DESIGNDIFF: Continuously Modeling Software Design Difference from Code Revisions

Xiao Wang
Stevens Institute of Technology
Hoboken, United States
xwang97@stevens.edu

Lu Xiao
Stevens Institute of Technology
Hoboken, United States
lxiao6@stevens.edu

Kaifeng Huang
Fudan University
Shanghai, China
kfhuang16@fudan.edu.cn

Bihuan Chen
Fudan University
Shanghai, China
chenbihuan@gmail.com

Yutong Zhao
Stevens Institute of Technology
Hoboken, United States
yzhao102@stevens.edu

Yang Liu
Nanyang Technological University
Singapore
yangliu@ntu.edu.sg

***Abstract*—The design structure of a system continuously evolves as the consequence of fast-paced code revisions. Agile techniques, such as continuous testing, ensures the function goals of a system with every code revision. However, there lacks an efficient approach that can continuously model the design difference resulting from every single code revision to facilitate comprehension and ensure the design quality. This paper contributes a novel design modeling approach, called *Design Differencing* (DESIGNDIFF), that models and visualizes the high-level design differences resulting from every code revision. This paper defines a complete and general set of 17 design change operators to capture the design difference from any code revision. We evaluated the potential of DESIGNDIFF in three aspects. First, a user study of 10 developers indicated that DESIGNDIFF can help practitioners to faster and better understand high-level design differences from real-life software commits. Second, DESIGNDIFF analyzed 14,832 real-life commits in five real-life projects: finding 4,189 commits altered the software design, 855 commits introduced and 337 commits eliminated design flaws. The latency between the flaw introduction and elimination is on average 2 months to 2 years! With an affordable performance overhead, DESIGNDIFF has great potential to benefit practitioners in more applications.**

***Index Terms*—software architecture, architecture flaws, reverse engineering**

I. INTRODUCTION

During the past decades, agile methods have gained prevalence to enable fast-paced software development [1]–[5]. The design structure of a system continuously evolves as the consequence of fast-paced code revisions. Developers often unintentionally introduce design flaws that violate well accepted design principles. Overtime, the system becomes ever challenging to understand and maintain [6]. Agile techniques, such as continuous testing [7]–[9], ensures the function goals of a system with every code revision. However, there still lacks an efficient approach that can continuously model the design difference resulting from every single code revision to facilitate comprehension and ensure the design quality.

Many agile methods advocate collective code ownership and continuous refactoring for enabling the "Built-In Quality" principle. The development team should perform Litter-Pickup refactoring following the boy-scout rule: Leave your code better than you found it. Therefore, developers need a way to quickly and correctly understand the high-level design differences resulting from their own and others' code revisions. Previous research has developed various techniques to recover the overall design of a software system to improve understanding [10]–[18]. The limitation is that existing approaches scan the entire code base to build the design model, and thus are too expensive to apply after every code revision in fast-paced development process. In fact, a code revision usually only alters (if at all) a very small part of the design. We argue that it is more efficient to model the design difference by focusing on the changed parts in an incremental fashion.

In addition, developers often unintentionally introduce design flaws, which are formed by problematic connections among software elements. Previous research has revealed typical design flaws that recur in different projects [19]–[21]. For example, *Cyclic Dependency* is a group of source files that form a dependency cycle. Strong evidences from dozens of real-life projects, both open source and industry, reveal that typical design flaws are responsible for poor maintenance quality and high long-term costs [19], [21]–[23]. The limitation is that these design flaws usually have already caused significant loss to the system when they can be detected by existing approaches. In practice, due to limited resources, the design flaws were seldom eliminated by refactoring and continued causing more loss to the project [24], [25]. We argue that a more effective solution is to instantly identify the introduction (as well as the elimination) of these design flaws with rapid code revisions.

This paper contributes an efficient approach, called *Design Differencing* (DESIGNDIFF). It automatically and efficiently models and visualizes the essential design differences resulting from any code revision as a sequence of design change operators. This paper defines a comprehensive set of 17 fundamental design change operators. These operators, applied in combinations and/or in sequences, can capture any drastically design structure change in a system. The unique advantage of our approach is that it focuses on modeling the changed parts,

instead of re-modeling the entire design structure of a system. As such, it is affordable to be applied after every code commit. DESIGNDIFF can also help to instantly detect the introduction and elimination of design flaws with rapid code revisions. This paper contributes the corresponding tool that automatically generates such design differences and visualization, taking a commit ID and the code base of the system as inputs. The tool developed and data used in this paper can be found here [1].

1) *Can* DESIGNDIFF *facilitate the understanding of the high-level design differences resulting from real-life software commits?* A user study involving 10 developers reviewing 20 real-life software commits of different complexity levels indicates that DESIGNDIFF can help improve the completeness and correctness of understanding with less time. In the study, we compared with the code diff viewer from GitHub [26] as the baseline. The rationale is that 1) GitHub is the most accessible tool that highlight code difference in every single code commit; and 2) to the best of our knowledge, there is no other available tool that can compute the high-level design difference from every single code revision. Therefore, we believe that DESIGNDIFF can be a handy tool for developers to perform design review for every single code commit.

2) *Can* DESIGNDIFF *help to instantly detect the introduction and elimination of design flaws with every single code revision?* We applied DESIGNDIFF on a total of 14,832 commits from five open source projects on the Apache Software Foundation [27]. We found that only 4,189 (28%) commits altered the design structure of a system. DESIGNDIFF further detected that 855 (20%) of the 4,189 design structure altering commits introduced four types of architectural flaws, including *cyclic dependency, parent calls child, sibling call each other* and *declare child class* which violated the SOLID design principles [28], [29]. On the contrary, only 337 (8%) commits fixed the flaws. The latency between the flaw introduction and elimination is on average 2 months to 2 years! This indicates that DESIGNDIFF can help to safeguard the design quality by providing instant warning to developers when design flaws are first introduced.

3) Finally, the overhead of DESIGNDIFF is averagely from 13 to 29 seconds for analyzing commits in different sizes, which is affordable to be employed for analyzing every single change.

II. BACKGROUND

In this section, we introduce the fundamental background.

a) Dependency Graph and Design Structure Matrix

A dependency graph captures the structural dependencies among software design elements [30]. The nodes are the software design elements in different granularity (e.g. components, source files, and methods); and the edges are the dependencies among software design elements [31]. It can be reverse engineered from the source code of a software system using a widely-used commercial tool, Understand [32].

A design structure matrix (DSM), proposed by Baldwin and Clark [33], can compactly represent a dependency graph of a software system as a square matrix [34]–[37]. The rows and columns in a DSM are labeled by the same set of software design elements in the same order. A cell along the diagonal represents self-dependency, and a non-empty off-diagonal cell captures the dependency between the element on the row and the element on the column. In Java, there are nine common types of structural dependencies among source files, extracted by Understand [32]. In common programming languages like Java and C++, each file usually contains the definition of one major class, therefore we use "class" and "file" interchangeably for the ease of presentation. Each dependency type is explained as follows.

- f_a *Implement* f_b: f_a (class) implements f_b (interface)
- f_a *Extend* f_b: f_a (child class) extends f_b (parent class)
- f_a *Call* f_b: f_a calls the methods declared in f_b
- f_a *Throw* f_b: f_a throws an exception of the type of f_b
- f_a *Cast* f_b: f_a is casted into the type of f_b
- f_a *Create* f_b: f_a creates an instance of f_b
- f_a *Typed* f_b: f_a declares a variable of the type of f_b
- f_a *Use* f_b: f_a uses a value assigned from the type of f_b
- f_a *Import* f_b: f_b is imported at the header of f_a

Fig. 1a presents an example of a file-level DSM of the Maze Game program (see Section III-A). Cell[1,7] says "Tp, Cr". It means that file 1, *Door*, declares a parameter type of (Typed) and create an instance of (Create) file 7, *Room*.

b) Modular Operators

According to the design rule theory proposed by Baldwin and Clark [33], any complex system is composed of high-level design rules and modules. They proposed to capture the dynamics of a modular structure by six simple modular operators, which are a powerful set of conceptual tools to capture the dynamics of a modular design. These six modular operators are:

- *Augmenting*: add a new module to a system
- *Excluding*: remove a module from a system
- *Porting*: a module ports the functions from another module
- *Inverting*: create a new design rule from existing module(s)
- *Splitting*: break a module into sub-modules
- *Substituting*: replace one module by another module

These six operators form the theory of design evolution. They can be applied in combinations and in sequences to drastically change the design structure of a system.

