Notepad which differ in features such as the title of the Notepad, color, and appearance. In addition, variation in platforms or hardware may lead to multiple versions of the Notepad. The ART provides a general solution (as shown in Figure 6.4) that can be easily customized as per the version requirements. It consists of a template hierarchy, in which upper-level templates adapt the lower-level templates. The topmost template, SPC, contains the specifications for various versions of the Notepad system. It instructs the ART Processor on how to customize the remaining templates to generate the code for a specific version of the Notepad system. Figure 6.5 shows expansion of some of the ART templates highlighted in the solution.

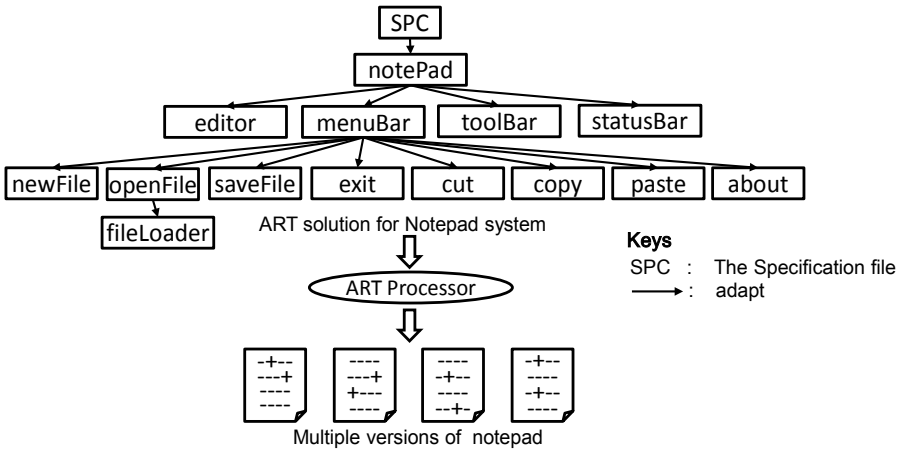


Figure 6.4. ART-template solution for the Notepad example

As shown in Figure 6.5, SPC defines the requirements for a custom notepad using different #set commands (lines 1–15). Customized title and background color for the notepad are defined in lines 4 and 5 respectively. Multi-value variables in lines 8–15 define customized menubar and its corresponding items for the required notepad. These customizations are then used by other lower-level templates in the hierarchy. These lower-level templates can be adapted using a #adapt command (line 16).

ART template notePad.art contains native code common to all the versions of Notepad as well as ART commands that mark the variation points among these versions. Each component of the Notepad (i.e., toolbar, editor, menubar, or statusbar) is designed as a separate template that can be reused and maintained as per requirements. These templates are adapted into notePad.art using #adapt commands (lines 12–18 in notePad.art).

The given solution further expands the templates for menubar items. Each item in the menubar has a name, an icon, and an associated action. The code for creating one menu-item is very similar to the code for creating other menu-items (except with a few parametric differences and possibly a little

addition/deletion of methods). Therefore, menuBar.art contains a generic solution for creating all kinds of menu-items. A specific menu-item can be generated using menuBar.art by adapting corresponding templates (line 30 in menuBart.art). For example, ART template newFile.art can be adapted for generating a menu-item for creating a new empty file.

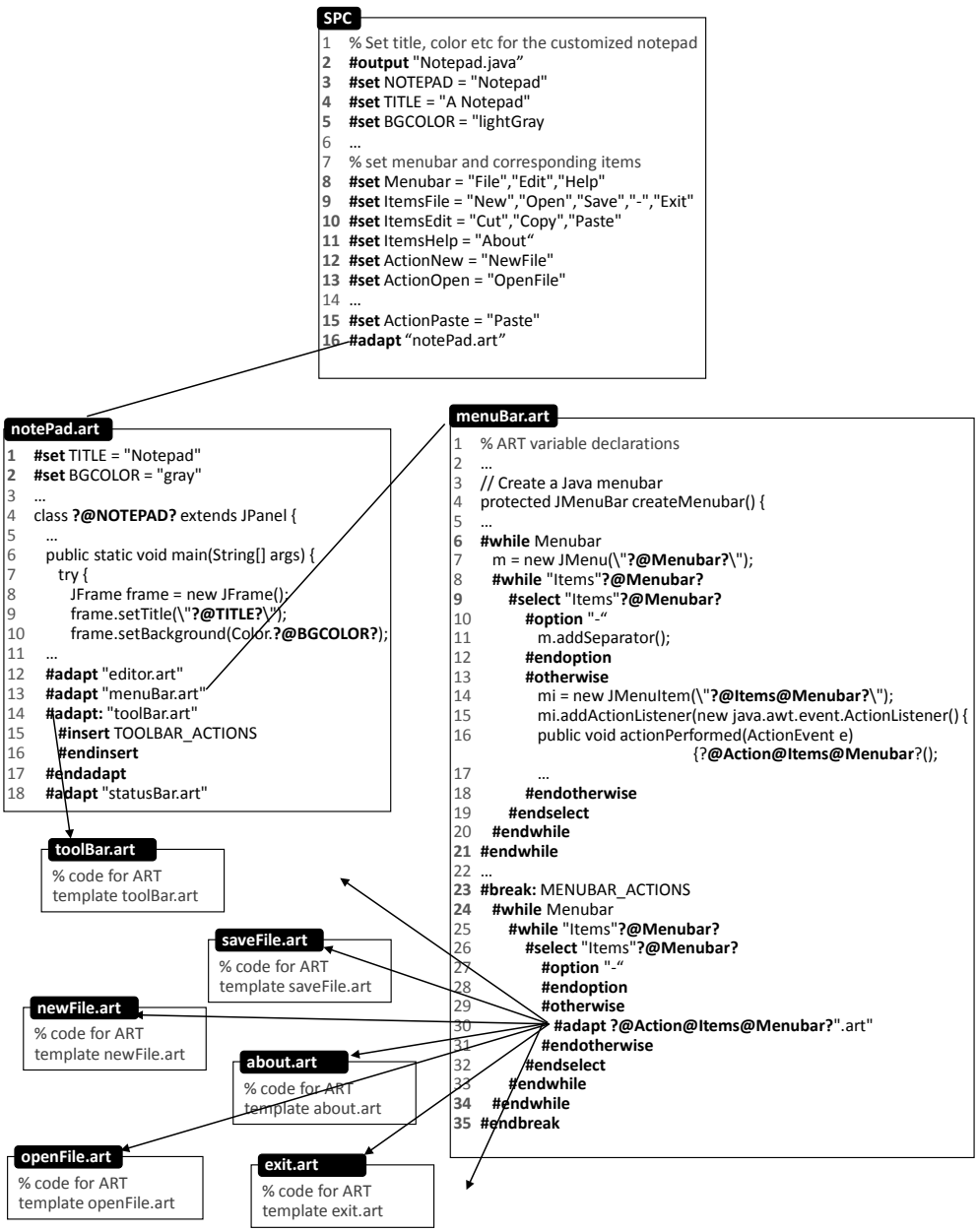


Figure 6.5. Code snippet for ART-template solution for the Notepad example

6.1.3. Linux Kernel Example

Linux kernel example illustrates the ART's ability to manage big clones, while a range of other techniques (e.g., cpp and Kconfig in Linux project) deal with other aspects of the overall variability management problem. Such seamless integration is necessary to allow developers to painlessly inject ART templates into projects in mature stages of evolution when big clones start emerging. ART syntax is user-defined to make such injection easy, without affecting already existing software solutions and people who work with them. In Linux kernel example, the ART can be viewed as an extension of cpp—ART commands syntactically resemble cpp directives and can be incrementally learned as extensions that enhance reuse capabilities of cpp. Figure 6.6 shows how the ART can be used in integration with cpp in the Linux kernel.

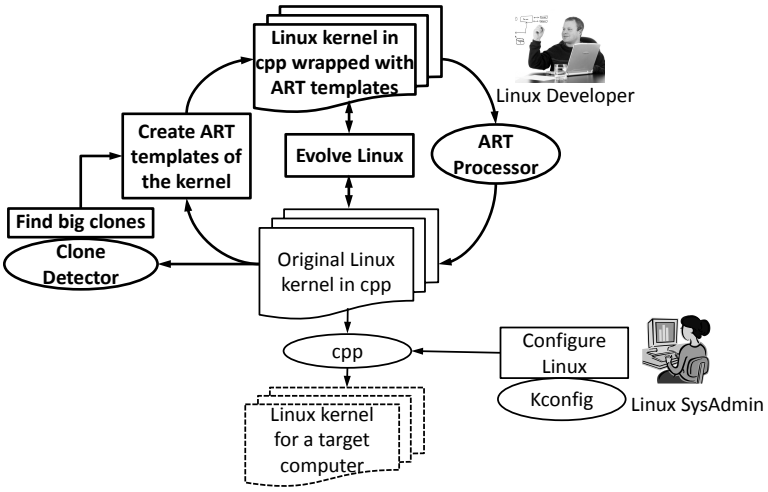


Figure 6.6. Working of the ART in integration with cpp for Linux kernel

As shown in Figure 6.6, there are two main user roles: the Linux Developer and the Linux SysAdmin. A Linux Developer is a member of the open-source community who contributes enhancements to the Linux kernel such as including new devices. The Linux SysAdmin adapts the kernel for her

computer using tools such as Kconfig. The Linux Developer can build ART templates on top of the Linux code managed by cpp. The ART templates do not affect the work of the Linux SysAdmin. Big clones are identified in the Linux kernel by a suitable clone detector. The Linux Developer then creates ART templates, and from that point onwards, big clones are maintained via the ART-template representation of the kernel.

ART templates can be converted back to the original Linux code using the ART Processor. The ART Processor instantiates the ART templates in the same way the C Preprocessor expands cpp directives. For example, for a template representing a group of similar files, the ART Processor generates code for these files based on specifications of deltas—differences between the template and each of these files. The generated Linux code is in the original form, and can be processed normally by Kconfig, cpp or the make tool. Figure 6.6 shows how the template view of the Linux kernel and the original Linux kernel can be used together in two independent cycles of maintaining and using the kernel.

Cloning in the Linux kernel has been extensively studied in the literature [150-152]. Our objective is not to have systematic clone analysis in the Linux kernel. Instead, we focused our effort on finding representative examples of various types of large-granular repetitions in the Linux kernel to illustrate the usage of our technique. We analyzed some parts of the /fs and /driver subsystems to find representative examples. The JBD file system, cloned files having code for drivers for different brands of touchscreen devices, etc. highlighted in Section 5.3 in Chapter 5 are few of such examples. We have already illustrated the ART-template solution for JBD file system in Section

5.3.4.2. We created ART-template solutions for other examples also. However, the construction follows the same mechanism as illustrated by various examples explained in the thesis. The original Linux code consists of 19,627 LOC (including C code, cpp directives, and comments), which are reduced to 12,453 LOC (including C code, cpp directives, comments, and ART commands) in the ART-template solution.

It is to mention that ART templates are not created for quick gains during development, but for long-term gains during software evolution and reuse. ART aims to benefit long-lived systems that undergo extensive evolutionary changes, or need to be tailored to the needs of multiple customers.

6.1.4. Quantitative Evaluation

For quantitative assessment, we compared the number of conceptual elements in the original source code, XVCL templates, and in the ART templates. We consider the following conceptual elements:

- *For the original source code:* LOC (native code, comments, without blanks), number of source files and directories, and McCabe's cyclomatic complexity.
- *For XVCL and ART templates:* LOC (native code, XVCL/ART commands, comments, without blanks), number of XVCL/ART templates, and any source files that are defined outside of templates, and McCabe's cyclomatic complexity.

Table 7 shows the quantitative analysis results.

Table 7. Quantitative analysis

	Java Buffer Library Example			Notepad Example		Linux Code Sample	
Conceptual Element	**Original Code**	**XVCL solution**	**ART solution**	**XVCL solution**	**ART solution**	**Original Code**	**ART solution**
LOC	16299	4149	3771	674	450	19,627	12,453
No. of Source Files	74	N/A	3	N/A	0	28	0
No. of XVCL/ART Templates	N/A	54	54	15	15	N/A	20
No. of Directories	1	8	8	1	1	7	6
McCabe V(G)	1114	329	289	12	12	1725	1156

The predecessor of the ART, i.e., the XVCL, has already been applied in many case studies including industrial projects [21, 22, 118, 129, 138, 140, 153, 155]. In these industrial projects, productivity impact of applying the XVCL was measured and evaluated. There are sufficient evidences from these projects that the overhead incurred by the application of the XVCL is smaller than benefits incurred by the XVCL. We have built the ART that further improves the capability of the XVCL, thus to be more impactful (the ART's improvement to the XVCL are highlighted in 'Related Works' Section 6.2.2). Further, apart from reducing the physical size and conceptual complexity, template views of the program emphasize important relationships among program elements that matter to programmers trying to understand and modify the code. Instead of dealing with each directory or file separately, programmers can comprehend them in groups, and see the commonalities and differences among members of each group. This is helpful in debugging and enhancing the code, as it reduces ripple effects and the risk of update anomalies. In this way, if one wants to change a file, it is easy to check whether the changes also affect the other files. For example, as illustrated in Figure 6.2 and Figure 6.3, similarities and differences are explicitly visible among the Java Buffer classes. Such relations are generally hidden in

conventional programs. Making them visible and easily tractable improves program maintenance. It also makes the impact of changes easy to comprehend.

6.1.5. Qualitative Evaluation

This subsection qualitatively accesses the strengths, weaknesses, and trade-offs involved in the application of the ART.

6.1.5.1. Aid in Program Understanding and Maintenance

Non-redundancy: ART templates eliminate duplicated code from the software systems. For example, in the detected clone classes, we eliminated 30–70% of the duplicated code. As both code and comments are important components for software maintenance and program understanding, the advantage of using the ART is that it is possible to manage both the cloned code and the comments with it. The ART allows a clean separation of various sources of changes that affect the program during evolution. ART templates reduce the number of points at which affected changes must be made. Changes made to one template consistently propagate to all contexts in which that template is adapted. Even if the changes are not uniform, adaptations can be made at specific variation points using ART commands without directly modifying the code fragments. The ART-template hierarchy explicitly reflects the impact of changes on the program structure. We can easily trace how different features affect the code.

Enhancing program understanding and conceptual integrity: According to Brooks [156], program understanding and conceptual integrity are the most important considerations in system design. Big clones often embody domain-

specific abstractions or design concepts. By formally capturing these abstractions and concepts, ART templates aid in program understanding and enhance the conceptual integrity of the design.

Creating templates can be considered as refactoring at the meta-level: In some cases, developers seek to improve certain program qualities but due to some unavoidable reasons cannot achieve this at the code level. In such cases, we can achieve this at the level of ART templates instead. We benefit from non-redundancy at the level of ART templates, while still keeping repetitions in programs (as it is often desirable or unavoidable [18, 140]).

Formally representing multiple design views: Program modules often belong to many logical groups that matter to developers at different times. Each logical partitioning reflects a certain aspect of the program design that matters at a given time in the development in a given context. For example, for a given business function in business software, the modules for user interface, business logic, and database are usually implemented in different system partitions. Logically, these modules belong to each other, and sometimes we must know which modules implement a given business function completely. However, only one logical partitioning can be formally represented in a program's physical structure. The ART provides a means to overlay programs with a web of meta-structures formally defining these logical partitions linked to the code, and without conflicts with the code.

6.1.5.2. Reusing Templates within a Version of the Software

In a large system such as the Linux kernel, there are many subsystems and modules in which similarities are found. Similar directories, files, or methods

may also be found across subsystems or modules. Each similarity group is managed by ART templates, as shown in the previous chapter. Therefore, non-redundant program views of similarities consist of many template hierarchies, one for each similarity group that is found to be worth exposing using the ART. As shown in Figure 6.7, the ART allows reusing of lower-level templates among the templates representing different clone classes that further simplify the overall non-redundant representation of the Linux kernel. Knowing the repetitions, their locations, and the exact nature of similarities and differences among replicated program structures is generally useful in understanding program design.