III. THE DESIGNDIFF APPROACH

The *Design Differencing* (DESIGNDIFF), which can incrementally model and visualize the design structure differences resulting from each code revision, is composed of two key parts:

- A *Difference Design Structure Matrix (Diff-DSM)* that captures the fundamental design structure differences in a design structure matrix (DSM)
- A complete set of 17 basic *design change operators* that can capture the high-level design difference.

We use a Maze Game program as a running example to illustrate the key ideas.

[1] https://drive.google.com/open?id=1blZgWoKRSn9s1SYOFoYWLk5MYkpZCNSw

A. A Running Example

We will use a Maze Game program as a running example to explain the key concepts in DESIGNDIFF and as one of the subjects to evaluate DESIGNDIFF. The Maze Game program is a homework in an undergraduate course in a Software Engineering program. This homework is designed to teach students the Factory Method (FM) and Abstract Factory (AF) design patterns. Students first implement a basic maze game, which creates a maze composed of black-and-white rooms, walls and doors, following the map-site specified in a configuration file. Next, students are asked to improve the program to create a colorful maze game using the FM and AF design patterns respectively. The functional properties of using the two design patterns are identical: the program should be able to create three different modes of maze games: 1) the black-and-white maze, 2) the red maze, and 3) the blue maze depending on runtime input.

B. Difference Design Structure Matrix

A Diff-DSM focuses on modeling the directly or indirectly changed parts in a code revision. Similar to the original DSM (introduced in II), a Diff-DSM is also a *n*n* square matrix, where the rows and columns represent impacted source files and each cell represents structural dependencies from a file on the row to the file on the column. Diff-DSM is unique in two aspects.

1) A Diff-DSM only captures files impacted, directly or indirectly, by a code change, c. The impacted files are in four types, distinguished by the prefix of each file in a *Diff-DSM*.

- f_{Add}: files that are newly added to the system in c
- f_{Remove}: files that are removed from the system in c
- f_{Modify}: files that are modified in c
- $f_{Referenced}$: files that do not themselves change in c, but are referenced by f_{Add}, f_{Remove}, or f_{Modify} in c

Example. Fig. 1a reports the Diff-DSM for the code change from a basic maze game to a colorful maze game using the FM design pattern. The first column lists the files that are directly or indirectly impacted in the change. Nine files are newly added, five files are referenced by the changed files, and one file is removed.

2) A Diff-DSM distinguishes structural dependency *types as added, removed, and unchanged in c.* The added and removed dependency types are marked with "+" and "-" (and also highlighted in blue and red text colors) in cells.

Example. In Fig. 1a, cell[15,3] says *"+Cl, +Cr"* in blue, indicating that these two dependency types (Call and Create) are newly added (as a result of newly added files). Cell[12,4] says *"-Tp, -Cr"* in red, denoting that *SimpleMaze* (row 12) no longer declares types (Typed) and creates the instance of (Create) the *Wall* (row 4) after change.

C. Design Change Operators

Diff-DSM captures the basic structure changes without capturing the high-level design difference. For example, it is possible that several files forming a new polymorphism structure are added to the system. The *Diff-DSM* only captures this case as a set of file and dependency additions, without capturing them as an introduction of a new polymorphism structure. Therefore, we define a complete set of 17 basic design change operators that, by logic, capture all possible design changes in a code change. They are defined and derived from Baldwin and Clark's design rule theory [33] and the object-oriented design philosophy [28], [29]. We will first discuss the rational of the 17 operators and justify why they are complete and general to different object-oriented programming languages, followed by the detailed definition of each operator. At the end, we use the running example to illustrate the operators.

Rational and Completeness. Software design evolution can be captured based on the grounds of Baldwin and Clark's six modular operators (see Section II). The first three types are the most fundamental: *Augmenting*, *Excluding* and *Porting*. Treating each source file as a fine-grained design module, *Augmenting* and *Excluding* map to adding and removing source files. *Porting* can be either *Importing* or *Deporting*, which map to the addition or removal of dependencies among source files.

In different programming languages, there are different types of structural dependencies. However, regardless of the specific language, polymorphism is an important design decision [28], [29]. The relationship between two source files can be categorized as polymorphic or regular (i.e. other non-polymorphic dependencies). For example, as introduced in Section II, in Java, Extend and Implement, are two concrete forms of polymorphic relationship. The other seven types are regular dependencies. Following this rationale, in the descending order of significance, *Augmenting* (Operator 1 to 5) can be elaborated into *Augmenting an Entire Polymorphism Tree*, *Augmenting a Partial Polymorphism Tree*, *Augmenting an Interface*, *Augmenting a Child*, and *Augmenting a Regular File* . Similar breakdown can be applied to *Excluding* (Operator 6 to 10) and *Importing* (Operator 11 to 13)/*Deporting* (Operator 14 to 16). To be comprehensive, operator 17 captures the modification to the dependency types.

These 17 operators are *complete* and *general* in capturing all the fundamental design changes in software evolution. Although we do not directly define operators based on the other three modular operators, namely *Inverting*, *Splitting*, and *Substituting*, the 17 operators already combine or can be combined to capture them. We will explore the dynamics of their combination in the future (see Section VIII).

Definitions. Given any code commit c, we define the following operators that can be used in combination and sequential to interpret the design structure difference resulting from c:

1. Augmenting an Entire Polymorphism Tree, T_a: All the files in T_a are newly added in c and they form a **new** polymorphism tree. That is, none of the files in T_a belongs any preexisting polymorphism tree in the code base before c. This operator indicates the introduction of new polymorphism design structure.

Namely,

2. Augmenting a Partial Polymorphism Tree, PT_a: The files in PT_a are newly added in c and they become part of a

preexisting polymorphism tree PT. This operator indicates the expansion of PT.

3. Augmenting an Interface, f_{i_a}: File f_{i_a} is added in c as a parent class or an interface of an preexisting file in the system. This operator indicates extracting general interfaces from existing functions for future extension. As such, more variations can be added as the child/concrete implementation in future code revisions.

4. Augmenting a Child, f_{c_a}: File f_{c_a} is newly added in c as a child/concrete implementation of a preexisting parent/interface in the system. This operator indicates a concrete extension to a parent/interface.

5. Augmenting a Regular File, f_a: File f_a is added in c, and f_a is *not* part of any polymorphism tree. This operator indicates direct addition of new functions.

6. Excluding an Entire Polymorphism Tree, T_r: An entire polymorphism tree T_r is removed in c. This operator eliminates polymorphism design structure that is no longer needed, which is the opposite of operator 1.

7. Excluding a Partial Polymorphism Tree, PT_r: PT_r is part of a polymorphism tree PT before c and is removed in c. The remaining part of PT after c is still a polymorphism tree. This operator reduces or simplifies the structure of PT.

8. Excluding an Interface, f_{i_r}: File f_{i_r} is removed in c. Before c, another file implements or extends f_{i_r}. This operator indicates the removal of a general interface.

9. Excluding a Child, f_{c_r}: File f_{c_r} is removed in c. Before c, f_{c_r} implements or extends another file before c. This indicates the removal of a concrete variation of an existing interface.

10. Excluding a Regular File, f_r: File f_r is removed in c, and f_r is not part of any polymorphism tree before c. This operator simply removes existing functions.

11. Importing Polymorphism, (f_c, f_p): Developers add polymorphism relationship between file f_c (child/concrete) and file f_p (parent/interface). Both f_c and file f_p exist before c. This operator is needed when developers identified polymorphism relationship between two existing files.

12. Importing Regular Dependency, (f_x, f_y): Before c, both file f_x and file f_y exist but f_x does not depend on f_y. In c, developer adds regular dependencies from f_x to f_y. This operator increases the coupling among files.

13. Enhancing Regular Dependency, (f_x, f_y): File f_x already depends on file f_y before c. In c, developer adds more types of regular dependencies from f_x to f_y. This operator strengthens the dependencies between files.

14. Deporting Polymorphism, (f_c, f_p): Developer removes the polymorphism relationship from file f_c (child/concrete) to file f_p (parent/interface) in c. This operator eliminates no longer needed polymorphism relationship between files.

15. Deporting Regular Dependency, (f_x, f_y): File f_x depends on file f_y before c. In c, developer completely removes all the dependencies from f_x to f_y. This operator eliminates dependencies between files, which in turn decouples the system.

16. Simplifying Regular Dependency, (f_x, f_y): File f_x depends on file f_y before c. In c, developer removes some types of dependency from f_x to f_y. This operator weakens the dependency between files but not completely eliminating it.

17. Modifying Regular Dependency, (fx, fy): File f_x depends on file f_y both before and after c. The types of regular dependencies from f_x to f_y are added and removed. This indicates the nature of the relationship between files changed. For example, it could change from a method call to an instance creation.

Example. The implementation of the colorful Maze Game using FM pattern can be modeled as a sequence of five operators, as shown in Fig. 1b:

1) Op1: *Augmenting Doors*: Add *BrownDoor* (row 2) and *GreenDoor* (row 3) as the child classes of *Door* (row 1);

2) Op2: *Augmenting Walls*: Add *DarkBlueWall* (row 5) and *RedWall* (row 6) as the child of *Wall* (row 4);

3) Op3: *Augmenting Rooms*: Add *LightBlueRoom* (row 8) and *PinkRoom* (row 9) as the child of *Room* (row 7).

3) Op4: *Excluding SimpleMaze*: Remove the original *SimpleMaze* (row 12) which only creates the black-and-white maze in the basic version.

2) Op5: *Augmenting Factory*. Lastly, create the factory method inheritance structure by adding three new files, i.e. *MazeFactory* (row 13) as the parent, and *BlueMazeFactory* (row 14) and *RedMazeFactory* (row 15) as two child classes. The parent class *MazeFactory* uses the basic products, namely, *Door*, *Wall*, and *Room*, to create a black-and-white maze. Each child factory class, *BlueMazeFactory* and *RedMazeFactory*, uses the specific concrete products, namely, *BrownDoor-DarkBlueWall-LightBlueRoom*, and *GreenDoor-RedWall-PinkRoom*, as the building blocks of red and blue mazes respectively.

IV. Approach Implementation and Applications

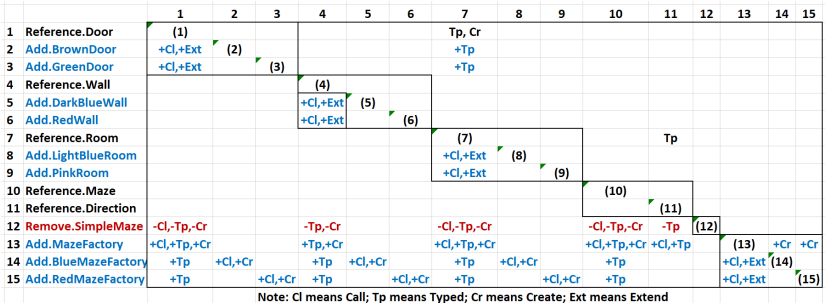
This section introduces the implementation of DesignDiff, followed by the discussion of its potential applications.

A. ArchDiff Implementation

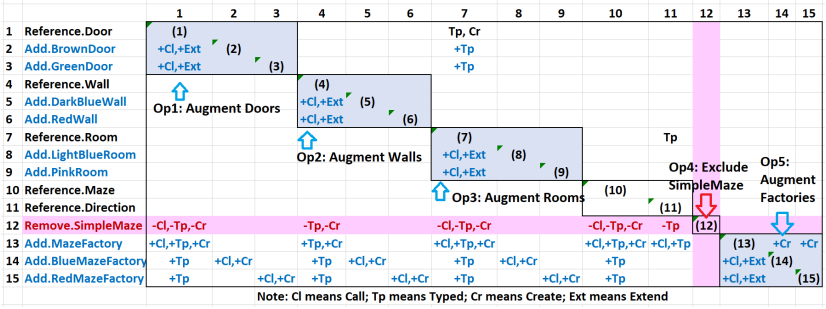
Given a commit ID and the code base of a system as inputs, DesignDiff automatically interprets and visualizes the design difference shown in Fig. 1. The implementation overview is illustrated in Fig. 2, composed of two main parts: 1) Diff Extraction and 2) DesignDiff Modeling.

a) Diff Extraction

This part extracts the directly and indirectly involved source files in a given commit c. We use Git APIs to identify source files directly revised in c, namely *ChangedFileSet*, and revert the code base into two status before and after the commit, namely *CodeBaseBefore* and *CodeBaseAfter*. Next, the "Change Extractor" retrieves two file sets, *ReferencedBefore* and *ReferenceAfter* that are not directly changed in c but are referenced by files in *ChangedFileSet*, from *CodeBaseBefore* and *CodeBaseAfter* respectively. *ReferencedBefore* and *ReferenceAfter* are necessary to comprehensively capture indirect design impacts of c. We combine *ChangedFileSet* with *ReferencedBefore* and *ReferenceAfter* respectively to form *InvolvedFileSetBefore* and *InvolvedFileSetAfter*, containing the directly and indirectly involved files before and after c.



(a) Diff-DSM



(b) Design Difference

Fig. 1: Operators for Colorful Maze Game Using Factory Method Pattern

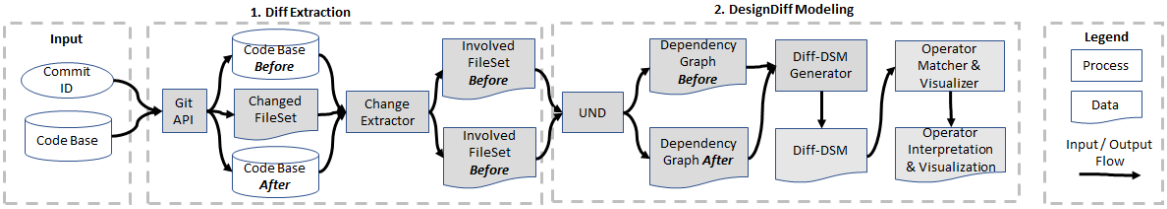


Fig. 2: DESIGNDIFF Implementation Overview

b) DESIGNDIFF *Modeling*

Next, we model a commit as a sequence of design change operators defined in Section III.

First, we use Understand [32] to calculate two subgraphs, denoted as G_{before} and G_{after}, formed by *InvolvedFileSetBefore* and *InvolvedFileSetAfter* respectively.

Next, the "Diff-DSM Generator", a simple graph comparator, identifies the difference between G_{before} and G_{after}. It first identifies the discrepancies of the nodes in G_{before} and G_{after}. As such files added, removed, and modified can be distinguished. The referenced files are tagged in "Diff Extraction". It then compares edges in G_{before} and G_{after} to identify added or removed dependency types between any two files. The output is a *Diff-DSM* highlighting file and dependency changes as shown in Fig. 1a.