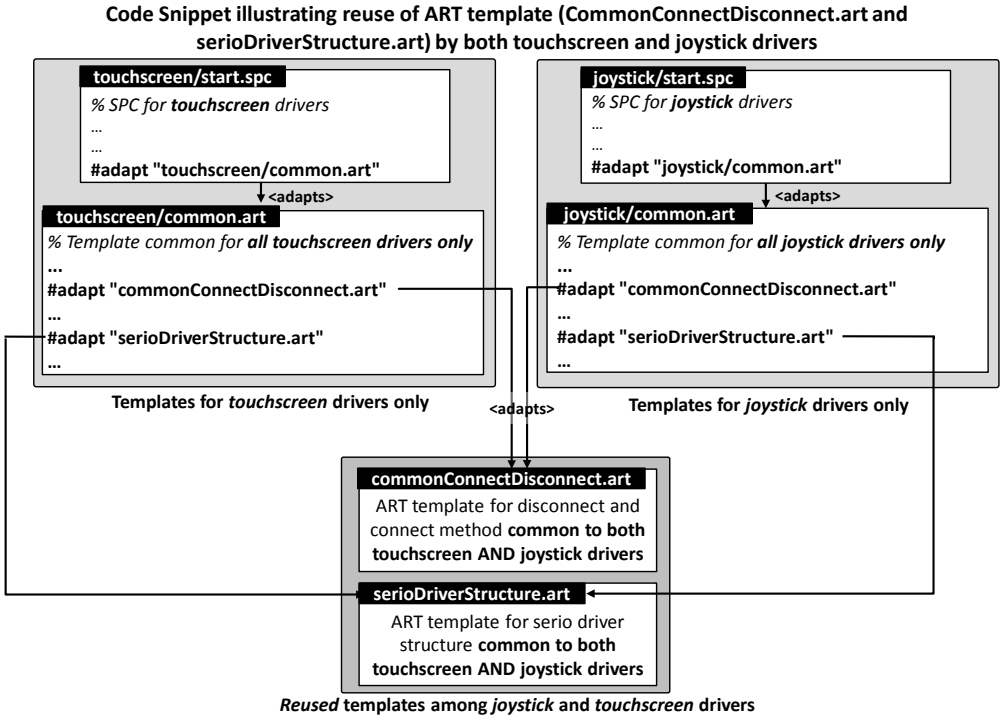


Figure 6.7. Template reuse: reusing ART templates

The example in Figure 6.7 shows how ART templates reveal implicit couplings among bigger structures that contain repetitions. The same functionality defined in the templates commonConnectDisconnect.art and serioDriverStructure.art is needed in /touchscreen/common.art and

/joystick/common.art. Templates for these two directories explicitly show the fact that this functionality is needed in both /touchscreen and /joystick drivers. If such implicit dependency among program modules is not documented, it may be overlooked during program evolution that may lead to errors.

6.1.5.3. Reusing Templates across Versions of the Software

Template reuse interconnects ART-template solutions developed for different clone classes from the bottom, as shown in Figure 6.7. It is also useful to interconnect partial ART-template solutions from the top, by introducing higher-level umbrella templates that trigger ART processing of some or all templates in the solutions. Umbrella templates help developers manage multiple versions of the software from a common base. Using umbrella templates, as shown in Figure 6.8, we represented the commonalities between two versions, together with the version-specific code in different templates.

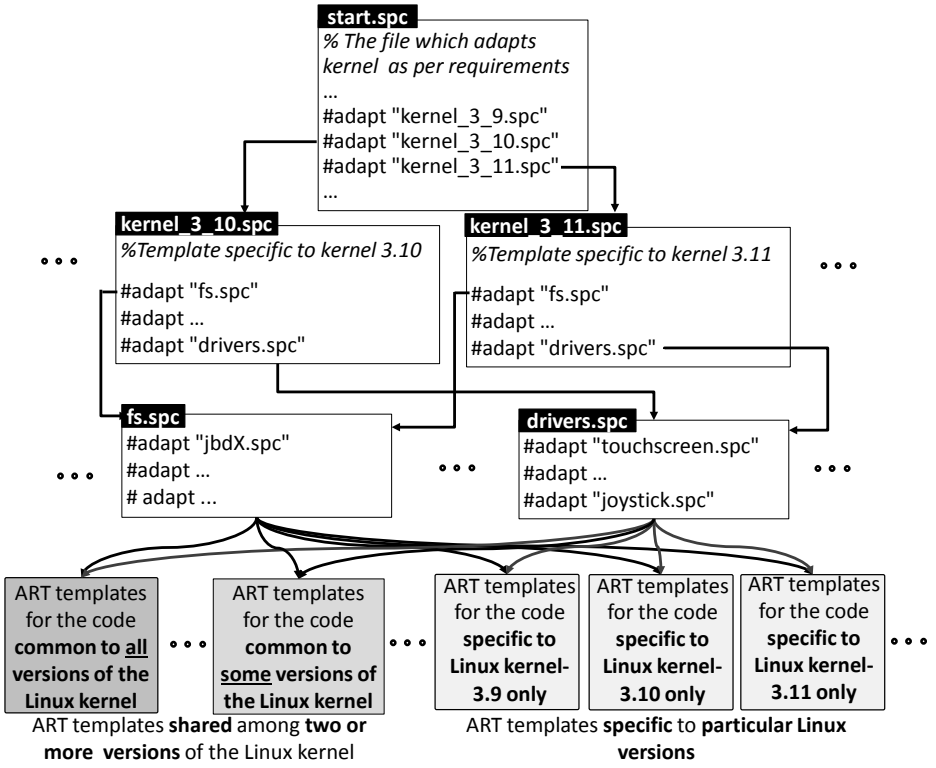


Figure 6.8. Umbrella templates for an overall ART-template solution

6.1.5.4. Handling Evolutionary Changes

Evolution often brings forward changes to the requirements and related code. In this case, ART templates help in easy but disciplined evolution of the software. In an ART-template solution, certain types of changes (e.g., in case of Notepad system—addition of a new menu-item in the Notepad which has code similar to the other menu-items except some parametric variations) can be easily accommodated by modifying the values of a few particular ART variables (e.g., by assigning one or more additional values to the respective ART variables). For drastic changes such as the additions of new methods (e.g., a method to print the content of the Notepad), the proposed solution merely requires adapting a few ART templates at various variation points using ART commands.

Similarly, for example, in the Linux kernel, there might be a need to add a new directory /jbd3, or add more files to the JBD directories. The ART has provisions to accommodate evolutionary changes to the templates (e.g., adding jbd3), without affecting existing code derived from the templates (e.g., jbd and jbd2). Assuming that the new directory /jbd3 also contains six files that are similar to their counterparts in the /jbd and /jbd2, we need to make the following changes to the templates shown in Figure 5.18:

jbdX.spc:

```
  #set dirName = "jbd", "jbd2", "jbd3"
  #set fileName = "checkpoint",…, "recovery"
  …
  #while dirName , action,…, tagByte
  #while filename
  #output ?@dirName?"/"?@fileName?".c"
       #adapt ?@fileName?".spc"
       …
```

checkpoint.spc:

#adapt: "checkpoint.art"
#select dirName
#option **jbd3**
…

checkpoint.art:

// Customizations to checkpoint.art specific to jbd3

// Customizations to the other ART templates considering jbd3

In case of new variation points between the template and the file in /jbd3, we place new #break commands in the template. These new #break commands will cater to the differences specific to /jbd3, injected by #insert commands in "#option jbd3" without affecting the /jbd or /jbd2.

6.1.6. Trade-offs and Threats to Validity of Results

The main returns on investment of applying the ART are increased reuse opportunities, reduced program understanding and maintenance efforts, and non-redundant source code. However, applying a new technique is not free; it entails cost and involves trade-offs. The flexibility of manipulating the code in an unrestricted way comes at the price of not being able to quarantine the correctness of the generated code. Unrestrictive program manipulation decreases the type-safety of the program. In addition, there is a trade-off between the benefits and cost of learning the new technique. ART syntax is very simple and consists of only few constructs (such as #adapt, #while, or #insert-break mechanism). Yet building quality ART templates requires skilled experts, and as such the benefits of the ART are offset by the burden of learning and adopting it.

The benefits of the ART depend on the degree of redundancy in a software system that cannot be fixed by simple refactoring. The bigger the size of

software systems, the higher the likelihood of redundancies and evolutionary changes, and hence the greater the benefits of using the ART. It follows that families of similar systems should be prime candidates for ART-template views, as there is much similarity among components of such systems. Thus, the proposed technique seems to have more direct relevance in the SPL context, where we have the role of domain engineer who is responsible for building reuse-based productivity solutions that serve many systems in long run. ART templates belong to that category of solutions.

6.2. Related Works

This section discusses various available works similar to the proposed work. The discussion has been grouped into various subsections.

6.2.1. Managing Redundancies in Software Systems

Simple-minded development often leads to cloning in various forms (the copy-paste-modify practice). As mentioned earlier, cloning may also be done for good reasons [17]. Still, non-redundancy has always been considered an important quality of well-designed software. The Software Engineering principle of generality encourages the avoidance of repetitions and the building of parameterized software solutions that can be reused in many contexts. Macros were an early attempt to make programs adaptable to various contexts. Goguen popularized the ideas of parameterized programming [157]. Among programming language features, type parameterization [158] (called generics in Ada, Eiffel, Java and C#, and templates in C++), higher-order functions, and inheritance can help avoid repetitions in certain situations. Design techniques such as iterators, design patterns, table-driven design (e.g.,

in compiler-compilers), and modularization with information hiding are supportive in building generic programs. The Standard Template Library (STL) is a prime example of engineering benefits gained by generality [144]. Techniques have also been proposed to lift sufficient code similarity from the code to the architectural level [159, 160].

Compared to other approaches that strive for generality, the ART uses templates and code generation to achieve non-redundancy. ART templates can represent any groups of clones (e.g., files, directories, or patterns of collaborating components) with arbitrary differences among them (as opposed to only type-parametric differences in C++ templates or Java generics). From the ART-template solution of a clone class, the ART Processor generates code for all the clone instances based on the specifications of deltas, i.e., the differences between the template and each of the clone instances in a clone class.

6.2.2. ART versus XVCL

The XVCL has been effectively used to achieve non-redundancy in the program areas where it matters. It includes many case studies including industrial projects [21, 22, 118, 129, 138, 140, 153, 155]. The ART improves and enhances the concepts of the XVCL, implementing them in a way that lets developers easily blend ART's management capabilities with other programming technologies of their choice. The user can define her own syntax to avoid conflicts with native languages, and to make it easy to use the ART with other management techniques. Despite user-defined syntax, the ART

further improves the user experience by providing the following improvements to the XVCL:

- *Easy to learn:* The XVCL is a dialect of XML and uses XML trees and a parser for processing. It is a prerequisite to know the XML syntax and rules for understanding and writing XVCL source. The ART parts with XML syntax and processing. It offers a cpp-based flexible and more readable user-defined syntax. This makes learning the ART easy. Figure 6.9 shows a code fragment in XVCL syntax and ART syntax.

ABC.xvcl

```
1  <?xml version="1.0"?>
2  <!DOCTYPE x-frame SYSTEM "DTD_location">
3  <x-frame name="ABC.xvcl" outdir="dir-name" outfile="file-name.c" language="c">
4     <set-multi variable="index" value="1,2,3"/>
5     <adapt x-frame="XYZ.xvcl">
6        <insert break="1">
7           break1_content
8        </insert>
9        <insert break="2">
10          break2_content
11       </insert>
12       <insert break="3">
13          break3_content
14       </insert>
15    </adapt>
16 </x-frame>
```

ABC.art

```
1  #output file-name.c
2  #set index = 1, 2, 3
3  #adapt: XYZ.art
4    #insert 1
5       break1_content
6    #endinsert
7    #insert 2
8       break2_content
9    #endinsert
10   #insert 3
11      break3_content
12   #endinsert
13 #endadapt
```

Figure 6.9. A code fragment in XVCL (left) vs ART (right) syntax

- *More generalized:* Unlike the XVCL, developers can easily blend the ART with the programming technologies of their choice. This is because the developers can define their own syntax (i.e., can redefine default ART syntax as per her requirements), and hence avoid conflicts with the base languages.
- *Expanding the customization options under #adapt command:* In the XVCL, the only command that you can place under the <adapt> is <insert>. The ART allows the use of any command under #adapt. Using #set, #while, and #select commands under #adapt is particularly very useful. For example, Figure 6.10 shows ART template

ByteBufferAs[T]BufferR[BL].spc for one of the similarity groups of the Java Buffer library. As shown in the figure, lines 9–16 define #select under #adapt command (line 8).

ByteBufferAs[T]BufferR[BL].spc

```
1  #set java_nio_packageName = "java.nio"
2  #set elmtType = "Char","Double","Float","Int","Long","Short"
3  #set elmttype = "char","double","float","int","long","short"
4  #set elmtSize = 1,3,2,2,3,1
5  #set ByteOrder = "B","L“
6  #while elmtType, elmttype, elmtSize
7    #while ByteOrder
8      #adapt: "BufferExample/ByteBufferAs[T]BufferR[BL]/ByteBufferAs[T]BufferR[BL].art"
9        #select elmtType
10         #option Char
11           #insert-after moreMethods
12             #adapt "BufferExample/ByteBufferAs[T]BufferR[BL]/ByteBufferAsCharBufferR_methods.art"
13           #endinsert
14         #endoption
15         …
16       #endselect
17     #endadapt
18   #endwhile
19 #endwhile
```

Figure 6.10. Using commands other than #insert under #adapt

- *Robust structure instead of unreadable loops:* In the XVCL, while loops using many multi-value variables can be quite confusing. The ART introduces a structure called set-loop (#setloop command) which gives the possibility to store and use more multi-value variables together as one loop descriptor data structure. Section 5.2.2 provides complete detail of the #setloop command with illustrative examples.
- *More flexible:* The ART is more flexible than the XVCL, as it allows the adaptation of a file even though the file might not contain any ART commands. Such adaptation would simply copy the adapted file to the output stream. For example, three buffer classes (i.e., Buffer.java, MappedByteBuffer.java, and StringCharBuffer.java) can be easily adapted without any modification in the complete ART-template solution of the Java Buffer library. This is not possible with the XVCL. One needs to convert them into XVCL files before adapting in the

XVCL solution. It incurs additional overhead to the XVCL when compared with the ART.

6.2.3. ART versus Preprocessors

One can also achieve non-redundancy by parameterizing and wrapping the code with preprocessors, shell scripts, and make files. An example of this can be found in the JDK Buffer library described in [140]. SUN developers used cpp, scripts, and make files to build a non-redundant representation from which actual Buffer classes are derived. A quick inspection of the code reveals that such representation may serve only its authors and cannot be considered a viable method to engineer programs.

Preprocessors (such as M4 [161], cpp [148]) are also one of the oldest mechanisms to achieve variability in software [162, 163]. They work on the principle of code expansion. A preprocessor allows macros in the code to be replaced by the text defined by the corresponding macros. This text may contain program code or may contain invocations to other macros. Further, preprocessor directives (such as #ifdef, #else, #endif in cpp; ifdef, ifelse in M4) allow marking variation points in the software. It enables preprocessors to include or exclude specific code segments in software [162, 163]. But, it is found that programs instrumented with preprocessor directives become difficult to understand, test, maintain, and reuse [164]. It may be error-prone and may not scale well [165]. Since preprocessors handle variant features at the implementation level only, it may cause problems when trying to tackle more complex change situations with preprocessors [164]. This observation is drawn from Nokia projects in which preprocessing and file-level configuration

management were used to manage variability. Similar problems with preprocessing were also reported in a research project FAME-DBMS [166, 167]. Still, preprocessors stay popular and seem indispensable for many tasks where programming languages features do not suffice [168].