The "Operator Matcher and Visualizer" takes a *Diff-DSM* as input and 1) matches the patterns of the 17 operators and 2) interprets the design difference as a sequence of operators. Each type of operator is identified by a respective matching procedure. The input is the *Diff-DSM* of a commit; the output is the instances of a particular type of operator formed by the involved files and/or dependencies. Due to space limit, we only illustrate the pseudo code of two representative matchers: *matchAugmentingEntirePolyTrees* in Algorithm 1 and *matchExcludingPartialPolyTrees* in Algorithm 2. The pseudo code of the other 15 matchers are included in the link below[2]. In Algorithm 1, the first step (line 1) is to traverse the added files in a *Diff-DSM*, $dDsm$, based on the polymorphism relationship. The output of this step are clusters of newly added files, namely $trees$, where files in each cluster are connected by the polymorphism relationship (i.e. form a polymorphism tree). We loop through the $trees$ (line 2 to line 8). In each iteration, we check whether the current $tree$ contains more than 1 source file (line 3); otherwise, it is not a real tree. Then we check whether files in the $tree$ share polymorphism relationship with files outside of it (line 4). If not, the $tree$ is added to the return set since it is an entirely new polymorphism tree. Algorithm 2 works similarly. The differences are in line 1 and line 4. In line 1, we traverse the removed files instead of the added files in $dDsm$ to identify polymorphism trees formed by the removed files. In line 4, we check whether files in a tree share polymorphism relationship with files outside of the tree. If so, it matches an excluded partial polymorphism tree operator. After all the operators are matched in a commit, the "Operator Visualizer" orders identified operators based on two heuristics:

[2]https://drive.google.com/open?id=1c1zKt-TuYb0_M9jllsdRS0bL5-crA5Tk

1) Instances of the same type of operators are visualized in one view due to their commonality; and 2) Operators related to polymorphism structure will be visualized first. The rational is to prioritize operators that are more likely to increase the system complexity. Admitted, these heuristics may not be optimal, we will discuss this in Section VIII.

Algorithm 1 matchAugmentingEntirePolyTrees

Input: $dDsm$
Output: new_trees
Declare: $new\_trees = \emptyset$
1: $trees = traverseAddedFiles(dDsm, \text{“}poly\text{”})$
2: **for** $tree$ in $trees$ **do**
3: **if** $sizeof(tree) > 1$ **then**
4: **if** $\forall f \in tree, dDsm.neighbors(f, \text{“}poly\text{”}) \subseteq tree$ **then**
5: $new\_trees.add(tree)$
6: **end if**
7: **end if**
8: **end for**
9: **return** new_trees

Algorithm 2 matchExcludingPartialPolyTrees

Input: $dDsm$
Output: $excluded\_partial\_trees$
Declare: $excluded\_partial\_trees = \emptyset$
1: $trees = traverseRemovedFiles(dDsm, \text{“}poly\text{”})$
2: **for** $tree$ in $trees$ **do**
3: **if** $sizeof(tree) > 1$ **then**
4: **if** $\exists f \in tree$ $\exists f_i \in dDsm.neighbors(f, \text{“}poly\text{”}), f_i \notin tree$ **then**
5: $excluded\_partial\_trees.add(tree)$
6: **end if**
7: **end if**
8: **end for**
9: **return** $excluded\_partial\_trees$

B. DESIGNDIFF *Potential Applications*

DESIGNDIFF is a promising tool for software practitioners in different application areas:

a) Design Structure Change Comprehension and Review

Software development is a collaborative and long-term effort [38]. In some agile methods, teams even advocate collective code ownership and continuous refactoring [39]. Therefore, it is important for developers to understand each other's changes at the high level, as well as their own changes in the past. Code review has been an important technique for ensuring quality [40]. DESIGNDIFF can be used to facilitate the code review emphasizing on the design change.

b) Continuous Design Structure Monitoring

Developers constantly make changes to a system. Due to the lack of experience or pressed by time, developers tend to introduce sub-optimal implementation that form "technical debts" [41], [42]. Therefore, it is important to provide continuous design quality monitoring to provide instant or early warning when changes introduce negative impact to the overall design structure. DESIGNDIFF enables such continuous design structure monitoring. For example, if a change introduced design flaws, such as cyclic dependencies or unhealthy inheritance relationships [43]–[45], developers should receive an instant warning to take immediate actions. On the opposite, if developers eliminated design flaws, they should also receive a positive feedback to acknowledge the effort.

c) Software Design Education

The topics in software design are challenging for students who lack practical experience [46]. Although students practice good software design in programming homework, it is often extremely inexplicit and inefficient for instructors to evaluate students' work in terms of the design correctness and quality. DESIGNDIFF is a promising tool to overcome challenges for both students and educators.

TABLE I: Study Subjects

Project	#Files	#Commits	LOC	Duration(#Ms)
Avro	896	643	126K	114
Chemistry	1053	1466	131K	107
Jena	5886	4780	515K	83
Mina	319	1702	23K	136
PDFBox	1289	6241	152K	119

V. EVALUATION SETUP

This section discussed how we evaluate DESIGNDIFF.

A. Research Question

We propose to answer the following research questions:

- RQ1: Can DESIGNDIFF facilitate the understanding of the high-level design differences resulting from real-life software system commits?
- RQ2: Can DESIGNDIFF help to instantly detect the introduction and elimination of design flaws with every single code revision?
- RQ3: What is the performance overhead of DESIGNDIFF?

B. Evaluation Subjects and Setup

To answer RQ1 and RQ2, we applied DESIGNDIFF on a total 14,832 of commits from five popular, real-life software systems [47]–[51]. They are Avro—a remote procedure call and data serialization framework; Chemistry—open source Content Management Interoperability Services; Jena—Semantic Web framework for Java; Mina—a Java network application framework; and PDFbox—a pure-Java library for manipulating PDF files. The basic facts about these projects are shown in Table I. As a particular note, DESIGNDIFF *identified 4,189 (28%) of the total 14,832 commits that altered the design structure of the projects*. The remaining 72% commits did not alter the software design structure.

a) RQ1 Setup

We conduct a user study and survey to evaluate the effectiveness of DESIGNDIFF for understanding the design structure changes in real-life software commits. We compare with using the diff viewer on GitHub [26], referred as GitDiff in the following context. GitDiff presents the detailed code revisions and highlights the added and removed lines of code in green and red background. We use GitHub as the baseline because 1) it is the most widely used and readily available tool

that highlight code difference in every single code commit; and 2) to the best of our knowledge, there is no other available tool that directly and automatically calculates and visualize the high-level design difference from every single code revision.

To create a fair comparison, we employ the following setting in the study: We selected 4 commits from each of the 5 projects (20 commits total) in 4 different complexity levels, approximated by the number of involved files: $<= 5$, (5, 10], (10, 15], and > 15. Intuitively, the more files involved, the more difficult to understand. We invited 10 developers with 1 to 6 years of real-world development experience in the study.

The study is arranged in five sessions shown in Table II. Each session is participated by 2 developers (column 2). Each developer reviews 8 commits from two projects (column 3) using DESIGNDIFF and GitDiff. For instance, in Session 1, participant *A* uses DESIGNDIFF on Avro and GitDiff on Chemistry; while participant *B* uses GitDiff on Avro and DESIGNDIFF on Chemistry. As such, each participant reviews 8 cases from two projects by DESIGNDIFF and GitDiff respectively; and each commit is reviewed four times, each time by a different participant, using DESIGNDIFF and GitDiff evenly. Therefore, the study results in a total 40 review cases using DESIGNDIFF and 40 cases using GitDiff.

Before each session, we provide a 45 minutes training/exercising to get participants familiarized with both methods. We ask the participants to summarize the design changes in each commit. We clarify our expectation by showing samples of expected summary and educate the participants with the 17 operators. We track the time spent on each case.

We evaluate each summary in a scale of 1 to 5. A 5 means a correct and complete summary; 4 if the summary contains minor inaccuracies; 3 if important information is missing or misinterpreted; 2 if the summary is largely vague/irrelevant; 1 if the summary is mostly vague/irrelevant. As an example, a participant got 1 using DESIGNDIFF on a summary: "inheritance change", without any specific information of which and how files are involved. To ensure fair evaluation, one author performs the initial grading and another author confirms.

We collect feedback from participants using a brief survey after the study. We ask three (two rating and one open-ended) questions in the survey: *1) How useful is* DESIGNDIFF *for helping understanding design changes? (Useful, Not Much Useful, Useless); 2) How useful is GitDiff for helping understanding design changes? (Useful, Not Much Useful, Useless); 3) What is the strength and weakness of* DESIGNDIFF *and GitDiff?*

RQ1 is answered through: 1) the quality and used time in understanding; and 2) participants' feedback.

b) RQ2 Setup

We focus on four types of design flaws that violate well accepted design principles [28], [29].

- Cyclic Dependency: A group of files form a dependency cycle [43]–[45].
- Parent Calls Child: The parent (abstract) classes should not depend on the child (concrete) classes. Otherwise, it violates the Dependency Inversion Principle [29]: the concrete should depend on the abstract but not vice versa.

TABLE II: Study Sessions

S#	Par.	Commits (Method)
1	A; B	Avro (A: DESIGNDIFF; B:GitDiff); Chemistry (A: GitDiff; B: DESIGNDIFF)
2	C; D	Chemistry (C: DESIGNDIFF; D: GitDiff); Jena (C: GitDiff; D: DESIGNDIFF)
3	E; F	Jena (E: DESIGNDIFF; F: GitDiff); Mina (E: GitDiff; F: DESIGNDIFF)
4	G; H	Mina (G: DESIGNDIFF; H: GitDiff); PDFBox (G: GitDiff; H: DESIGNDIFF)
5	I; J	PDFBox (I: DESIGNDIFF; J: GitDiff); Avro (I: GitDiff; J: DESIGNDIFF)

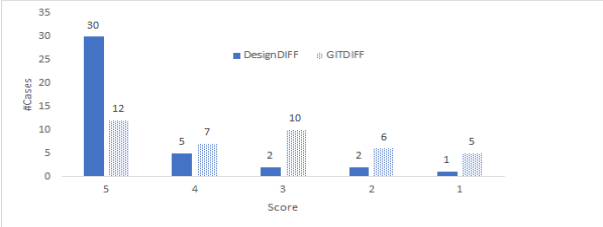


Fig. 3: Summary Quality

- Sibling Calls: According to the Livos Substitution Principle [29], the subclass should be able to substitute each other. But if the sibling classes call each other, they cannot be used interchangeably due to their coupling.
- Declare Child: Developers should declare the abstract/parent class, instead of the concrete/child, to increase the flexible and interchangeable instantiation of concrete/child classes. Declaring a child class violates both Likov Substitution and Dependency Inversion Principles [29].

DESIGNDIFF can help detect commits that introduce these flaws. For example, to detect the introduction of cyclic dependencies, we calculate cycles in a *Diff-DSM* formed by after-change files-and-dependencies that do not exist in before-change files-and-dependencies. Other flaws can be detected in similar ways following the before-after criteria using a *Diff-DSM*. The negation of flaw detection pin-points the flaw elimination.

c) RQ3 Setup

We measure the performance overhead of running DESIGNDIFF against the 4,189 design structure altering commits on a PC with I7 CPUS, 2.6GHz, 4 cores, 32 GB RAM, 64-bit Operating System.

VI. EVALUATION RESULTS

RQ1: Can DESIGNDIFF facilitate the understanding of the high-level design differences resulting from real-life software system commits? We first present the understanding quality and time by using DESIGNDIFF and GitDiff, then present the survey results in four findings:

1) DESIGNDIFF *improves the quality of understanding obviously, compared to GitDiff.* As introduced in Section V, we grade the 40 cases by DESIGNDIFF and 40 cases by GitDiff in the scale of [1,5]. Fig. 3 shows the grade distribution.

When using DESIGNDIFF, 30 (75%) cases received 5', indicating the completeness and correctness in the summaries of design changes. Participants lost points mostly (8 in 10 cases) due to missing minor dependency changes or being inaccurate when describing the changed files. In 2 cases, the participate submitted vague summary or mixed up changed dependency types. The Mann-Whitney U-tests indicates that DESIGNDIFF can statistically significantly improve the understanding quality with p-value 0.0003.

TABLE III: Efficiency Improvement for Participants

Par.	Avg. Time (Minutes)			Avg. Quality (Grade [1,5])		
	DESIGNDIFF	GitDiff	Saved	DESIGNDIFF	GitDiff	Improved
D	7	8.5	1.5	5	3.5	1.5
H	4.5	6.5	2.0	4.25	3.75	0.5
C	4.5	7.5	3.0	4.25	1.75	2.5
I	5.75	10.5	4.8	5	4.25	0.75
F	8	18.25	10.3	5	3.25	1.75
B	4	9.75	5.8	5	4.5	0.5
Avg			4.5			1.25
E	7.75	2.5	-5.3	4.75	2.25	2.5
A	9.5	4.25	-5.3	4.5	2.75	1.75
Avg			-5.3			2.1
G	6.25	4	-2.3	5	5	0
J	2.75	2	-0.8	2.5	2.75	-0.25
Avg			-1.5			-0.1

In comparison, when using GitDiff, only 12 (30%) cases received 5'. The summaries suffer from the following issues: 1) The summaries failed to completely capture the dependency change among files (13 in 28 cases). In particular, participants tend to miss porting/deporting dependencies or dependency change that involve referenced files. Consistent with the survey feedback, participants complain that GitDiff only explicitly distinguishes changes to files as added, removed, or modified by the background color (all green for added file, all red for removed, and mixed background for modified file). However, the dependency changes are implicit and tedious to capture. 2) The summaries did not contain detailed information of which and how files are involved in a change (12 in 28 files). For example, when an entire polymorphism tree is added, the summary only vaguely indicates "some inheritance structure is added", without providing details. 3) The summaries mistaken the roles of files and the nature of their dependencies(5 in 28). For example, participants mixed up child and parent or mistaken polymorphism relationship as regular dependency. 4) The summaries look like a detailed "laundry list" without cohesive understanding (2 in 28 cases). For example, "Augmenting an Entire Polymorphism Tree" formed by a parent class A and three child classes B, C, and D is described fragmented as B extends A, C extends A, and D extends A following the order of how changed files are listed on GitDiff. In 1 case, the participant did not submit the summary, later confirmed by the participant having difficulty identifying any meaningful design changes from GitDiff.

2) For the majority (60%) participants, DESIGNDIFF *both improves the understanding quality (by an average 1.25') and reduces the review time (by an average of 4.5 minutes).* The data is shown in Table III. The first column shows the unique ID of each participant. Column 2 to 4 shows the average time each participant spent when using DESIGNDIFF and GitDiff, and the time saved by using DESIGNDIFF. Similarly, Column 5 to 7 shows the grade received when using DESIGNDIFF and GitDiff, and the grade improved by using DESIGNDIFF.

For 2 participants, they spent 5 more minutes when using DESIGNDIFF, but the understanding quality is significantly improved by 1.75' and 2.5'. According to the survey, DESIGNDIFF provides systematically guidance that helps users remain engaged until high quality understanding is achieved; while GitDiff tends to overwhelm and discourage users from further pursuing understanding by details.