Like preprocessors, the ART also works on the principle of code expansion. Some of the ART commands have close counterparts too. For example, closest alternatives to ART command #adapt are #include directive in cpp and include (or sinclude) in M4. However, unlike #include and include/sinclude directives, #adapt command allows the same template to be customized differently in different scenarios in which it is reused (by specifying customizations under extended #adapt command as shown in Section 5.2.2). Similarly, #set command in the ART corresponds to setting variables during preprocessing (using #define directive in cpp; define in M4). However, ART variables allow the variable's values to be propagated along the adapted templates. Similar to #select command in the ART, #ifdef directive allows conditional compilation. But compared to the ART, conditional compilation does not allow variable references and expressions. It makes preprocessors less flexible. Further, as compared to preprocessors, as explained in Section 5.2.2, the ART supports breakpoints (insert-break mechanism). Breakpoints serve as anchors where additional code can be injected. It makes the ART capable to handle unexpected variations. Constructing ART templates at the first glance may look complex. However, the fact is that the ART is governed by only five important constructs (i.e., #adapt, #output, insert-break mechanism, loops, and selection) that are neatly integrated to form a method that can be learned easily. Experience with the ART predecessor, the XVCL, demonstrates that

large code can be effectively managed, achieving non-redundancy in the program areas where it matters. For example, the XVCL was used to represent a family of web portals [21] achieving improvement of all major maintainability metrics such as the physical size, the number managed files, and the effort to perform enhancements. Therefore, large code can be effectively managed using the ART.

6.2.4. Variability Management in SPL

Companies today often develop and maintain custom versions of the same software system for different customers using SPL [9]. The core idea is to manage the system family as a whole from a base of core assets designed for ease of adaptation in various reuse contexts.

Motivated by the problems of managing variability purely at the code level, SPL approach emphasizes architectural design and design level configuring of software. Despite benefits of architecture- and component-based approaches for reuse, problems of configuring code cannot be avoided. Mappings between SPL features and specific variation points in code affected by SPL features remain complex. Problems magnify in the presence of feature dependencies, when the presence or absence of one feature affects the way other features are implemented [21, 164, 166, 169-171]. The impact of feature variability and dependencies that cannot be contained at the level of architecture must be handled in code, often manifesting as overly complex conditions under #if, or many nesting levels under #ifdef cpp directives. Such code becomes difficult to understand, test, maintain, and reuse. It is difficult to see which code belongs to which option, and to understand or change program in general.

It is common to use a range of variability techniques to aid in configuring architectures and code, such as preprocessing, software configuration management tools, parameter configuration files or wizards, parameterization, build tools, and sometimes design patterns. These common variability techniques are easily available and each one is easy to use, however they were not designed to work together or handle problems of the scale. Therefore, an overall solution to variability management cannot be smoothly integrated and automated, and may require extensive manual, error-prone interventions during reuse-based development [169, 171]. In view of those problems, effective strategies for automated and reliable variability management in SPL remains one of the main challenges for SPL practice and a central theme of SPL research [9, 172].

In the SPL context, the ART attempts to capture and streamline the end-to-end process of adapting software from the specifications of variant features to the architectural structures and the code. ART templates can manipulate any textual file independent of their contents. Therefore, the ART can also manage variability in documentation and test cases, keeping all textual SPL core assets in sync with evolving code.

Techniques proposed in research to manage variability in SPL are mostly based on the principle of separation of concerns, introduced by Dijkstra in the early 1980s [173]. The goal of separation of concerns is to deal with concerns one by one, independently from other concerns. When applied at the level of design and implementation, separation of concerns attempts to compose software from components implementing different concerns. Concerns that fit nicely into conventional modules are easy to deal with. The challenge is to

tackle cross-cutting concerns that are tightly coupled with the rest of a program, and cannot be easily modularized in a conventional way. There have been attempts to bring separation of concerns down to the design and implementation levels. Aspect-oriented programming (AOP) [121], multi-dimensional separation of concerns (MDSOC) from IBM [174], feature-oriented programming (FOP) [175], and colored IDE (CIDE) [166] are among the most widely published such techniques.

In AOP [121], various computational aspects are programmed separately and weaved at specified join points into the base program. AOP can separate a range of programming aspects, such as persistence, synchronization, or authentication/authorization. Separated aspects can be easily modified, added, or deleted to/from the program modules. Because of this, a number of authors have proposed AOP as a variability technique in SPL. A study to test this hypothesis revealed difficulties in using AspectJ to deal with features that have a chaotic impact on the base code [170]. While AOP deals with big chunks of functionalities (i.e., aspects) reasonably, it lacks a mechanism to handle variations at the lower-levels of granularity. The ART, on the other hand, can handle variations at any level of the granularity. Also, there is a fixed set of joinpoints defined in AOP. Compared to this, breakpoints in the ART can be defined anywhere in the program whenever needed. Using breakpoints, we can explicitly mark the variation points where specific code to a variant can be easily inserted. However, there is also a disadvantage of the ART as compared to AOP. The ART requires additional cost in creating templates for the code before adaptation. Whereas in case of AOP, there is no need to modify the existing program before weaving begins.

MDSOC [174] permits separations of overlapping concerns along multiple dimensions of compositions and decompositions. MDSOC introduces hyperslices that encapsulate specific concerns, and can be composed in various configurations to form custom programs. Unlike the ART, hyperslices are written in the underlying programming language, and can be composed by merging or overriding program units by name, and in many other ways. Unlike MDSOC, the ART is independent of the underlying programming language. It does not rely on any type of the abstract specifications that are associated with the programming language of the native code. Actually, the ART offers uniform mechanism to handle variability. It means that it can be used to handle variability in a variety of interrelated SPL assets such as architecture, code components, domain models, documentations, test cases, etc.

FOP [175] is based on the principle of feature modularization and composition into a base program. Feature modularization helps in understanding and maintaining the feature code. Feature composition extends the base program with the required features. FOP provides a powerful solution for feature management in many situations, but may not be geared for features that have complex mappings to the code [166]. Therefore, Kästner et al. [166] relaxed the requirement for feature modularization, and revisited the idea of keeping feature-related code together with the base code. They proposed a tool CIDE, that provides a visual means for understanding and manipulating the features. CIDE represents a base program as an abstract syntax tree, which makes it language-dependent. Compared with these techniques, the ART is strictly language-independent. The ART's adaptations are defined in an operational

way, and take place at designated variation points marked with the ART commands only.

Further, compared to all these techniques, the ART may facilitate generation of multiple custom program structures from their template representation (using #while command). However, there is no counterpart to this in these techniques that are based on the principle of separation of concerns only. Further, ART expressions and #select command allow concerns to be parameterized. It helps in enhancing the users' abilities to define variations in the code at any level of granularity, from a code fragments to class, to file, to subsystems, or to any component of higher granularity.

As mentioned in Section 6.2.3, preprocessors can also be used to separate code for variant features [162]. The ART adds a non-redundancy layer on top of separation of concerns achieved by preprocessors, without changing the way preprocessors are configured in native code. Non-redundant ART-template views of programs lessen variability management, as one variation point in an ART template represents 'n' variations points in instances of that template, where 'n' is the number of instances of the template in a program. The capability to deal with redundancies is what distinguishes the ART from the techniques proposed by others.

6.3. Conclusions

In this chapter, we quantitatively and qualitatively evaluated the effectiveness, usefulness, and benefits of managing code clones using the ART. We highlighted the applications of the ART on three case studies: the Java buffer library, Notepad system, and a part of the Linux kernel. We also presented

discussions of and comparisons of the proposed work with existing related works.

The benefits of managing clones using the ART include increased reuse opportunities, reduced program understanding and maintenance efforts, simplification of product line core assets due to non-redundancy, and easier comprehension and traceability of change impact during evolution. ART templates help the developers in implementing maintenance changes in a more reliable way. It is to mention that ART templates are not created for quick gains during development, but for long-term gains during software evolution and reuse. ART aims to benefit long-lived systems that undergo extensive evolutionary changes, or need to be tailored to the needs of multiple customers.

Chapter 7.

CONCLUSIONS AND FUTURE WORK

In the thesis, we formalized the concept of collaborative patterns and work out for efficient and scalable algorithms for detecting collaborative patterns in software systems. We demonstrated usefulness of collaborative clone detection in software reuse, re-engineering, and maintenance. This chapter concludes the thesis. While thesis is summarized in Section 7.1, Section 7.2 outlines future research directions.

7.1. Summary

We surveyed state-of-the-art works done in the area of clone detection. We reviewed existing clone taxonomies, detection approaches, and evaluation techniques. Appendix A provides a comprehensive literature survey on relevant prior work. This survey gave us rudimentary details of state-of-the-art works available in the area of software clone detection.

Based on the literature reviewed, we found that existing clone detection techniques mainly focus on detecting similar code fragments, files, or directories. But many design-level similarity patterns appear as the recurring

configurations of collaborating components such as methods, functions, classes, or any physical or logical groups of program entities. We call such types of recurring configurations as the collaborative patterns. Collaborative patterns often represent program structures exhibiting specific behavior meaningful to developers who need to understand programs, reengineer legacy code for reuse, or to refactor or simply maintain programs. Unfortunately, unless manually documented, collaborative patterns remain implicit in code.

We formalized the notion of collaborative patterns in Chapter 2. The term collaborative pattern is defined precisely in terms of a directed graph. In the directed graph, nodes are program entities and edges are calling relationships among the program entities. We further showed possible classification of collaborative patterns. Collaborative patterns are higher-level clones of large granularity that can be identified by systematically combining small-granular cloned program entities at various levels. Based on this, we presented our approach for detecting collaborative patterns in Chapter 3. The proposed approach first finds method clones and calling-relationship information from the subject program, and then uses this information for detecting collaborative patterns. We implemented the proposed approach into a tool called CoPAD (Collaborative Patterns Detector). We also evaluated the proposed approach via experimentation in Chapter 4.

Finally, we proposed a methodology to manage high-level clones of large granularity (collaborative patterns as well as other large-granular code clones) by presenting a meta-programming technique and tool, the ART (Adaptive Reuse Technique). The ART is an enhanced, lightweight and XML-free version of the XVCL. It manages families of redundant software systems by

providing a common base of non-redundant, adaptable, and reusable meta-components called ART templates. We presented the ART and detailed methodology of using the ART in Chapter 5. We evaluated quantitatively and qualitatively the strengths, weakness, and trade-offs involved in the application of the ART in Chapter 6.

The main novelty of the research lies in the formulation of the concept of collaborative patterns, in the development of the technique for detecting collaborative patterns, and in the development of the technique for managing such patterns in software systems.

7.2. Future Research Directions

The current approach for detecting collaborative pattern uses only calling relationship information among the corresponding program entities. Approaches for collaborative pattern detection based on temporal relations among program entities can be devised as the part of future work as extensions to the current approach.

Visualization is one of the important techniques for similarity analysis. In our list of future works, we plan to develop a rudimentary graphical user interface for visualization and analysis of collaborative patterns.

The proposed approach for detecting collaborative patterns is based on the similarity of program text among the program entities involved in the pattern. In the future research, we will perform investigations for the detection of collaborative patterns in which the program entities are functionally similar.

BIBLIOGRAPHY

[1] I. Sommerville, *Software Engineering*, 7th ed.: Pearson Addison Wesley, 2004.

[2] C. K. Roy, and J. R. Cordy, *A survey on software clone detection research*, Technical Report 541, Queen's University at Kingston, 2007.

[3] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: a systematic review," *Information and Software Technology,* vol. 55, no. 7, pp. 1165-1199, 2013.

[4] H. A. Basit, U. Ali, S. Haque, and S. Jarzabek, "Things structural clones tell that simple clones don't," in 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 275-284.

[5] H. A. Basit, and S. Jarzabek, "A data mining approach for detecting higher-level clones in software," *IEEE Trans. Softw. Eng.,* vol. 35, no. 4, pp. 497-514, 2009.

[6] C. Domann, E. Juergens, and J. Streit, "The curse of copy&paste - Cloning in requirements specifications," in 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM), 2009, pp. 443-446.

[7] Y. Jia, and M. Harman, "Clone detection using dependence analysis and lexical analysis," *Department of Computer Science, King's College London*, pp. 5-8, 2007.

[8] R. Koschke, "Survey of research on software clones," in Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software, 2007.

[9] P. Clements, and L. Northrop, *Software product lines: practices and patterns*: Addition-Wesley, 2002.

[10] M. Bharavi, and K. K. Shukla, "Data mining techniques for software quality prediction," *Software Design and Development: Concepts, Methodologies, Tools, and Applications*, A. Information Resources Management, ed., pp. 401-428, Hershey, PA, USA: IGI Global, 2014.

[11] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An empirical study on inconsistent changes to code clones at the release level," *Science of Computer Programming,* vol. 77, no. 6, pp. 760-776, 2012.

[12] L. Zhenmin, S. Lu, S. Myagmar, and Z. Yuanyuan, "CP-Miner: finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Softw. Eng.,* vol. 32, no. 3, pp. 176-192, 2006.

[13] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in 18th ACM symposium on Operating Systems Principles, Alberta, Canada, 2001, pp. 73-88.

[14] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?," in 31st International Conference on Software Engineering (ICSE), 2009, pp. 485-495.

[15] A. Lozano, and M. Wermelinger, "Assessing the effect of clones on changeability," in IEEE International Conference on Software Maintenance (ICSM), 2008, pp. 227-236.

[16] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?," in 7th IEEE Working Conference on Mining Software Repositories (MSR), 2010, pp. 72-81.

[17] C. Kapser, and M. W. Godfrey, ""Cloning considered harmful" considered harmful," in 13th Working Conference on Reverse Engineering (WCRE), 2006, pp. 19-28.

[18] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in 10th European Software Engineering Conference held jointly with 13th International Symposium on Foundations of Software Engineering (ESEC-FSE), 2005, pp. 187-196.

[19] L. Yun, X. Zhenchang, P. Xin, L. Yang, S. Jun, Z. Wenyun, and D. Jinsong, "Clonepedia: summarizing code clones by common syntactic context for software maintenance," in IEEE International Conference on Software Maintenance and Evolution (ICSME), 2014, pp. 341-350.

[20] D. C. Rajapakse, and S. Jarzabek, “Using server pages to unify clones in web applications: A trade-off analysis,” in 29th International Conference on Software Engineering (ICSE), 2007, pp. 116-126.

[21] U. Pettersson, and S. Jarzabek, “Industrial experience with building a web portal product line using a lightweight, reactive approach,” in 10th European Software Engineering Conference held jointly with 13th International Symposium on Foundations of Software Engineering (ESEC/FSE), Lisbon, Portugal, 2005, pp. 326-335.

[22] H. A. Basit, D. C. Rajapakse, and S. Jarzabek, “Beyond templates: a study of clones in the STL and some general implications,” in 27th International Conference on Software Engineering (ICSE), 2005, pp. 451-459.

[23] J. H. Johnson, “Identifying redundancy in source code using fingerprints,” in Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering, 1993, pp. 171-183.

[24] J. H. Johnson, “Substring matching for clone detection and change tracking,” in International Conference on Software Maintenance (ICSM), 1994, pp. 120-126.

[25] B. S. Baker, “A program for identifying duplicated code,” *Computing Science and Statistics,* vol. 24, pp. 49-57, 1992.

[26] E. Buss, R. D. Mori, W. M. Gentleman, J. Henshaw, H. Johnson, K. Kontogiannis, E. Merlo, H. Muller, J. Mylopoulos, S. Paul, A. Prakash, M. Stanley, S. R. Tilley, J. Troster, and K. Wong, “Investigating reverse engineering technologies for the CAS program understanding project,” *IBM Syst. J.,* vol. 33, no. 3, pp. 477-500, 1994.

[27] B. S. Baker, “On finding duplication and near-duplication in large software systems,” in Working Conference on Reverse Engineering (WCRE), 1995, pp. 86-95.