For the remaining 2 participants, the effectiveness of DESIGNDIFF did not kick off in the study. They spent more time (0.8 and 2.3 minutes) using DESIGNDIFF, without improving the understanding quality. We conjecture that this is impacted by individual background and experience. The DESIGNDIFF uses a matrix representation, thus the participants' performance will be compromised if he/she is not comfortable with reading a matrix. This is supported by the comments from the survey: two participants suggested that we replace the DSM view by a graph view, which is more intuitive to them. However, we believe that this is amendable with more training and exercises.

3) *The survey shows that all participants favor* DESIGNDIFF *over GitDiff due to its explicitness in showing the design changes.* All 10 participants think that DESIGNDIFF is *useful* for understanding design structure differences. In comparison, 4 participants think that GitDiff is *useful*; 5 think that GitDiff is *not much useful*; and 1 thinks that GitDiff is *useless*. However, four participants complained the complexity of the DESIGNDIFF visualization and the high learning curve. They suggest simplifying the view, especially for large changes, such as by eliminating trivial operators like "Modify Regular Dependency". As mentioned earlier, two participants suggested that we use a graph-based view to replace the DSM view. This suggestion is opposite to the appreciation for the compactness of the DSM view from other participants. For GitDiff, two participants appreciated the clarity of the background color for highlighting the added and removed lines of code. However, most participants complained the fragmentary information and overwhelming details that challenge the understanding of high-level design differences.

RQ1 Answer: DESIGNDIFF can help most participants to better and faster understand the design structure differences in real-life software project commits. Participants mostly favor DESIGNDIFF over GitDiff due to its clarity and explicitness in showing the essential design change.

RQ2: Can DESIGNDIFF help to instantly detect the introduction and elimination of design flaws with every single code revision?

With the help of DESIGNDIFF, we found that, among the 4,189 commits that changed the design structure of the studied projects, totally 855 commits introduced and 337 commits eliminated design flaws. The detailed information is shown in Table IV. The first column lists the project name and the number of commits that changed the design structure. The 2nd and 3rd column show the numbers of commits that introduced and eliminated *Cyclic Dependency* respectively. The 4th column shows the latency (in days) between the introduction and elimination of each flaw instance. The following columns show the similar information for the other three design flaws. We can make the following observations from the data:

1) Overall, developers are 2 (Cyclic Dependency) to 5 (Parent Call Child) times more likely to introducing than eliminating design flaws. For instance, in Avro, 37 commits (which is 15% of the total 248 design altering commits) introduced new cyclic dependencies among the revised files.

TABLE IV: Real-time Detection of Commits with the Flaw Introduction and Removal

Project	Cyclic Dependency			Parent Call Child			Sibling Call			Declare Child Class		
(#Arch. Commits)	#In. (%)	#El. (%)	L(D)	#In. (%)	#El. (%)	L(D)	#In. (%)	#El. (%)	L(D)	#In. (%)	#El. (%)	L(D)
Avro (248)	37 (15%)	7 (2.4%)	1179	7 (2.8%)	1 (0.4%)	-	9 (3.6%)	1 (0.4%)	162	24 (9.7%)	4 (1.6%)	1
Chemistry (321)	15 (4.6%)	6 (1.8%)	5	5 (1.6%)	1 (0.3%)	-	9 (2.8%)	3 (0.9%)	8	11 (3.4%)	3 (0.9%)	-
Mina (601)	37 (6.1%)	22 (3.7%)	56	6 (1%)	1 (0.2%)	1	23 (3.8%)	3 (0.5%)	5	31 (5.2%)	11 (1.8%)	76
PDFBox (1392)	149 (10.7%)	75 (5.4%)	579	37 (2.7%)	5 (0.4%)	1599	53 (3.8%)	13 (0.9%)	990	111 (8%)	35 (2.5%)	94
Jena (1627)	88 (5.4%)	65 (4%)	174	20 (1.2%)	10 (0.6%)	377	72 (4.4%)	31 (1.9%)	5	111 (6.8%)	40 (2.5%)	35
Avg. In./Rm.	8.4% / 3.4% = 2.4		399	1.9% / 0.4% = 5		645	3.7% / 0.9% = 4		234	6.6% / 1.9% = 3.5		51

In comparison, only 7 commits (2.4%) eliminated cycles. This is consistent with prior empirical experience that design flaws are not granted sufficient attention. Therefore, they are easily introduced, but seldom get eliminated. This indicates that it is important to provide real-time warning of the introduction of flaws so that the developers can take immediate corrections before maintenance consequences accumulate. Meanwhile, developers should also receive positive feedback when they make an effort to eliminate the flaws.

2) The latency between the introduction and elimination of a design flaw is on average 51 days (Declare Child Class) to 645 days (Parent Call Child)! As shown in Table IV, the latency before elimination varies drastically from case by case: from 1 day to 1559 days (sub-columns "L(D)"). To clarify, not every introduced flaw is removed; and not every removed flawed can be traced back to its original introduction on Git log. The latency is calculated based on matched introduction and elimination cases. The data indicate that most design flaws are not eliminated in a timely fashion. Admittedly, the latency is impacted by many factors, such as unawareness, lack of experienced and time, priority of the issues, etc. However, we believe that if developers can receive instant warning from DESIGNDIFF, they have the option to eliminate these flaws immediately after the introduction with a fresh memory.

RQ2 Answer: DESIGNDIFF successfully detected 855 commits that introduced and 337 commits that eliminated four types of design flaws among the total 4,189 commits that changed the design structure of the studied projects. The latency between introduction and elimination of a design flaw is on average from 51 to 645 days. Based on our evaluation, we believe that DESIGNDIFF has the potential to provide instant warning and positive feedback to the introduction and elimination of design flaws. This enables the long-term health of software design against rapid code revisions.

RQ3: What is the performance overhead of DESIGNDIFF? We track the execution time of DESIGNDIFF, covering Diff Extractor, DiffDSM Generator, and Operator Matcher and Visualizer, on commits with $<= 5$, (5, 10], (10, 15], and > 15 source files. The profiling data show that the average processing time is 13, 15, 19, 29 seconds for the four complexity levels respectively. Large portions of time were consumed on Diff Extractor and Operator Visualizer. The Diff Extractor could end up scanning a large amount of the code for even a small change due to a large number of referenced files. In addition, the Visualizer relies on an open source library [52] to render different operators in a .xlsx file. Overall, the overhead of DESIGNDIFF is affordable for running after every code change. We plan to further optimize DESIGNDIFF in the future.

VII. RELATED WORK

a) Code Differencing

Numerous prior work focused on analyzing code differences in a single code change. They are in three types: 1) Text-based approach that computes the inserted, removed, or changed lines of code between two versions of a source file [53]–[57]; 2) Tree-based approaches that capture the syntactic difference in the format of Abstract Syntax Trees [58]–[65]; and 3) Graph-based approach that uses graph representations of source code, such as control flow graph, to shown program semantic change [66]–[70]. In a particular note, Apiwattanapong et. al. [66] analyzed the semantic changes of OO programs, which are relevant to the polymorphism related operators in this paper. Their approach outputs a control-flow graph of the internal structure of a method, composed of an entry point, method invocations, variable declarations, and branching points, such as try and catch, and finally an exiting point. They employed dynamic binding to identify method invocations that may affect different part of the program at run-time due to the polymorphism design. In comparison, our work focused on capturing the changes of the inter-dependencies among existing/added files/classes, and group related changes into operators, such as *Augmenting an Partial PolyMorphsim Tree*, to facilitate the comprehension of the high-level design.

b) Design Differencing

Previous work has also focused on identifying design difference between two versions of software program. Our work is highly relevant to [71]–[74], where the design differences are also described in the notion of change operators. In [73], the authors identified *add parent/base class* and *remove parent/base class*. They focused on just one particular parent or base class being added or removed, without capturing the overall polymorphism structure formed by a group of files. In [74], the authors described the *extract super class* and *flatten polymorphism* changes. The *extract super class* is similar to *add a parent/base class*, and it focuses on cases where a super class is extracted from an existing class. The *flatten polymorphism* is the opposite of *extract super class*, where a parent class and its child are combined into one flat class. In [71], [72], the authors capture the general *inheritance/interface change* similar to the previous two studies. Meanwhile, these related work also identify fine-grained changes, such as *change visibility* [71], [72], *change attribute* [73], [74], which are not captured in this work. The uniqueness of our work is two-fold. First, the polymorphism operators (i.e. operators 1 to 4, 6 to 9, 11, and 14) stand for different high-level design

decisions, which are not formally differentiated in previous work. For example, *Augmenting an Entire Polymorphism Tree* aggregates a group of newly added files that form an completely new inheritance tree, which indicates the employment of polymorphism for addressing a complicated aspect of design. Similarly, *Augmenting Partial Polymorphism Tree* groups related files that expand an existing polymorphism tree, which indicates the increase in the complexity of an existing polymorphism structure. In prior work, these two scenarios are not automatically captured, instead they will be illustrated as multiple add-parent and add-class operators. Similarly, each of the other polymorphism operators has a unique meaning that is not specifically differentiated in related work. Second, our approach is designed for continuous modeling. It performs a light-weighted comparison of the before and the after change DSMs composed of only the change-related files. The strength is that, as shown in RQ2 and RQ3, it can be used to detect the introduction and elimination of design flaws from every single commit with affordable cost. In comparison, previous approaches are designed for analyzing two remote versions of code using comprehensive code analysis and heuristics for mapping elements and extracting the fine-grained code elements, such as attributes and methods. The advantage of related work is to captured a combination of fine-grained and design structure changes, but they may not be suitable for the continuous monitoring evaluated in this paper.

c) Architecture Recovery and Analysis

Architecture recovery is the process of reverse-engineering system architecture from binary code, source code, or configuration files. Numerous techniques and tools were built to automatically group implementation entities, e.g. files, classes, or functions, into high level architectural elements, such as components/connectors [10]–[18], modules [75]–[77], and design spaces [22], [36], [37]. Architecture visualization facilitates the high-level understanding [78]–[81]. Another important goal of architecture recover is to identify architectural decay and flaws [20], [45], [82]–[91]. To a very limited extend, prior work focused on the analysis of architecture evolution [23], [71], [92]–[95]. However, they mostly work at consecutive versions of a software system and are limited to provide timely guidance for developers after every single code change. According to a recent study [96], developers need improved awareness of the architectural changes in the daily development activities. Our work complements existing literature by contributing a continuous design change modeling and monitoring approach.

VIII. Threats to Validity and Limitations

a) Threats to Validity

First, we defined the 17 change operators based on the different patterns of dependency change among existing/added source files. We cannot guarantee that these 17 operators can correctly and completely capture the actual intentions of developers who made the changes. Second, we acknowledge the potential basis that we only compared with GitDiff, which is not intended for analyzing software design. Third, we acknowledge the potential basis in grading the design difference summaries submitted by the study participants due to individual understanding and experience. Fourth, 10 participants in RQ1 are currently graduate students who have 1-6 years of real-world system development experience. And the 20 commits in the study may not cover all possible change scenarios. We plan to conduct more comprehensive user studies in the future. Fifth, DesignDiff employs heuristics to order and visualize the design change operators (see Section IV). We acknowledge it is not optimal for all different commits. In future work, we plan to customize the ordering strategy based on the content of each code change, such as by grouping operators involving the same set of files together.

b) Limitations

First, we acknowledge that the time saved by using DesignDiff compared to using GitDiff is not statistically significant for all the participants. Only 60% participants were able to save time by using DesignDiff. In future work, we plan to simplify the visualization to make it is easier to understand. Second, we acknowledge that we did not conduct statistic analysis on the survey data of RQ1, given the limited number of participants. Third, DesignDiff focuses on only four operators: *Augmenting*, *Excluding*, *Importing (Deporting)*, and *Inverting*, of the original six operators. To clarify: *Inverting* is captured as polymorphism related *Augmenting*. However, *Splitting* and *Substituting* are not directly captured, but can be captured by combinations of *Augmenting*, *Excluding*, *Importing* and *Deporting*. We plan to investigate the dynamic combination of the 17 operators in future work. Fourth, we acknowledge that the four design flaws studied in RQ2 are not complete to cover all possible flaws in practice. However, practitioners can leverage DesignDiff to define their own flaw detection rules based on project needs. Fifth, the cyclic dependencies identified by DesignDiff only involve files within the Diff-DSM of a commit. Cycles formed by both files in and out of a Diff-DSM cannot be calculated without the knowledge of a full system DSM. We plan to explore efficient ways to identified the latter type of cycles in the future.

IX. Conclusion

This paper contributed a new design modeling and analysis approach, DesignDiff, to incrementally capture, interpret, and visualize the essential design structure difference resulting from every single code change. The evaluation on three application contexts proved the great potential of DesignDiff. A user study of 10 participants and 20 real-life software commits indicated that DesignDiff can help practitioners to better and faster understand the impacts of detailed code changes to the software design structure. Evaluation on 14,832 real-life software commits proved that DesignDiff can enable continuous design structure monitoring by providing instant warning and positive feedback to design flaw introduction and elimination.

Acknowledgements

This work was supported in part by the National Science Foundation of the US under grants CCF-1823074.

REFERENCES

[1] Ken Schwaber and Mike Beedle. *Agile software development with Scrum*, volume 1. Prentice Hall Upper Saddle River, 2002.

[2] Robert C Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[3] Alistair Cockburn. *Agile software development: the cooperative game*. Pearson Education, 2006.

[4] Alistair Cockburn and Jim Highsmith. Agile software development: The people factor. *Computer*, (11):131–133, 2001.

[5] Agile Alliance. Agile manifesto. *Online at http://www. agilemanifesto. org*, 6(1), 2001.

[6] Lakshitha de Silva and Dharini Balasubramaniam. Controlling software architecture erosion: A survey. *J. Syst. Softw.*, 85(1):132–151, January 2012.

[7] David Saff and Michael D Ernst. An experimental evaluation of continuous testing during development. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 76–85. ACM, 2004.

[8] Sean Stolberg. Enabling agile testing through continuous integration. In *2009 Agile Conference*, pages 369–374. IEEE, 2009.

[9] David Talby, Arie Keren, Orit Hazzan, and Yael Dubinsky. Agile software testing in a large-scale project. *IEEE software*, 23(4):30–37, 2006.

[10] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. A comparative analysis of software architecture recovery techniques. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 486–496, 2013.

[11] V. Tzerpos and R.C. Holt. ACDC: an algorithm for comprehension-driven clustering. In *Working Conference on Reverse Engineering (WCRE)*, 2000.

[12] Brian S. Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE TSE*, 2006.

[13] N. Anquetil and T. Lethbridge. File clustering using naming conventions for legacy systems. In *Conference of the Centre for Advanced Studies on Collaborative Research*, 1997.

[14] N. Anquetil and T.C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice*, 1999.

[15] Joshua Garcia, Daniel Popescu, Chris Mattmann, Nenad Medvidovic, and Yuanfang Cai. Enhancing architectural recovery using concerns. In *ASE*, 2011.

[16] Anna Corazza, Sergio Di Martino, and Giuseppe Scanniello. A probabilistic based approach towards software system clustering. In *European Conference on Software Maintenance and Reengineering (CSMR)*, 2010.