[28] R. Koschke, R. Falke, and P. Frenzel, “Clone detection using abstract syntax suffix trees,” in 13th Working Conference on Reverse Engineering (WCRE), 2006, pp. 253-262.

[29] H. A. Basit, S. J. Puglisi, W. F. Smyth, A. Turpin, and S. Jarzabek, “Efficient token based clone detection with flexible tokenization,” in 6th European Software Engineering Sonference and ACM SIGSOFT

symposium on the Foundations of Software Engineering (ESEC/FSE), 2007, pp. 513-516.

[30] N. Göde, and R. Koschke, “Incremental clone detection,” in 13th European Conference on Software Maintenance and Reengineering (CSMR), 2009, pp. 219-228.

[31] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, “Index-based code clone detection: incremental, distributed, scalable,” in International Conference on Software Maintenance (ICSM), 2010, pp. 1-9.

[32] A. Santone, “Clone detection through process algebras and Java bytecode,” in 5th International Workshop on Software Clones (IWSC), 2011, pp. 73-74.

[33] A. Cuomo, A. Santone, and U. Villano, “A novel approach based on formal methods for clone detection,” in 6th International Workshop on Software Clones (IWSC), 2012, pp. 8-14.

[34] R. Koschke, “Large-scale inter-system clone detection using suffix trees,” in 16th European Conference on Software Maintenance and Reengineering (CSMR), 2012, pp. 309-318.

[35] T. Lavoie, and E. Merlo, “An accurate estimation of the Levenshtein distance using metric trees and Manhattan distance,” in 6th International Workshop on Software Clones (IWSC), 2012, pp. 1-7.

[36] H. Sajnani, J. Ossher, and C. Lopes, “Parallel code clone detection using MapReduce,” in 20th International Conference on Program Comprehension (ICPC), 2012, pp. 261-262.

[37] W. Toomey, “Ctcompare: code clone detection using hashed token sequences,” in 6th International Workshop on Software Clones (IWSC), 2012, pp. 92-93.

[38] R. Koschke, “Large-scale inter-system clone detection using suffix trees and hashing,” *Journal of Software: Evolution and Process,* vol. 26, no. 8, pp. 747-769, 2013.

[39] H. Sajnani, and C. Lopes, “A parallel and efficient approach to large scale clone detection,” in 7th International Workshop on Software Clones (IWSC), 2013, pp. 46-52.

[40] N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley, “The development of a software clone detector,” *International Journal of Applied Software Technology,* vol. 1, no. 3/4, pp. 219-236, 1995.

[41] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, “Clone detection using abstract syntax trees,” in International Conference on Software Maintenance (ICSM), 1998, pp. 368-377.

[42] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: a multilinguistic token-based code clone detection system for large scale source code,” *IEEE Trans. Softw. Eng.,* vol. 28, no. 7, pp. 654-670, 2002.

[43] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, “CP-Miner: a tool for finding copy-paste and related bugs in operating system code,” in 6th Conference on Symposium on Opearting Systems Design & Implementation (OSDI), 2004, pp. 289-302.

[44] S. Lee, and I. Jeong, “SDD: high performance code clone detection system for large scale source code,” in 20th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), 2005, pp. 140-141.

[45] W. S. Evans, C. W. Fraser, and M. Fei, “Clone detection via structural abstraction,” in 14th Working Conference on Reverse Engineering (WCRE), 2007, pp. 150-159.

[46] J. Lingxiao, G. Misherghi, S. Zhendong, and S. Glondu, “DECKARD: scalable and accurate tree-based detection of code clones,” in 29th International Conference on Software Engineering (ICSE), 2007, pp. 96-105.

[47] P. Bulychev, and M. Minea, “Duplicate code detection using anti-unification,” in Spring/Summer Young Researchers’ Colloquium on Software Engineering, Russia, 2008, pp. 51-54.

[48] S. Livieri, and K. Inoue, “YACCA: code clone detection on multi-core processors,” in Workshop on Accountability and Traceability in Global Software Engineering, 2008, pp. 19.

[49] C. K. Roy, and J. R. Cordy, “NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization,” in 16th International Conference on Program Comprehension (ICPC), 2008, pp. 172-181.

[50] W. Evans, C. Fraser, and F. Ma, “Clone detection via structural abstraction,” *Software Quality Journal,* vol. 17, no. 4, pp. 309-330, 2009.

[51] Y. Jia, D. Binkley, M. Harman, J. Krinke, and M. Matsushita, “KClone: a proposed approach to fast precise code clone detection,” in 3rd International Workshop on Detection of Software Clones (IWSC), 2009.

[52] C. K. Roy, “Detection and analysis of near-miss software clones,” in International Conference on Software Maintenance (ICSM), 2009, pp. 447-450.

[53] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello, “A tree kernel based approach for clone detection,” in International Conference on Software Maintenance (ICSM), 2010, pp. 1-5.

[54] M. Funaro, D. Braga, A. Campi, and C. Ghezzi, “A hybrid approach (syntactic and textual) to clone detection,” in 4th International Workshop on Software Clones (IWSC), 2010, pp. 79-80.

[55] C. K. Roy, and J. R. Cordy, “Near-miss function clones in open source software: an empirical study,” *J. Softw. Maint. Evol.,* vol. 22, no. 3, pp. 165-189, 2010.

[56] J. R. Cordy, and C. K. Roy, “The NiCad clone detector,” in 19th International Conference on Program Comprehension (ICPC), 2011, pp. 219-220.

[57] Y. Dang, S. Ge, R. Huang, and D. Zhang, “Code clone detection experience at microsoft,” in 5th International Workshop on Software Clones (IWSC), 2011, pp. 63-64.

[58] Y. Higo, U. Yasushi, M. Nishino, and S. Kusumoto, “Incremental code clone detection: a PDG-based approach,” in 18th Working Conference on Reverse Engineering (WCRE), 2011, pp. 3-12.

[59] Y. Higo, and S. Kusumoto, “Code clone detection on specialized PDGs with heuristics,” in 15th European Conference on Software Maintenance and Reengineering (CSMR), 2011, pp. 75-84.

[60] Y. Yang, and G. Yao, “CMCD: count matrix based code clone detection,” in 18th Asia Pacific Software Engineering Conference (APSEC), 2011, pp. 250-257.

[61] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie, “XIAO: tuning code clones at hands of engineers in practice,” in 28th Annual Computer Security Applications Conference, Orlando, Florida, 2012, pp. 369-378.

[62] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, “Folding repeated instructions for improving token-based code clone detection,” in 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2012, pp. 64-73.

[63] Y. Yang, and G. Yao, “Boreas: an accurate and scalable token-based approach to code clone detection,” in 27th International Conference on Automated Software Engineering (ASE), 2012, pp. 286-289.

[64] M. F. Zibran, and C. K. Roy, “IDE-based real-time focused search for near-miss clones,” in 27th Annual ACM Symposium on Applied Computing, Trento, Italy, 2012, pp. 1235-1242.

[65] B. Muddu, A. Asadullah, and V. Bhat, “CPDP: a robust technique for plagiarism detection in source code,” in 7th International Workshop on Software Clones (IWSC), 2013, pp. 39-45.

[66] M. S. Uddin, C. K. Roy, and K. A. Schneider, “SimCad: an extensible and faster clone detection tool for large scale software systems,” in 21st International Conference on Program Comprehension (ICPC), 2013, pp. 236-238.

[67] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, “Gapped code clone detection with lightweight source code analysis,” in 21st International Conference on Program Comprehension (ICPC), 2013, pp. 93-102.

[68] W. Qu, Y. Jia, and M. Jiang, “Pattern mining of cloned codes in software systems,” *Information Sciences,* vol. 259, pp. 544-554, 2014.

[69] R. Komondoor, and S. Horwitz, "Using slicing to identify duplication in source code," *Static Analysis*, pp. 40-56: Springer Berlin Heidelberg, 2001.

[70] J. Krinke, “Identifying similar code with program dependence graphs,” in 8th Working Conference on Reverse Engineering (WCRE), 2001, pp. 301-309.

[71] M. Gabel, J. Lingxiao, and S. Zhendong, “Scalable detection of semantic clones,” in 30th International Conference on Software Engineering (ICSE), 2008, pp. 321-330.

[72] L. Jiang, and Z. Su, “Automatic mining of functionally equivalent code fragments via random testing,” in 18th International Symposium on Software Testing and Analysis (ISSTA), 2009, pp. 81-92.

[73] H. Kim, Y. Jung, S. Kim, and K. Yi, “MeCC: memory comparison-based clone detector,” in 33rd International Conference on Software Engineering (ICSE), Waikiki, HI, USA, 2011, pp. 301-310.

[74] P. Schugerl, “Scalable clone detection using description logic,” in 5th International Workshop on Software Clones (IWSC), 2011, pp. 47-53.

[75] S. Yoshioka, N. Yoshida, K. Fushida, and H. Iida, “Scalable detection of semantic clones based on two-stage clustering,” in IEEE International Symposium on Software Reliability Engineering (ISSRE), 2011, pp. 3-4.

[76] R. Elva, and G. Leavens, “Jsctracker: a semantic clone detection tool for Java code,” *University of Central Florida, Department of EECS, University of Central Florida,* vol. 4000, 2012.

[77] R. Elva, and G. T. Leavens, “Semantic clone detection using method IOE-behavior,” in 6th International Workshop on Software Clones (IWSC), 2012, pp. 80-81.

[78] J. Li, and M. D. Ernst, “CBCD: cloned buggy code detector,” in International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2012, pp. 310-320.

[79] C. Dandois, and W. Vanhoof, "Semantic code clones in logic programs," *Logic-Based Program Synthesis and Transformation*, Lecture Notes in Computer Science, E. Albert, ed., pp. 35-50: Springer Berlin Heidelberg, 2013.

[80] T. Kamiya, “Agec: an execution-semantic clone detection tool,” in 21st International Conference on Program Comprehension (ICPC), 2013, pp. 227-229.

[81] H. A. Basit, and S. Jarzabek, “Detecting higher-level similarity patterns in programs,” in 10th European Software Engineering Conference held

jointly with 13th International Symposium on Foundations of Software Engineering (ESEC-FSE), Lisbon, Portugal, 2005, pp. 156-165.

[82] Q. Wenyi, P. Xin, X. Zhenchang, S. Jarzabek, and Z. Wenyun, "Mining logical clones in software: Revealing high-level business and programming rules," in 29th IEEE International Conference on Software Maintenance (ICSM), 2013, pp. 40-49.

[83] R. K. Keller, R. Schauer, S. Robitaille, and P. Page, "Pattern-based reverse-engineering of design components," in International Conference on Software Engineering (ICSE), 1999, pp. 226-235.

[84] J. M. Smith, and D. Stotts, "SPQR: flexible automated design pattern extraction from source code," in 18th International Conference on Automated Software Engineering (ASE), 2003, pp. 215-224.

[85] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, "Automatic design pattern detection," in 11th International Workshop on Program Comprehension (ICPC), 2003, pp. 94-103.

[86] N. Tsantalis, A. Chatzigeorgiou, S. T. Halkidis, and G. Stephanides, "A novel approach to automated design pattern detection," in 10th Panhellenic Conference on Informatics, 2005, pp. 238-248.

[87] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design pattern detection using similarity scoring," *IEEE Trans. Softw. Eng.,* vol. 32, no. 11, pp. 896-909, 2006.

[88] S. Romano, G. Scanniello, M. Risi, and C. Gravino, "Clustering and lexical information support for the recovery of design pattern in source code," in 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 500-503.

[89] A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design pattern recovery through visual language parsing and source code analysis," *Journal of Systems and Software,* vol. 82, no. 7, pp. 1177-1193, 2009.

[90] A. Binun, and G. Kniesel, "DPJF - design pattern detection with high accuracy," in 16th European Conference on Software Maintenance and Reengineering (CSMR), 2012, pp. 245-254.

[91] S. Paydar, and M. Kahani, "A semantic web based approach for design pattern detection from source code," in 2nd International eConference on Computer and Knowledge Engineering, 2012, pp. 289-294.

[92] U. Tekin, U. Erdemir, and F. Buzluca, “Mining object-oriented design models for detecting identical design structures,” in 6th International Workshop on Software Clones (IWSC), 2012, pp. 43-49.

[93] Y. Dongjin, Z. Yanyan, G. Jianlin, and W. Wei, “From sub-patterns to patterns: an approach to the detection of structural design pattern instances by subgraph mining and merging,” in 37th Annual Computer Software and Applications Conference (COMPSAC), 2013, pp. 579-588.

[94] D. Yu, G. Jianlin, and W. Wei, “Detection of design pattern instances based on graph isomorphism,” in IEEE International Conference on Software Engineering and Service Science, 2013, pp. 874-877.

[95] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, “MAPO: mining and recommending API usage patterns,” in 23rd European Conference on Object-Oriented Programming, Italy, 2009, pp. 318-343.

[96] G. Uddin, B. Dagenais, and M. P. Robillard, “Analyzing temporal API usage patterns,” in 26th International Conference on Automated Software Engineering (ASE), 2011, pp. 456-459.

[97] E. Kodhai, and S. Kanmani, “Method-level code clone detection through LWH (Light Weight Hybrid) approach,” *Journal of Software Engineering Research and Development,* vol. 2, no. 1, pp. 1-29, 2014.

[98] K. Salwa, and Abd-El-Hafiz, “A metrics-based data mining approach for software clone detection,” in 36th Annual Computer Software and Applications Conference (COMPSAC), 2012, pp. 35-41.

[99] A. Marcus, and J. I. Maletic, “Identification of high-level concept clones in source code,” in 16th International Conference on Automated Software Engineering (ASE), 2001, pp. 107-114.

[100] C. J. Kapser, and M. W. Godfrey, “Supporting the analysis of clones in software systems: Research Articles,” *J. Softw. Maint. Evol.,* vol. 18, no. 2, pp. 61-82, 2006.

[101] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, “Comparison and evaluation of clone detection tools,” *IEEE Trans. Softw. Eng.,* vol. 33, no. 9, pp. 577-591, 2007.

[102] R. Koschke, “Frontiers of software clone management,” in Frontiers of Software Maintenance (FoSM), 2008, pp. 119-128.

[103] K. Kumar, and S. Jarzabek, “Detecting design similarity patterns using program execution traces,” in 2014 ACM SIGPLAN conference on Systems, Programming, and Applications: Software for Humanity (SPLASH), Portland, Oregon, USA, 2014, pp. 55-56.

[104] K. Kumar, “Detecting collaborative patterns in programs,” in 30th IEEE International Conference on Software Maintenance and Evolution (ICSME), Victoria, BC, Canada, 2014, pp. 664-664.

[105] K. Kumar, S. Jarzabek, and D. Daniel, “ART: a meta-programming language for configuring variants in software,” in 12th Asian Symposium on Programming Languages and Systems (APLAS), Singapore, 2014.

[106] K. Kumar, S. Jarzabek, and D. Daniel, “ART for enhancing software maintainability and reusability by managing big clones,” Submitted for journal publication, 30pp., 2015.

[107] S. Jarzabek, and K. Kumar, “Weak separation of tightly coupled concerns with generic program representations,” in PTI 17th KKIO Software Engineering Conference, Miedzyzdroje, Poland, 2015, pp. 119-136.

[108] S. Jarzabek, and K. Kumar, “On Engineering Benefits of Integrating Separation of Concerns and Genericity Principles within a Unified Program Representation Framework,” Submitted for publication, 21pp., 2015.

[109] D. Lo, and S. C. Khoo, “SMArTIC: towards building an accurate, robust and scalable specification miner,” in 14th International Symposium on Foundations of Software Engineering (FSE), Portland, Oregon, USA, 2006, pp. 265-275.

[110] C. K. Roy, and J. R. Cordy, “An empirical study of function clones in open source software,” in 15th Working Conference on Reverse Engineering (WCRE), 2008, pp. 81-90.

[111] J. Mayrand, C. Leblanc, and E. M. Merlo, “Experiment on the automatic detection of function clones in a software system using metrics,” in International Conference on Software Maintenance (ICSM), 1996, pp. 244-253.

[112] A. Walenstein, A. Lakhotia, and R. Koschke, “The second international workshop on detection of software clones: Workshop report,” *SIGSOFT Softw. Eng. Notes,* vol. 29, no. 2, pp. 1-5, 2004.

[113] H. A. Basit, and S. Jarzabek, “A case for structural clones,” in International Workshop on Software Clones (IWSC), 2009.

[114] H. A. Basit, U. Ali, and S. Jarzabek, “Viewing simple clones from structural clones' perspective,” in 5th International Workshop on Software Clones (IWSC), Waikiki, Honolulu, HI, USA, 2011, pp. 1-6.

[115] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*: Addison-Wesley Longman Publishing Co., Inc., 1995.

[116] G. Antoniol, R. Fiutem, and L. Cristoforetti, “Design pattern recovery in object-oriented software,” in 6th International Workshop on Program Comprehension, 1998, pp. 153-160.

[117] A. Alnusair, T. Zhao, and G. Yan, “Rule-based detection of design patterns in program code,” *International Journal on Software Tools for Technology Transfer,* vol. 16, no. 3, pp. 315-334, 2014/06/01, 2014.

[118] "XVCL," 25-Jan-2014; http://xvcl.comp.nus.edu.sg/cms/.

[119] K. Reinloo. "Who is calling this method?," 11-June-2014; https://plumbr.eu/blog/who-is-calling-this-method.

[120] K. Ali, and O. Lhoták, “Application-only call graph construction,” in 26th European Conference on Object-Oriented Programming, Beijing, China, 2012, pp. 688-712.

[121] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-oriented programming,” in European Conference on Object-Oriented Programming, 1997, pp. 220-242.

[122] "WALA: IBM T.J. Watson libraries for analysis," 04-Feb-2015; http://wala.sourceforge.net/.

[123] J. Kärkkäinen, and P. Sanders, “Simple linear work suffix array construction,” in 30th International Conference on Automata, Languages and Programming, 2003, pp. 943-955.

[124] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, "Linear-time longest-common-prefix computation in suffix arrays and its

applications," *Combinatorial Pattern Matching*, pp. 181-192: Springer Berlin Heidelberg, 2001.

[125] "CloneDR," 10-Jan-2014; http://www.semdesigns.com/Products/CloneDR/.

[126] "AspectJ," 06-Jan-2014; http://eclipse.org/aspectj/.

[127] Z. Yali, H. A. Basit, S. Jarzabek, A. Dang, and M. Low, "Query-based filtering and graphical view generation for clone analysis," in International Conference on Software Maintenance (ICSM), 2008, pp. 376-385.

[128] "JHotDraw," 19-Feb-2015; http://www.jhotdraw.org/.

[129] S. Jarzabek, *Effective software maintenance and evolution: A reuse-based approach*: Auerbach Publications, 2007.

[130] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida, "SHINOBI: A tool for automatic code clone detection in the IDE," in 16th Working Conference on Reverse Engineering (WCRE), 2009, pp. 313-314.

[131] M. Rieger, "Effective clone detection without language barriers," PhD Thesis at the University of Bern, 2005.

[132] V. Bauer, and B. Hauptmann, "Assessing cross-project clones for reuse optimization," in 7th International Workshop on Software Clones (IWSC), 2013, pp. 60-61.

[133] M. Fowler, *Refactoring: improving the design of existing code*: Addison-Wesley Professional, 1999.

[134] F. Arcelli Fontana, M. Zanoni, A. Ranchetti, and D. Ranchetti, "Software clone detection and refactoring," *ISRN Software Engineering,* vol. 2013, pp. 1-8, 2013.

[135] N. Tsantalis, and G. P. Krishnan, "Refactoring clones: A new perspective," in 7th International Workshop on Software Clones (IWSC), 2013, pp. 12-13.

[136] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano, "Extracting code clones for refactoring using combinations of clone metrics," in 5th International Workshop on Software Clones (IWSC), USA, 2011, pp. 7-13.

[137] G. P. Krishnan, and N. Tsantalis, “Unification and refactoring of clones,” in IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014, pp. 104-113.

[138] S. Jarzabek, and S. Li, “Unifying clones with a generative programming technique: a case study,” *J. of Software Maintenance and Evolution: Research and Practice,* vol. 18, no. 4, pp. 267-292, 2006.

[139] S. Schulze, S. Apel, and C. Kästner, “Code clones in feature-oriented software product lines,” in 9th International Conference on Generative Programming and Component Engineering, 2010, pp. 103-112.

[140] S. Jarzabek, and L. Shubiao, “Eliminating redundancies with a "composition with adaptation" meta-programming technique,” in 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC-FSE), 2003, pp. 237-246.

[141] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, “An exploratory study of cloning in industrial software product lines,” in 17th European Conference on Software Maintenance and Reengineering (CSMR), 2013, pp. 25-34.

[142] S. Giesecke, "Generic modelling of code clones," *Duplication, Redundancy, and Similarity in Software*, Dagstuhl Seminar Proceedings R. Koschke, E. Merlo and A. Walenstein, eds., pp. 1-23, Germany, 2007.

[143] C. K. Roy, M. F. Zibran, and R. Koschke, “The vision of software clone management: Past, present, and future (Keynote paper),” in IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014, pp. 18-33.

[144] D. R. Musser, G. J. Derge, and A. Saini, *STL tutorial and reference guide: C++ programming with the standard template library*: Addison-Wesley Professional, 2009.

[145] H. A. Basit, D. C. Rajapakse, and S. Jarzabek, “An empirical study on limits of clone unification using generics,” in 17th International Conference on Software Engineering and Knowledge Engineering (SEKE), 2005, pp. 109-114.

[146] K. Kontogiannis, "Managing known clones: issues and open questions," *Duplication, Redundancy, and Similarity in Software*, Dagstuhl Seminar Proceedings, 2007.

[147] N. Göde, *Clone evolution*: Logos Verlag Berlin GmbH, 2011.

[148] R. M. Stallman, and Z. Weinberg, "The C preprocessor," *Free Software Foundation*, 1987.

[149] "ANTLR," 22-Jan-2014; http://www.antlr.org/.

[150] A. Kadav, and M. M. Swift, "Understanding modern device drivers," in International Conference on Architectural Support for Programming Languages and Operating Systems, UK, 2012, pp. 87-98.

[151] C. Kapser, and M. W. Godfrey, "Toward a taxonomy of clones in source code: A case study," in Conference on Evolution of Large Scale Industrial Software Architectures (ELISA'03), 2003, pp. 67-78.

[152] G. Casazza, G. Antoniol, U. Villano, E. Merlo, and M. Di Penta, "Identifying clones in the Linux kernel," in International Workshop on Source Code Analysis and Manipulation (SCAM), 2001, pp. 90-97.

[153] S. M. Swe, H. Zhang, and S. Jarzabek, "XVCL: a tutorial," in 14th International Conference on Software Engineering and Knowledge Engineering (SEKE), 2002, pp. 341-349.

[154] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, *Feature-oriented domain analysis (FODA) feasibility study*, DTIC Document, 1990.

[155] S. Jarzabek, U. Pettersson, and H. Zhang, "University-industry collaboration journey towards product lines," in 12th International Conference on Software Reuse (ICSR), Pohang, South Korea, 2011, pp. 223-237.

[156] F. P. Brooks, Jr., "No silver bullet essence and accidents of software engineering," *Computer,* vol. 20, no. 4, pp. 10-19, 1987.

[157] J. A. Goguen, "Parameterized programming," *IEEE Trans. Softw. Eng.,* vol. SE-10, no. 5, pp. 528-543, 1984.

[158] R. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock, "A comparative study of language support for generic programming," in 18th Annual Conference on Object-oriented Programing, Systems, Languages, and Applications (OOPSLA), USA, 2003, pp. 115-134.

[159] T. Mende, R. Koschke, and F. Beckwermert, “An evaluation of code similarity identification for the grow-and-prune model,” *J. Softw. Maint. Evol.,* vol. 21, no. 2, pp. 143-169, 2009.

[160] P. Frenzel, R. Koschke, A. P. J. Breu, and K. Angstmann, “Extending the reflexion method for consolidating software variants into product lines,” in 14th Working Conference on Reverse Engineering (WCRE), 2007, pp. 160-169.

[161] "M4 " 15-May-2015; http://www.gnu.org/software/m4/m4.html.

[162] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, “An analysis of the variability in forty preprocessor-based software product lines,” in 32nd ACM/IEEE International Conference on Software Engineering (ICSE), Cape Town, South Africa, 2010, pp. 105-114.

[163] C. Kästner, P. G. Giarrusso, and K. Ostermann, “Partial preprocessing C code for variability analysis,” in 5th Workshop on Variability Modeling of Software-Intensive Systems, Belgium, 2011, pp. 127-136.

[164] A. Karhinen, A. Ran, and T. Tallgren, “Configuring designs for reuse,” in 19th International Conference on Software Engineering (ICSE), 1997, pp. 701-710.

[165] H. Spencer, “ifdef Considered Harmful, or Portability Experience with C News,” in Summer'92 USENIX Conference, 1992.

[166] C. Kästner, S. Apel, and M. Kuhlemann, “Granularity in software product lines,” in 30th International Conference on Software Engineering (ICSE), Leipzig, Germany, 2008, pp. 311-320.

[167] M. Rosenmüller, N. Siegmund, H. Schirmeier, J. Sincero, S. Apel, T. Leich, O. Spinczyk, and G. Saake, “FAME-DBMS: tailor-made data management solutions for embedded systems,” in EDBT workshop on Software Engineering for Tailor-made Data Management, Nantes, France, 2008, pp. 1-6.

[168] M. D. Ernst, G. J. Badros, and D. Notkin, “An empirical analysis of C preprocessor use,” *IEEE Trans. Softw. Eng.,* vol. 28, no. 12, pp. 1146-1170, 2002.

[169] P. Ye, X. Peng, Y. Xue, and S. Jarzabek, “A case study of variation mechanism in an industrial product line,” in 11th International Conference on Software Reuse (ICSR), 2009, pp. 126-136.

[170] C. Kästner, S. Apel, and D. Batory, “A case study implementing features using AspectJ,” in 11th International Software Product Line Conference (SPLC), 2007, pp. 223-232.

[171] S. Deelstra, M. Sinnema, and J. Bosch, "Experiences in software product families: Problems and issues during product derivation," *Software Product Lines*, Lecture Notes in Computer Science R. Nord, ed., pp. 165-182: Springer Berlin Heidelberg, 2004.

[172] M. Svahnberg, J. v. Gurp, and J. Bosch, “A taxonomy of variability realization techniques: Research Articles,” *Softw. Pract. Exper.,* vol. 35, no. 8, pp. 705-754, 2005.

[173] E. W. Dijkstra, "On the role of scientific thought," *Selected Writings on Computing: A Personal Perspective*, pp. 60-66: Springer, 1982.

[174] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton, “N degrees of separation: multi-dimensional separation of concerns,” in International Conference on Software Engineering (ICSE), 1999, pp. 107-119.

[175] D. Batory, J. N. Sarvela, and A. Rauschmayer, “Scaling step-wise refinement,” in 25th International Conference on Software Engineering (ICSE), Portland, Oregon, 2003, pp. 187-197.

[176] E. Juergens, “Research in cloning beyond code: a first roadmap,” in International Workshop on Software Clones (IWSC), 2011, pp. 67-68.

[177] B. S. Baker, and R. Giancarlo, “Sparse dynamic programming for longest common subsequence from fragments,” *Journal of Algorithms,* vol. 42, no. 2, pp. 231-254, 2002.

[178] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, “Replacing suffix trees with enhanced suffix arrays,” *J. of Discrete Algorithms,* vol. 2, no. 1, pp. 53-86, 2004.

[179] E. Juergens, F. Deissenboeck, and B. Hummel, “CloneDetective - a workbench for clone detection research,” in 31st International Conference on Software Engineering (ICSE), 2009, pp. 603-606.

[180] X. Yan, J. Han, and R. Afshar, “CloSpan: mining closed sequential patterns in large datasets,” in International Conference Data Mining (SDM), 2003, pp. 166-177.

[181] T. F. Smith, and M. S. Waterman, “Identification of common molecular subsequences,” *Journal of molecular biology,* vol. 147, no. 1, pp. 195-197, 1981.

[182] S. Horwitz, “Identifying the semantic and textual differences between two versions of a program,” in ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), USA, 1990, pp. 234-245.

[183] "Simian," 12-Jan-2014; http://www.harukizaemon.com/simian/.

[184] H. Liu, Z. Ma, L. Zhang, and W. Shao, “Detecting duplications in sequence diagrams based on suffix trees,” in Asia Pacific Software Engineering Conference (APSEC), 2006, pp. 269-276.

[185] H. Störrle, “Towards clone detection in UML domain models,” in 4th European Conference on Software Architecture, Copenhagen, Denmark, 2010, pp. 285-293.

[186] E. P. Antony, M. H. Alalfi, and J. R. Cordy, “An approach to clone detection in behavioural models,” in 20th Working Conference on Reverse Engineering (WCRE), 2013, pp. 472-476.

[187] H. Störrle, “Towards clone detection in UML domain models,” *Software & Systems Modeling,* vol. 12, no. 2, pp. 307-329, 2013.

[188] F. Deissenboeck, B. Hummel, E. Jurgens, B. Schatz, S. Wagner, J. F. Girard, and S. Teuchert, “Clone detection in automotive model-based development,” in 30th International Conference on Software Engineering (ICSE), 2008, pp. 603-612.

[189] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, “Complete and accurate clone detection in graph-based models,” in 31st International Conference on Software Engineering (ICSE), 2009, pp. 276-286.

[190] B. Hummel, E. Juergens, and D. Steidl, “Index-based model clone detection,” in 5th International Workshop on Software Clones (IWSC), Waikiki, Honolulu, HI, USA, 2011, pp. 21-27.

[191] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson, “Models are code too: near-miss clone detection for Simulink models,” in 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 295-304.

[192] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson, "Near-miss model clone detection for Simulink models," in 6th International Workshop on Software Clones (IWSC), 2012, pp. 78-79.

[193] J. R. Cordy, "The TXL source transformation language," *Science of Computer Programming,* vol. 61, no. 3, pp. 190-210, 2006.

[194] F. Hermans, B. Sedee, M. Pinzger, and A. v. Deursen, "Data clone detection and visualization in spreadsheets," in International Conference on Software Engineering (ICSE), USA, 2013, pp. 292-301.

[195] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit, "Can clone detection support quality assessments of requirements specifications?," in 32nd International Conference on Software Engineering (ICSE), 2010, pp. 79-88.

[196] M. Fisher, and G. Rothermel, "The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms," *SIGSOFT Softw. Eng. Notes,* vol. 30, no. 4, pp. 1-5, 2005.

[197] S. Asaithambi, and S. Jarzabek, "Towards test case reuse: a study of redundancies in android platform test libraries," in International Conference on Software Reuse (ICSR), 2013, pp. 49-64.

[198] I. Keivanloo, C. K. Roy, and J. Rilling, "SeByte: a semantic clone detection tool for intermediate languages," in 20th IEEE International Conference on Program Comprehension (ICPC), 2012, pp. 247-249.

[199] I. Keivanloo, C. K. Roy, and J. Rilling, "Java bytecode clone detection via relaxation on code fingerprint and Semantic Web reasoning," in 6th International Workshop on Software Clones (IWSC), 2012, pp. 36-42.

[200] A. De Lucia, G. Scanniello, and G. Tortora, "Identifying clones in dynamic web sites using similarity thresholds," in International Conference on Enterprise Information Systems, 2004, pp. 391-396.

[201] M. Lebon, and V. Tzerpos, "Fine-grained design pattern detection," in 36th IEEE Annual Computer Software and Applications Conference (COMPSAC), 2012, pp. 267-272.

Appendix A.

LITERATURE REVIEW

Clone detection in software systems is an active area of research. Various tools and techniques have been proposed in the literature for detecting cloned program entities. Literature suggests that cloning may occur in source code or even in other software artifacts such as test cases, use cases, or UML models [176]. Based on the type of clones typically possible in software systems, Figure A.1 shows taxonomy for software clones. In this appendix, we discuss some of the reported works on software code clone detection in line with this taxonomy. We begin with brief description of the clone types and existing literatures. Thereafter, we compare the salient features of reviewed clone detection approaches in tabular form.

This appendix is organized as follows: Sections A.1 and A.2 discuss existing literature on software clone detection. Tables (Table 8–Table 11) at the end of subsections provide tabular comparison of similar types of code clone detection techniques. Through the literature, we found that cloning occur in software artifacts other than code too; some cases are presented in Sections A.3 and A.4. Section A.5 provides chronology of the clone detection techniques.

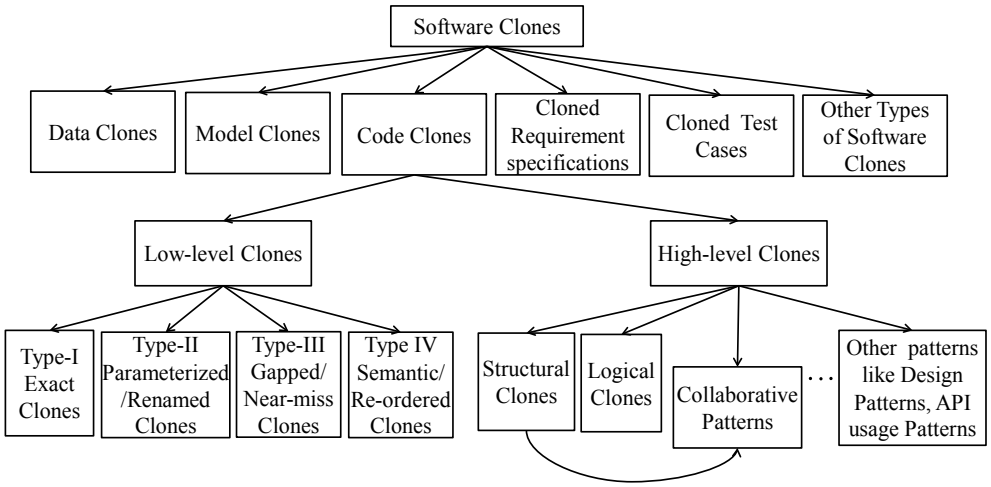


Figure A.1. Taxonomy for software clones

A.1. Low-level Clone Detection

There are basically two kinds of similarity between code fragments: one is syntactic similarity (having similar text) and another is semantic similarity (similar functionality) [2]. In early 1995, Davey et al. [40] provided the first ever clone topology based on the levels of these similarities. They divided the cloned code fragment into four types with an increasing level of subtlety from one type to other. The definitions and types of low-level clones described in this section are based upon the intuitions derived directly from this topology. Based upon the syntactic similarity, cloned fragment can be divided into three types (Type I, Type II, and Type III). Semantic similarities are referred as type IV clones. In the following subsections, we present the review of available literature on clone detection based on these four types.

A.1.1. Type I (exact clones) Clone Detection

Type I clones refer to identical code fragments except with possible variations at the levels of comments, layout, and whitespaces. Almost all the clone detection techniques address type I clone. However, state-of-the-art work

dealing with detecting exact match is by Johnson [23, 24]. It detects exact repetition of the text using Karp-Rabin Fingerprinting algorithm. He applied different transformations (such as removal of white space characters, removal of comments, or replacement of each identifier by an identifier marker) on the source code to remove uninterested text from it. Resulted code is then divided into substring such that each character of the code appears in at least one of the substring. Thereafter, matching substrings are identified. The author also addressed the concept of near-miss clones, but the problem is with the transformations he used. The transformations used produce too much false positives.

A.1.2. Type II (parameterized/named) Clone Detection

Type II clones refer to exactly similar code fragments except with possible variations in user-defined identifiers, literals, layout, types, and comments. Many state-of-the-art works deal with detection of type II clones. A tabular comparison of selected type II clone detection techniques is presented at the end of this subsection (Table 8). In some cases, '?' symbol is used to represent unsurety about the entry.

One of the leading works for type II clone detection in early 1990s was by Baker [25]. He reported a tool named *Dup* that represents source code as a sequence of lines and detects code clones on line-by-line basis. However, it cannot detect code clones written in different coding styles, having different variable names. During the same period, another metric-based approach for detecting type II clones was proposed by Buss et al. [26]. They transformed source code into tuples representing their complexity values. After that, they

used Euclidean distance to measure similarity between code fragments. In the same year, Mayrand et al. [111] also worked for detecting duplicate or near duplicate functions in a large software systems. They also used metric-based technique to identify duplicated functions. They used 21 function metrics and grouped them into four points of comparisons—name, layout, expressions, and control flow. Then, they identified the cloning between functions by comparing these metrics.

A clone detection technique using abstract syntax suffix trees was proposed by Koschke et al. [28]. The proposed approach finds clones in linear time and space. Initially, the program is parsed and an abstract syntax tree (AST) is generated. The generated AST is then used to generate serialized AST. Thereafter, suffix-tree based detection is used for detecting identical clone fragments. The authors also addressed that their tool can detect type-III clones using Baker's technique [177].

A token-based efficient clone detection algorithm that significantly reduces the memory usage was proposed by Basit et al. [29]. In the first step, the code is transformed into a token-sequence. Then, suffix-array based data structure and a straight forward variation of existing algorithm [178], which they called NERF (Non Extendible Repeat Finder), is used to locate repeated substrings in the token-sequence.

An incremental clone detector was proposed by Göde and Koschke [30] in 2009. It detects clones based on the results of the previous revision's analysis. They transformed each of source files into token-sequences and stored them in a token table. After that, a generalized suffix tree (GST) is created from the token-sequences. Bakers's algorithm pdup [25] is then used to find clones

from the generated GST. Unfortunately, GSTs require substantially more memory than read-only suffix-trees. It is because GSTs need extra links for traversing during the update operations. It makes the approach difficult to scale for larger systems. An attempt to make scalable and incremental clone detection tool was made by Hummel el al. [31]. They used token-based representation of the source code which is further normalized and grouped into statements. The authors adapted their own tool ConQAT [179], and ran it in a pipeline fashion to make the proposed approach highly scalable and incremental.

A formal-method based clone detection approach for Java was proposed by Cuomo et al. [33]. Authors claimed it to be the first formal-method based approach for detecting source code clones. They first compiled the source code, and then analyzed the resulting Java bytecode for detecting code clones. In the first step, authors transformed the complied Java code into Calculus of Communicating Systems (CCS) specifications. Then, they used equivalence relations to determine cloned code fragments. According to the authors, based on the types of clones to be detected, different types of equivalence relations can be considered. Here, the authors used weak equivalence in their implementation. They used Concurrency Workbench of New Century (CWB-NC), one of the most popular formal verification tools, to check the weak equivalence.

Toomy [37] designed a tool named *ctcompare* which is able to detect code clones in large software systems using hashed token-sequences. In the first step, they lexically analyzed the source code to produce sequences of tokens. Then, sequences of tokens are broken into overlapping tuples. These tuples are

further hashed. Hashed values are then used to identify type I and type II clone pairs.

A parallel and efficient approach for clone detection was recently proposed by Sajnani and Lopes [36, 39]. They proposed a technique that horizontally scales clone detection across multiple machines using MapReduce framework. Koschke [34, 38] worked for an inter-system clone detection technique. In the proposed technique, once the code is tokenized, suffix tree is generated for either the subject program or the corpus, whichever is smaller. Then, every file of the corpus is compared with the suffix tree. The author used hashing to reduce the number of file comparisons.

Table 8. Summary of selected Type II clone detection techniques

Technique (Authors; *Tool*; Year)	**Internal Source Code Representation**	**Clone Granularity**	**Comparison Granularity**	**Comparison Technique**	**Evaluation/ Validation**	**Language Paradigm**	**Scalability**
Baker [25, 27]; *Dup;* 1993, 1995	Strings of symbols over an alphabet of integers (tokens)	Free	Line	Suffix-tree based algorithm	X Window System (0.7M lines); SS from production system (1.1M lines)	C, C++, Java	Yes
Buss et al. [26]; 1994	Set of tuples	Free	Fragments of code represented by tuples	Metric-based technique	Information not provided	Information not provided	Information not provided
Mayrand et al. [111]; 1996	AST further transformed to their own intermediate representation language	Fixed (function clone)	Functions represented by set of 21 metrics	Metric-based technique	Two telecommunication monitoring systems (total 1M lines)	C, C++	Yes
Koschke et al. [28]; 2006	Serialized abstract syntax trees	Fixed (?)	Subtree	Suffix-tree based comparison technique	Wget 1.5.3 (16K); Bison 1.32 (19K); SNNS 4.2 (105K); PostgreSQL 7.2 (235K)	C	Yes
Basit et al.[29]; 2007	Tokens	Free	Sequence of tokens	Suffix-array based LCP match	On Linux Part (3025K LOC)	C (later extended to C++, Java)	Yes

Technique (Authors; *Tool*; Year)	Internal Source Code Representation	Clone Granularity	Comparison Granularity	Comparison Technique	Evaluation/ Validation	Language Paradigm	Scalability
Göde and Koschke [30]; 2009	Tokens	Free (?)	Subtree of GST	Modified version of Baker's pdup [25]	Ant (125K); Apache (145K); ArgoUML (130K); gcc(1.2M); gimp (.6M)	C, Java	Incremental but not scalable to large systems
Hummel el al. [31]; 2010	Tokens	Free (?)	Substring	Adapted version of ConQAT [179]	Jabref 2.6 (115K); One commercial software (460K); Linux Kernel 2.6.33.1 (11M)	Java, ABAP, C	Very highly scalable
Cuomo et al.[33]; 2011, 2012	CCS specifications generated from Java bytecode	Fixed	Corresponding CCS specification of source code	Weak equivalency verified using CWB-NC	Two small Java projects (3K and 8K)	Java	Low
Toomy [37]; *Ctcompare*; 2012	Tokens	Free	Substring	Hashing	Unix data set (13.2M)	C	Highly scalable
Sajnani and Lopes [36, 39]; 2012, 2013	Token key - value pair	?	Code block	Hash-based	700 open source Java projects (23M)	Java	Yes
Koschke [34, 38]; 2012, 2013	Tokens	?	Subtree	Hashed suffix-tree based comparison	Ubuntu 10.04 source corpus; Gnome; IJaDataset	C, Java	Highly scalable

A.1.3. Type III (gapped/near-miss) Clone Detection

Type III clones are fragments of duplicated code modified further by addition, deletion, and/or modification of statement(s). The literature available on Type III clones is discussed below. Table 9 at the end of this subsection gives summary of selected type III clone detection techniques.

An early attempt to detect type III clones was by Davey et al. [40]. They used neural networks to find similar blocks of source code. In their first attempt, they used self-organizing neural networks (a Self Organizing Map (SOM)) to cluster feature vectors associated with the procedures. But, the limitations with

the approach were long training time and fixed number of clone classes that were created. They updated their system with dynamic competitive learning (DCL) networks to overcome the above limitations.

In 1998, Baxter et al. [41] proposed an AST-based clone detection technique. In the first step, they parsed the source code to generate an AST. Then, they used tree-based matching to detect exact and near miss clones from the generated AST. They further extended their tool as CloneDR [125] which supports many languages such as Ada, C#, PHP, Python, VB, Fortran, PLSQL, and XML.

One of the pioneer tool for detecting type III clones is CCFinder [42] developed by Kamiya et al. in 2002. CCFinder uses a token-based technique that first converts the source code into a token-sequence. The authors further defined their own set of language specific transformation rules. Based on these transformation rules, the token-sequence is updated, i.e., some tokens are added, removed, or changed. Then, suffix-tree based sub-string matching algorithm is used to find clone pairs/clone classes on the transformed token-sequence.

Another efficient clone detection tool, SDD (Similar Data Detection) was devised by Lee and Jeong [44] in 2005. This tool uses indexes and inverted-indexes of code fragments and their positions to detect code clones. An index has information about position and the corresponding chunk. On the other hand, an inverted-index includes chunks and the corresponding positions. SDD uses an n-neighbor distance algorithm to detect similar fragments in the source code.

CP-Miner [12, 43] is another state-of-the-art token-based tool for detecting type III clones. It uses a frequent subsequence mining algorithm, CloSpan (Closed Sequential Pattern Mining) [180] to identify similar sequences of tokenized tokens.

Jiang et al. [46] devised an algorithm for detecting similar trees and applied it on the tree representation of the source code to detect similar code fragments. They generated characteristics vectors to approximate the structural information within ASTs in the Euclidean space, and then used locality sensitive hashing to efficiently cluster similar vectors and thus code. In the first step, a parser translates the source files into parse trees which are then used to produce fixed-dimension vectors. These vectors are then clustered with respect to their Euclidean distances.

Anti-unification was used by Bulychev and Minea [47] to detect code clones. In the first step, source files of the program are converted into an XML representation of their ASTs. Then, anti-unification is applied to group similar ASTs into equivalence classes called clusters. Each cluster corresponds to a clone class.

The first tool for code clone detection that took full advantages of multi-core processors was YACCA, proposed by Livieri and Inoue [48] in 2008. It leverages multi-code processors by evenly distributing the total workload between the cores. It uses a parameterized detection algorithm to detect similar code fragments. Also, it is language independent and to some extent can be used to detect cross-language code clones.

Jia et al. [51] proposed an efficient and precise clone detection tool called KClone. It uses combination of lexical and local dependence analysis to achieve precision, while retaining speed. It uses two-step approach for detecting code clones. In the first step, fast lexical analyzer is used to detect basic clone-pairs, i.e., type I and type II clones. Using the detected basic clone-pairs information, second step detects type III clone-pairs from the source code.

Roy and Cordy [49, 52] proposed a multi-pass hybrid clone detection technique for detecting type III clones. They implemented the technique in the form of a tool called NiCad [56]. NiCad can be either used in command-line mode or it can be easily embedded in IDEs and other environments.

A hybrid (syntactic and textual) approach for clone detection was proposed by Funaro et al. [54]. They combined AST representation of the source code with the string (text-based) representation of the source code. AST representation of the code helped the authors to retrieve structural similarities, while the string representation helped in refining the results through direct comparison. In the first step, the authors transformed the source code into ASTs which are further converted to a forest of ASTs. The forest is then serialized by encoding into a string representation using an invertible mapping function. This string representation is then searched for repeated substrings. The mapping function is used again to decode these substrings into sub-trees. To generate the corresponding code fragments, the reconstructed sub-trees are then matched back to their original ASTs.

In 2011, Higo et al. [58] came up with new idea of clone detection. They combined incremental clone detection with PDG (program dependency graph)

based representation of the source code. They defined some metrics/functions to compare the similarity between two nodes of PDGs. The advantage of their approach is that it can detect non-continuous code clones more effectively than other incremental clone detection techniques. In the same year, Higo proposed another tool Scorpio [59] with his colleague Kusumoto, where they applied two-way slicing to detect clones. Since using only forward or backward slicing does not detect all similar sub-graphs. Using two-way slicing overcomes this problem. Scorpio is also a PDG-based tool that uses hashing at the node-level. Nodes with same hashed values correspond to cloned segments.

Another state-of-the-art tool for clone detection is CMCD (Count Matrix Clone Detection) [60] proposed by Yang and Yao. It uses a count matrix to represent characteristics of a particular code segment. A count matrix consists of n count vectors where each vector corresponds to a variable in the code segment. A count vector records various features such as number of time the variable is defined, used, or called. These count matrices are used to find similarity between code segments using bipartite graph matching algorithm.

Microsoft Research Asia team designed an efficient, scalable, and parallelizable clone detector, XIAO [61]. Microsoft has already integrated the XIAO with Microsoft Visual Studio 2012. Among other, one of the main features of XIAO is its compatibility. By default, it provides supports for C, C#, and C++; but it also allows the users to plug-in their own parsers into the system to support other languages.

A metric-based data mining approach for detecting function clones was presented by Salwa and Hafiz [98]. In the first step, fragments and corresponding metrics are extracted from the source code at the function level.

Then, the software is partitioned into three types of clusters—primary, intermediate, and single—using data mining clustering algorithm. All type I and type II clones are included in primary cluster, whereas intermediate cluster corresponds to type III clones.

Murakami et al. [62] devised a new clone detection algorithm which is free from the influence of presence of repeated instructions in the software. They transformed every repeated instruction present in the source code into a special form, and then applied a suffix-array based algorithm to find repeated code segments. In the first step, token-sequences are created from the source files of the code. Then, hash values are computed from these token-sequences. In the next step, repeated instructions are removed from the hash-sequence. Thereafter, identical subsequences are identified using a suffix-array based algorithm.

An accurate and scalable token-based clone detection tool was by Yuan and Guo [63]. However, it differs from the other token-based clone detection techniques in the sense that it does not use token-sequences while comparing the code segments. Instead, it represents the code segments using count matrices. Then, cosine similarity and proportional similarity functions are used for similarity measures.

Some of the recent advancements in clone detection research are SimCad [66], CDSW [67]. SimCad is highly scalable and fast clone detection algorithm. It uses multi-level index based searching to speed up the clone detection process. It detects clones as code fragments (e.g., function or code block), the boundary of which are predefined during the source code pre-processing step. CDSW detects gapped code clones using Smith-Waterman algorithm [181]. Smith-

Waterman algorithm is an algorithm for identifying similar alignments between two base sequences. In the first step, CDSW transform the source files into token-sequences. Next step calculates the hash value for each statement. At the end of this step, each source file is transformed into one hash-sequence. Similar hash sub-sequences are then identified from hash-sequence using modified version of the Smith-Waterman algorithm.

Qu et al. [68] combined spatial space analysis with graph-based mining to find code clones from software systems. They used PDGs for source code representation. First, they applied spatial pattern search on the PDGs, and then used graph-based pattern mining algorithm to find out candidate code clones. They further used false positive pruning and pattern composition techniques to improve the detection results.

Table 9. Summary of selected Type III clone detection techniques

Technique (Authors; *Tool*; Year)	**Internal Source Code Represen-tation**	**Clone Granular-ity**	**Comparison Granularity**	**Comparison Technique**	**Evaluation/ Validation**	**Language Paradigm**	**Scalability**
Davey et al. [40]; 1995	Set of fixed length feature vectors	Fixed (procedure)	Feature vector of module unit	Neural networks based	5MB of arbitrary selected source code	Informatio n not available	*SOM* based*:* No; *DCL* based*:* better than *first*
Baxter et al. [41]; 1998	AST	Free	Sub-tree	AST-based tree matching	Process Control System (400K C code)	C, C++, Java, COBOL, and others	No
Kamiya et al. [42]; *CCFinder*; 2002	Sequence of transforme d tokens	Free	Token sub-sequences	Suffix-tree matching algorithm	On several systems:- FreeBSD; NetBSD,; Linux and others	C, C++, Java, COBOL, and others	Highly scalable; among the renowned tools
Li et al. [12, 43]; *CP-Miner*; 2004, 2006	Sequence of numbers (tokens)	Free	Token sub-sequence	Frequent Sub-sequence Mining	Linux; FreeBSD; Apache and others	C, C++	Highly scalable; among renowned tools

Technique (Authors; *Tool*; Year)	**Internal Source Code Represen-tation**	**Clone Granular-ity**	**Comparison Granularity**	**Comparison Technique**	**Evaluation/ Validation**	**Language Paradigm**	**Scalability**
Lee and Jeong [44]; *SDD*; 2005	Strings of code (text)	Free	Substring (multi-word)	Based on n-neighbor indexed algorithm	JDK 1.5 (1M); httpd (84K); tuby 1.8.2 (.2M) and others	Language independen t	Very scalable
Jiang et al. [46]; *DECKARD* ; 2007	Parse trees normalized to vectors	Free	Fixed length characteristic s vectors	Locality Sensitive Hashing	Linux kernel; JDK	Any language with a formally specified grammar	Yes
Evans et al. [45, 50]; *Asta*; 2007, 2009	ASTs	Fixed (function clone)	AST nodes	Graph theoretic approach on associative array	Netbean-javadoc; eclipse-ant; eclipse-jdtcore;	Java, C#	Yes (scalability addressed in [50]; 2009)
Bulychev and Minea [47]; *Clone Digger*; 2008	ASTs	?	Subtree	Anti-unification	BioPython project and NLTK project	Python, Java	?
Livieri and Inoue [48]; *YACCA*; 2008	Token-sequences	Free	Token sub-sequence	Repeating substrings detection	Information not provided	Language independen t	Highly scalable (multi-core)
Roy and Cordy [49]; *NICAD*; 2008	Program text	Fixed (function and block)	Line	Longest Common Subsequence s algorithm	More than 20 open source systems	C, Java, C#, Python, WSDL [56]	Yes
Funaro et al. [54]; *SynTex;* 2010	ASTs further normalized to strings	Free	Substring	Searching common substring	Evaluated on Bellon's benchmark [101]	Java (?)	?
Corazza et al. [53]; 2010	ASTs	Fixed	Subtree of fixed granularity (claas-tree, method-tree, statement tree)	Based on Tree Kernel function	Federico II (an academic Java system)	Java	Very low
Higo et al. [58]; 2011	PDG	?	PDGs' edge tree	Based on equivalency among edge trees	Ant	Java	Highly scalable; an incremental clone detection technique addressing type III clones

Technique (Authors; *Tool*; Year)	**Internal Source Code Represen-tation**	**Clone Granular-ity**	**Comparison Granularity**	**Comparison Technique**	**Evaluation/ Validation**	**Language Paradigm**	**Scalability**
Higo and Kusumoto [59]; *Scorpio*; 2011	PDG	Free	PDGs' nodes	Program slicing	Eclipse Ant; NetBeans Javadoc; Eclipse – jdtcore; j2sdk1.4.0-javax-swing	Java	Among most scalable tools
Yuan and Guo [60]; *CMCD*; 2011	Count matrix for each method	Fixed (method)	Count matrix	Bipartite graph matching algorithm	JDK 1.6 and other small student projects	Language independen t	Medium
Dang et al.[57, 61]; *XIAO*; 2011, 2012	Tokens	Fixed (?)	Subsequence of tokens	Metric based	Industrial validation by Microsoft Engineers	C, C++, C#	Yes, incorporate d in MS Visual Studio
Salwa and Hafiz [98]; 2012	Text	Fixed (function clones)	Code fragments	Fractal Clustering	Weltab; SNNS	C	Low
Yuan and Guo [63]; *Boreas*; 2012	Tokens	Free	Count matrixes	Using Cosine Similarity Function and Proportional Similarity Function	Java SE Developme nt Kit 7 (2.2M); Linux kernel 2.6.38.6 (10M)	Java, C, C++	Highly scalable with more precision
Zibran and Roy [64]; 2012	AST	Fixed	Preprocessed suffix tree	Suffix-tree based k-difference hybrid algorithm.	Weltab (10K); PostGreSQL (154K)	C, C#, and Java	Medium
Murakami et al. [62]; *FRISC*; 2012	Token sequence	Free	Statements	Suffix-array based algorithm	Netbeans, ant, jdtcore, weltab, cook etc.; evaluated on Bellon's benchmarks	Java and C	Yes, supports multi-threading
Muddu et al. [65]; *CPDP*; 2013	Tokens	Free	Block of statements	Karp-Rabin Greedy String Tiling algorithm	25 open source Java projects from github	Java	high
Murakami et al. [67]; *CDSW*; 2013	Tokens	Free	Statement level	Hashing	Netbeans, ant, jdtcore, weltab, cook etc.; evaluated on Bellon's benchmarks	Java and C	Scalable than AST or PDG based approaches

Technique (Authors; *Tool*; Year)	Internal Source Code Represen-tation	Clone Granular-ity	Comparison Granularity	Comparison Technique	Evaluation/ Validation	Language Paradigm	Scalability
Qu et al. [68]; 2014	PDG further transforme d to graphic sequence in sequential space	?	Encoded graphic subsequence s	Uses spatial space analysis and then graph-based pattern mining	On a health care software system (near 1.1M)	?	High

A.1.4. Type IV (semantic and re-ordered) Clone Detection

When two code fragments have functional similarities, they are termed as type IV clones. In these types of clones, it is not necessary for the cloned fragments to be copy of each others. Table 10 compares selected type IV clone detection techniques.

In 1990, Horwitz [182] published a key article on the detection of semantic and textual differences between two versions of the program. However, a successful attempt to detect type IV clones was by Komondoor and Horwitz [69] in 2001. The approach was based on using program slicing to detect isomorphic sub-graphs of a PDG. They first created PDGs for each procedure, and then used backward and forward slicing to detect code clone from the subject program. In the same year, another attempt to detect type IV clones was by Krinke [70]. It is also a PDG-based clone detection approach that uses k-length patch matching to find out maximal induced sub-graphs.

A text-based semantic clone detection approach was proposed by Marcus and Maletic [99] in 2001. They used latent semantic indexing to find semantic similarities between different program entities. But when comments do not

exist in the code and names of identifiers in the corresponding entities are completely different, the approach fails to detect clones.

An early attempt to make efficient and scalable type IV clone detection technique was by Gebel et al. [71] in 2008. The authors adapted the approach used by Deckard clone detection tool [46], making it more efficient and scalable. They started with the PDG-based representation of the source code. Then, selected PDGs are mapped to an AST forest. After that, standard tree-based detection algorithm, Deckard, is modified and adapted to locate cloned code in the software.

Comparison-based clone detector, MeCC [73] detects semantic clones by comparing programs' abstract memory states. These abstract memory states are estimated at each procedure exit-points using path-sensitive semantic-based static analysis.

Semantic web reasoners were used by Schugerl [74] for detecting semantic clones in 2011. The proposed approach uses Hadoop map-reduce framework to scale the detection process. He used description logics to model source code, and then applied semantic web reasoners to find similar code fragments.

A two-stage clustering technique was used for detecting semantic clones by Yoshioka el al. [75]. In the first step, code fragments are extracted from the source code. Then, these extracted code fragments are classified into clusters based on their characteristics. This step is divided into two stages. In the first stage, code fragments are coarsely classified in order to obtain good enough result in a short time. In the second stage, the results of the first stage are finely classified to obtain more precise clusters. In the last step, these resultant

clusters are converted into a collection of clone sets (i.e., sets of semantically similar code fragments).

Li and Ernst developed a tool CBCD (Cloned Buggy Code Detector) [78], which when given an example of buggy code, searches the subject program for code fragments which are semantically similar to this buggy code. The authors made a claim of presence of duplicated buggy codes in software systems by performing empirical studies. They proposed their own tool to detect the presence of these buggy cloned codes in software systems. For detecting cloned buggy codes, they first transformed the subject code and the buggy code both into PDGs using CodeSurfer. Then, they adapted the igraph's implementation of sub-graph isomorphism matching to detect similar codes.

JSCTracker [76, 77] detects semantic clones in Java methods using methods' IOE (Input, Output, and Effects) behaviors. IOE behavior includes return values of methods as well as their effects on the pre-states and post-states of the heap. In the first step of the algorithm, they generated decorated AST from the source code of the subject program. Then, two-step filtering based on the syntactic and semantic information encoded in the AST is used to find out candidate clones. In the last step, these candidate clones are tested and collected into equivalence clone classes.

Table 10. Summary of selected Type IV clone detection techniques

Technique (Authors; *Tool*; Year)	**Internal Code Represen-tation**	**Clone Granular-ity**	**Comparison Granularity**	**Comparison Technique**	**Evaluation/ Validation**	**Lan-guage Para-digm**	**Scalabil-ity**
Komondoor and Horwitz [69]; 2001	PDG	Fixed (procedure?)	PDG subgraphs	Using Program slicing	Three Unix utilities; some IBM project code	C	No

Technique (Authors; *Tool*; Year)	**Internal Code Represen-tation**	**Clone Granular-ity**	**Comparison Granularity**	**Comparison Technique**	**Evaluation/ Validation**	**Lan-guage Para-digm**	**Scalabil-ity**
Krinke [70]; *Duplix*; 2001	PDG	Free	PDG subgraphs	*k*-length patch matching	Author's selected test programs	C	No
Marcus and Maletic [99]; 2001	Text	Fixed (function, file, program)	Sentences	Latent Semantic Indexing approach	Mosaic v2.7 (95K)	C	?
Gabel et al.[71]; 2008	PDGs further transforme d to ASTs	Free	Characteristi c vectors	Adaptation from *Deckard* [46]	GIMP; GTK+; MySQL; PostgreSQL ; Linux kernel	C, C++	Highly scalable
Jiang and Su [72]; *EQMINER*; 2009	Program text represented in some intermediat e form	Free	Code fragments represented in intermediate forms	Clustering based on Representativ e Based Partitioning (devised by authors itself)	Linux kernel 2.6.24	C	Scalable but very slow
Kim et al. [73]; *MeCC*; 2011	Abstract memory states	Fixed (procedural level)	Programs' abstract memory states	Abstract memory state comparison	Python; Apache; PostgreSQL	C	Partial scalable
Schugerl [74]; *DL-Clone*; 2011	Description Logic (DL) model further transforme d to concepts and relations (using OWL)	Fixed (?)	OWL representatio ns of blocks	Semantic web reasoners	JDK 1.5 (randomly selected 620 files)	Java	High, can be paralleliz ed
Yoshioka et al. [75]; *Takana*; 2011	Text based (represente d as feature vectors)	Free	Set of feature vectors	Feature clustering	eclipse-ant; eclipse-jdtcore; j2sdk.1.4.0-swing; jdk 1.4.2, jdk1.5.0	Java	Yes
Li and Ernst [78]; *CBCD*; 2012	PDGs	Free	Subgraphs of PDGs	Subgraph isomorphism matching using igraph	Git; Linux kernel; PostgreSQL	C, C++	Yes
Elva and Leavens [76, 77]; *JSCTracker* ; 2012	Decorated ASTs	Fixed (method level)	MethodType and MethodEffec t information encoded in AST	Equivalence class having same MethodType and MethodEffect information	DSpace; JabRef	Java	No

A.2. High-level Clone Detection

The researchers also proposed other types of software clones possible at code level. This section provides brief overview of such clone detection techniques.

A.2.1. Structural Clone Detection

Basit and Jarzabek [5, 81] proposed an algorithm for detecting design-level similarities such as similar methods, files, or directories calling them structural clones. They first classified the code clones into two types—simple clones and structural clones. Simple clones refer to fragments of duplicated contiguous code (so called type I and type II clones). Then, they used these cloned contiguous code fragments (i.e., simple clones) to find out structural clones. They defined structural clones as the "patterns of inter-related classes emerging from design and analysis spaces; patterns of components at the architecture level; design solutions repeatedly applied by programmers to solve similar problems".

The approach is as follows: In the first step, simple clones are detected from the subject program by applying token-based technique [29]. Then, recurring patterns of simple clones are detected using Frequent Closed Itemset Mining (FCIM) algorithm. The output of this step is a list of clone patterns occurring frequently in the subject program. For each such clone pattern, the subject program is searched for files having that clone pattern. In the last step, these searched files are clustered into similarity groups using two metrics: file percentage coverage and file token coverage. Each similarity group corresponds to a structural clone. They have implemented the approach as a

tool, Clone Miner and experimentally confirmed that the tool can find many useful higher-level design similarities, and is scalable to big programs.

A.2.2. Logical Clone Detection

Logical clones were proposed by Qian et al. [82] in 2013. They called them as code clones "revealing high-level business and programming rules". Logical clones involve similar or relevant concerns and topics but are not necessarily similar at the code-level.

The authors used semantic clustering and graph mining techniques for detecting logical clones. In the first step, they represented the source code as a meta-model consisting of methods, entity classes, and persistent data objects along with the invoke/access relations among them. Given this initial program meta-model, they clustered similar methods (i.e., methods sharing similar topics) into functional clusters using semantic clustering. They used Simian [183], a text-based clone detector, for detecting clones in the code. In the last step, based on the program-model produced by model extraction step, sub-graph pattern mining algorithm is used for detecting logical clones.

A.2.3. Other High-level Clone Detection

There are some works that deal with detection of design patterns [83-94] and API usage patterns [95, 96]. Among few initial works, an attempt to detect design patterns (template methods, factory methods, and bridge) in C++ systems was by Keller et al. [83]. Smith and Stotts [84] presented a tool, they called System for Pattern Query and Recognition (SPQR), that detects a suite of elemental design patterns.

Tsantalis et al. [87] presented a solution to design pattern detection problem that uses similarity score between design patterns and graph representation of the program to detect occurrences of the design patterns. Romano et al. [88] applied text clustering on classes of a system, and then used existing techniques and tools, DPR [89] and Pattern4 [87], on the clusters to identify design-pattern instances. Semantic web was used for detecting design patterns in Java source code [91] in 2012. In the same year, Tekin et al. [92] proposed a sub-graph mining based design pattern detection algorithm for object-oriented software systems.

Yu et al. [94] in 2013 presented an approach for detecting design patterns (in particular Decorators) using graph isomorphism. In their another work [93], they worked for detecting instances of structural design patterns from source codes. They first transformed the source code into Class-Relationship directed graphs. In the next step, they identified instances of sub-patterns that would be the possible constituents of pattern instances based on sub-graph isomorphism.

A.3. Cloning Beyond Code

Most of the software clone detection research mainly focuses on code clones. However, cloning also occurs in other software artifacts such as UML models, use cases, test cases, spreadsheets, and compiled code (e.g., Javabyte code). This section summarizes some of the existing works in this direction. Table 11 provides summary of selected works.

A.3.1. Model Clone Detection

Cloning has been found in specification models such as UMLs [184-187] as well as in code generation models such as Matlab and Simulinks [188-192].

Liu et al. [184] detected clones in UML sequence diagrams using a suffix-tree based algorithm. In the first step, two-dimensional sequence diagrams are converted into one-dimensional arrays. In the next step, suffix trees are constructed from the one-dimensional arrays. The constructed suffix trees are then used to find longest common prefixes, which correspond to clone candidates in the sequence diagrams. Another attempt for detecting clones in UML models was by Störrle [185, 187]. He proposed a tool MQ_{lone} as a prototype for his approach. The technique was based on model querying.

An approach for detecting clones in data flow models was proposed by Deissenboeck et al. [188] in 2008. They used a graph-based detection approach consisting of three steps. The first step is preprocessing, which converts the model into labeled graph-model. Labels are assigned to nodes using some normalization function. Then, clone pairs are extracted from the normalized graph-model using some heuristics in a breath-first search manner. The similarity function uses maximum weighted bipartite matching to find similar nodes. In the last step, clones pairs are clustered to find out sub-structures that are used more than twice in the model.

Another efficient tool for detecting clones in Matlab and Simulink models is ModelCD [189], which detects both exact and approximate matches of model clones. It uses two novel graph-based clone detection algorithms–eScan and aScan, enabling systematic and incremental discovery of model clones. In the first step of the approach, a sparse labeled digraph is generated from the model. Then, eScan algorithm detects exact matches using canonical labeling. Finally, aScan algorithm is applied to find approximate matches.

An index-based incremental and distributable technique to detect model clones is by Hummel et al. [190]. In the first step, a directed labeled multi-graph is generated from the model. Isomorphic sub-graphs in the multi-graph correspond to clone classes in the model.

Some of the recent attempts for detecting model clones are [186, 191, 192]. Antony et al. [186] proposed an algorithm for detecting clones in UML behavioral models in 2013. They adapted NiCad for detecting clones in behavioral models. In the first step, they transformed the XMI-file representation of the behavioral models into TXL [193] source transformation language. After normalizing TXL representation, NiCad is used to detect clones. Alalfi [191, 192] built a tool SIMONE to detect structurally meaningful type III (i.e., near-miss) clones in Simulink models.

A.3.2. Data Clone Detection

Clone detection has been applied to spreadsheets by Hermans et al. [194]. They adapted an existing text-based clone detection algorithm [23], and devised an algorithm for detecting clones in spreadsheets. They called the detected clones as data clones. They used cell values as fingerprints, and removed values that do not occur as formula and plain text. Subsequently, values that occur in multiple places are grouped into clone clusters to detect groups of cells that are possibly copied. In order to visualize data clones, authors generated dataflow diagrams showing the relationship between worksheets that contains clones, and added pop-ups to both parts of a clone indicating the source and the copied side of the clone.

A.3.3. Detection of Clones in Requirements Specification

Juergens et al. [195] analyzed 28 requirements specification documents written in natural language (English and German) with total of more than 8,500 pages. In the first step, they converted the source documents into plain text. Then, they applied ConQAT [179] on the plain text to find repeated substrings.

Table 11. Summary of clone (software artifacts other than code) detection techniques

Technique (Authors; *Tool*; Year)	**Types of Clones Detected**	**Internal Code Representation**	**Comparison Granularity**	**Comparison Technique**	**Evaluation /Validation**	**Language Paradigm/ Scope**	**Remarks**
Liu et al. [184]; *Duplication Detector*; 2006	Model Clones	one-dimensional array further transformed to suffix tree	Subtree of suffix tree	Suffix-tree based comparison	Two industrial projects	UML sequence diagrams	High precision and recall
Deissenboeck et al. [188]; 2008	Model Clones	Labeled model graph	Blocks/node of graph	Maximum weighted bipartite matching	Two industrial models	Matlab/ Simulink	Scalable in nature
Pham et al. [189]; *ModelCD*; 2009	Model Clones	Sparse, labeled directed graph	Subgraph	Canonical labeling followed by vector based counting approach	On four open source models	Matlab/ Simulink	Scalability Medium; precision low
Störrle [185, 187]; MQ_{lone}; 2010, 2013	Model Clones	XMI files further transformed to prolog code	?	Model matching and similarity	Library Management System	UML Domain models	Scalability (?)
Juergens et al. [195]; 2010	Cloned Requirement Specifications	Plain text	Text string	Common substring matching	28 requirement specifications	Requirement Specifications	Scalability (?)
Hummel et al. [190]; 2011	Model Clones	Directed, labeled multigraph	Subgraph	Clone index based hashing	SIM; MUL; SEM; ECW; MPC	Matlab/ Simulink	Incremental and scalable
Alalfi et al. [191, 192]; *SIMONE;* 2012	Model Clones	Normalized text form	Normalized text groups	Using NiCaD [49]	Some publicly available Simulink models	Simulink Models	Author claim scalability but not evaluated on large systems

Technique (Authors; *Tool*; Year)	Types of Clones Detected	Internal Code Representation	Comparison Granularity	Comparison Technique	Evaluation /Validation	Language Paradigm/ Scope	Remarks
Antony et al. [186]; 2013	Model Clones	TXL representation	Text group	Using NiCaD [49]	On four reverse-engineered models	UML behavioral model	Yes
Hermans et al. [194]; 2013	Data Clones (exact and near miss)	Text based cell represented as lookup table	Cell in cluster forms	Using cluster finding and matching; based on [23]	Evaluated on EUSES spreadsheet corpus [196]; further case study with Delft university and Foodbank	Spreadsheets	High precision and scalable

A.4. Other Possible Directions

Much work has been done for code clone detection. We also found the cases claiming that cloning is possible beyond the code level (Section A.3). We already discussed some of these cases in the above subsections. We also found some other works citing cloning in test cases [197], Java byte code [198, 199], and websites [200]. Juergens [176] suggests cloning to be possible in other software artifacts too. Some of the possibilities are for cloning in the feature models, schemas, system architectures, process models, configuration files, etc.

A.5. Chronology of Clone Detection Techniques

Table 12 provides chronology of clone detection techniques.

Table 12. Chronology of clone detection techniques

NOMENCLATURE: **Low-level Clones:** **T1**: Type I Clone; **T2**: Type II Clone; **T3**: Type III Clone; **T4**: Type IV Clone; **High-level Clones**: **MFC**: Method/Function Clone; **CC**: Class Clone; **FC**: File Clone; **LC**: Logical Clone; **OHC**: Other High-level Clones; **CP**: Collaborative Pattern; **Beyond Code Clones:** **MC**: Model Clone; **DC:** Data Clone; **OT:** Other types of Clone.

Detection Technique (Authors; Tool; Year)	Code Clones										Cloning Beyond Code		
	Low-level Clones				High-level Clones								
	Textual Similarity			T4	Structural Clone			LC	OHC	CP	MC	DC	OT
	T1	T2	T3		MFC	CC	FC						
Johnson [23, 24]; 1993, 1994	Y												
Baker [25, 27]; *Dup*; 1993, 1995	Y	Y											
Buss et al. [26]; 1994	Y	Y											
Davey et al. [40]; 1995	Y	Y	Y										
Mayrand et al. [111]; 1996	Y	Y			Y								
Baxter et al. [41]; 1998	Y	Y	Y										
Keller et al. [83]; 1999									Y				
Komondoor and Horwitz [69]; 2001	Y	Y	Y	Y									
Krinke [70]; *Duplix*; 2001	Y	Y	Y	Y									
Marcus and Maletic [99]; 2001	?	?	?	Y					Y				
Kamiya et al. [42]; 2002	Y	Y	Y				Y						
Heuzeroth et al. [85]; 2003									Y				
Smith and Stotts; [84]; *SPQR;* 2003									Y				
Li et al. [12, 43]; *CP-Miner*; 2004, 2006	Y	Y	Y										
Lee and Jeong [44]; *SDD*; 2005	Y	Y	Y										
Basit and Jarzabek [5, 81]; *Clone Miner*; 2005, 2009	Y	Y			Y		Y						
Koschke et al. [28]; 2006	Y	Y											
Liu et al. [184]; *DuplicationDetector*; 2006											Y		
Tsantalis et al. [87]; *Pattern4*; 2006									Y				
Basit et al. [29]; *Repeated Tokens Finder (RTF);* 2007	Y	Y											
Jiang et al. [46]; *DECKARD*; 2007	Y	Y	Y										

Detection Technique (Authors; Tool; Year)	Code Clones										Cloning Beyond Code		
	Low-level Clones				High-level Clones								
	Textual Similarity			T4	Structural Clone			LC	OHC	CP	MC	DC	OT
	T1	T2	T3		MFC	CC	FC						
Evans et al. [45, 50]; *Asta*; 2007, 2009	Y	Y	Y		Y								
Bulychev and Minea [47]; *Clone Digger*; 2008	Y	Y	Y										
Livieri and Inoue [48]; *YACCA*; 2008	Y	Y	Y										
Gabel et al. [71]; 2008				Y									
Deissenboeck et al. [188]; 2008											Y		
Roy and Cordy [49, 52, 55, 56, 110]; *NICAD*; 2008-11	Y	Y	Y		Y								
Göde and Koschke [30]; 2009	Y	Y											
Jia et al. [51]; *KClone*; 2009	Y	Y	Y										
Jiang and Su [72]; *EQMINER*; 2009				Y									
Pham et al. [189]; *ModelCD*; 2009											Y		
Hummel el al. [31]; 2010	Y	Y											
Funaro et al. [54]; *SynTex;* 2010	Y	Y	Y										
Corazza et al. [53]; 2010	Y	Y	Y										
Juergens et al. [195]; 2010													Y
Störrle [185, 187]; MQ_{lone}; 2010, 2013											Y		
Schugerl [74]; *DL-Clone*; 2011	Y	Y	Y	Y									
Higo et al. [58]; 2011	Y	Y	Y										
Yuan and Guo [60]; *CMCD*; 2011	Y	Y	Y		Y								
Higo and Kusumoto [59]; *Scorpio*; 2011	Y	Y	Y										
Kim et al. [73]; *MeCC*; 2011	Y	Y	Y	Y	Y								
Yoshioka et al. [75]; *Takana*; 2011	?	?	?	Y									
Hummel et al. [190]; 2011											Y		
Romano et al. [88]; 2011									Y				
Uddin et al. [96]; 2011									Y				
Cuomo et al. [32, 33]; 2011, 2012	Y	Y											
Dang et al.[57, 61]; *XIAO*; 2011, 2012	Y	Y	Y										

Detection Technique (Authors; Tool; Year)	Code Clones										Cloning Beyond Code		
	Low-level Clones				High-level Clones								
	Textual Similarity			T4	Structural Clone			LC	OHC	CP	MC	DC	OT
	T1	T2	T3		MFC	CC	FC						
Toomy [37]; *ctcompare*; 2012	Y	Y											
Lavoie and Merlo [35]; 2012	Y	Y											
Salwa and Hafiz [98]; 2012	Y	Y	Y		Y								
Yuan and Guo [63]; *Boreas*; 2012	Y	Y	Y										
Zibran and Roy [64]; 2012	Y	Y	Y										
Murakami et al. [62]; *FRISC*; 2012	Y	Y	Y										
Li and Ernst [78]; *CBCD*; 2012	Y	Y	Y	Y									
Elva and Leavens [76, 77]; *JSCTracker*; 2012				Y	Y								
Alalfi et al. [191, 192] ; SIMONE; 2012											Y		
Keivanloo et al. [198, 199]; *SeByte*; 2012													Y
Lebon and Tzerpos [201]; 2012									Y				
Tekin et al. [92]; 2012									Y				
Paydar and Kahani; [91]; 2012									Y				
Binun and Kniesel [90]; *DPJF*; 2012									Y				
Sajnani and Lopes [36, 39]; 2012, 2013	Y	Y											
Koschke [34, 38]; 2012, 2013	Y	Y											
Uddin et al. [66]; *SimCad*; 2013	Y	Y	Y										
Muddu et al. [65]; *CPDP*; 2013	Y	Y	Y	?									
Murakami et al. [67]; *CDSW*; 2013	Y	Y	Y										
Qian et al. [82]; *MiLoCo*; 2013								Y					
Antony et al. [186]; 2013											Y		
Hermans et al. [194]; 2013												Y	
Yu et al. [93, 94]; 2013									Y				
Qu et al. [68]; 2014	Y	Y	Y										
Kodhai and Kanmani [97]; 2014					Y								
Proposed Work; *CoPAD*; 2015										Y			

Appendix B.

GLOSSARY

False Negative: a clone, but not detected as a clone by the clone detector.

False Positive: not a clone, but detected as a clone by the clone detector.

Precision: percentage of reported clones which are genuine.

Program Entity: variable, statement, code fragment, function, class method, class, source file, directory, module, subsystem (last two are designated groups of files and/or directories).

Recall: percentage of genuine clones that are reported.

Simple Clone: small-granular (generally 4–6 lines of code) cloned contiguous code-fragments.

Software Clone: a recurring configuration of program entities or software artifacts inter-related in some meaningful way, and where similarity among corresponding entities in the clone-instances has been already established by means of some similarity metrics.

Structural Clone: recurring patterns of simple clones in a software system.

Subject Program: the source code under consideration from which we have to detect software clones.

True Negative: not a clone, also not detected as a clone by the clone detector.

True Positive: a clone, also detected as a clone by the clone detector.

Type I Clones: identical code fragments except with possible variations at the levels of comments, layout, and whitespaces.

Type II Clones: exactly similar code fragments except with possible variations in user-defined identifiers, literals, layout, types, and comments. All type I clones fit under this category.

Type III Clones: type II clones modified further by addition, deletion, and/or modification of statement(s).

Type IV Clones: functionally similar code fragments not necessarily to be syntactically similar.