[17] Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello. Investigating the use of lexical information for software system clustering. In *European Conference on Software Maintenance and Reengineering (CSMR)*, 2011.

[18] Janardan Misra, KM Annervaz, Vikrant Kaulgud, Shubhashis Sengupta, and Gary Titus. Software clustering: Unifying syntactic and semantic features. In *Working Conference on Reverse Engineering (WCRE)*, 2012.

[19] Ran Mo, Yuanfang Cai, Rick Kazman, and Lu Xiao. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*, pages 51–60. IEEE, 2015.

[20] Sunny Wong, Yuanfang Cai, Miryung Kim, and Michael Dalton. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 411–420. ACM, 2011.

[21] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. Architecture anti-patterns: Automatically detectable violations of design principles. *IEEE Transactions on Software Engineering*, 2019.

[22] Robert Schwanke, Lu Xiao, and Yuanfang Cai. Measuring architecture quality by structure plus history analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 891–900. IEEE Press, 2013.

[23] Lu Xiao, Yuanfang Cai, Rick Kazman, Ran Mo, and Qiong Feng. Identifying and quantifying architectural debt. In *Proceedings of the 38th International Conference on Software Engineering*, pages 488–498. ACM, 2016.

[24] Gábor Szőke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. Empirical study on refactoring large-scale industrial systems and its effects on maintainability. *Journal of Systems and Software*, 129:107–126, 2017.

[25] Ewan Tempero, Tony Gorschek, and Lefteris Angelis. Barriers to refactoring. *Communications of the ACM*, 60(10):54–61, 2017.

[26] https://github.com/.

[27] https://www.apache.org/.

[28] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[29] Robert C Martine. Design principles and design patterns. 2000.

[30] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse engineering approach to subsystem structure identification, 1993.

[31] Rainer Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: A research survey. *Journal of Software Maintenance*, 15(2):87–109, March 2003.

[32] https://scitools.com/.

[33] Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity Volume 1*. MIT Press, Cambridge, MA, USA, 1999.

[34] Y. Cai and K. J. Sullivan. Modularity analysis of logical design models. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 91–102, Sep. 2006.

[35] A. Avritzer, D. Paulish, and Y. Cai. Coordination implications of software architecture in a global software development project. In *Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, pages 107–116, Feb 2008.

[36] Lu Xiao, Yuanfang Cai, and Rick Kazman. Design rule spaces: A new form of architecture insight. In *Proceedings of the 36th International Conference on Software Engineering*, pages 967–977. ACM, 2014.

[37] Yuanfang Cai, Lu Xiao, Rick Kazman, Ran Mo, and Qiong Feng. "design rule spaces: A new model for representing and analyzing software architecture". *IEEE Transactions on Software Engineering*, Accepted.

[38] Frederick P. Brooks, Jr. *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[39] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[40] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 192–201, New York, NY, USA, 2014. ACM.

[41] Philippe Kruchten, Robert L. Nord, Ipek Ozkaya, and Davide Falessi. Technical debt: Towards a crisper definition report on the 4th international workshop on managing technical debt. *SIGSOFT Softw. Eng. Notes*, 38(5):51–54, August 2013.

[42] *TechDebt '18: Proceedings of the 2018 International Conference on Technical Debt*, New York, NY, USA, 2018. ACM.

[43] Tosin Daniel Oyetoyan, Daniela S. Cruzes, and Reidar Conradi. A study of cyclic dependencies on defect profile of software components. *J. Syst. Softw.*, 86(12):3162–3182, December 2013.

[44] Heiko Koziolek. Sustainability evaluation of software architectures: A systematic review. In *Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS*, QoSA-ISARCS '11, pages 3–12, New York, NY, USA, 2011. ACM.

[45] Ran Mo, Yuanfang Cai, R. Kazman, and Lu Xiao. Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells. In *2015 12th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 51–60, May 2015.

[46] Jules Moloney and Rajaa Issa. Materials in architectural design education software: A case study. *International Journal of Architectural Computing*, 1(1):46–58, 2003.

[47] https://avro.apache.org/.

[48] https://github.com/.

[49] https://jena.apache.org/.

[50] https://mina.apache.org/.

[51] https://pdfbox.apache.org/.

[52] https://poi.apache.org/.

[53] Eugene W Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[54] Webb Miller and Eugene W Myers. A file comparison program. *Software: Practice and Experience*, 15(11):1025–1040, 1985.

[55] Muhammad Asaduzzaman, Chanchal K Roy, Kevin A Schneider, and Massimiliano Di Penta. Lhdiff: A language-independent hybrid approach

for tracking source code lines. In *2013 IEEE International Conference on Software Maintenance*, pages 230–239. IEEE, 2013.

[56] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Tracking your changes: A language-independent approach. *IEEE Software*, 26(1):50–57, 2009.

[57] Steven Reiss. Tracking source locations. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 11–20. IEEE, 2008.

[58] Sudarshan S Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Acm Sigmod Record*, volume 25, pages 493–504. ACM, 1996.

[59] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324. ACM, 2014.

[60] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering*, 33(11):725–743, 2007.

[61] Beat Fluri and Harald C Gall. Classifying change types for qualifying change couplings. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 35–45. IEEE, 2006.

[62] Masatomo Hashimoto and Akira Mori. Diff/ts: A tool for fine-grained structural change analysis. In *2008 15th Working Conference on Reverse Engineering*, pages 279–288. IEEE, 2008.

[63] G. Dotzler and M. Philippsen. Move-optimized source code tree differencing. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 660–671, Sep. 2016.

[64] Yoshiki Higo, Akio Ohtani, and Shinji Kusumoto. Generating simpler ast edit scripts by considering copy-and-paste. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 532–542, Piscataway, NJ, USA, 2017. IEEE Press.

[65] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. COMPUT*, 18(6), 1989.

[66] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. A differencing algorithm for object-oriented programs. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ASE '04, pages 2–13, Washington, DC, USA, 2004. IEEE Computer Society.

[67] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 234–245, New York, NY, USA, 1990. ACM.

[68] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: a semantic-graph differencing tool for studying changes in large code bases. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 188–197, Sep. 2004.

[69] Jackson and Ladd. Semantic diff: a tool for summarizing the effects of modifications. In *Proceedings 1994 International Conference on Software Maintenance*, pages 243–252, Sep. 1994.

[70] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 712–717, Berlin, Heidelberg, 2012. Springer-Verlag.

[71] Zhenchang Xing and Eleni Stroulia. Umldiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 54–65, New York, NY, USA, 2005. ACM.

[72] Zhenchang Xing and Eleni Stroulia. Differencing logical uml models. *Automated Software Engineering*, 14(2):215–259, 2007.

[73] Rimon Mikhaiel, Nikolaos Tsantalis, Natalia Negara, Eleni Stroulia, and Zhenchang Xing. Differencing uml models: a domain-specific vs. a domain-agnostic method. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 159–196. Springer, 2011.

[74] Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, Jacques Robin, and Marie-Pierre Gervais. Detecting complex changes and refactorings during (meta) model evolution. *Information Systems*, 62:220–241, 2016.

[75] Sunny Wong, Yuanfang Cai, Giuseppe Valetto, Georgi Simeonov, and Kanwarpreet Sethi. Design rule hierarchies and parallelism in software development tasks. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 197–208. IEEE Computer Society, 2009.

[76] Yuanfang Cai and Sunny Wong. Design rule hierarchy, task parallelism, and dependency analysis in logical decision models, August 30 2011. US Patent App. 13/819,136.

[77] Lu Xiao and Tingting Yu. Ripple: a test-aware architecture modeling framework. In *Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering*, pages 14–20. IEEE Press, 2017.

[78] A. P. Sawant and N. Bali. Diffarchviz: A tool to visualize correspondence between multiple representations of a software architecture. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 121–128, June 2007.

[79] S. Duszynski, J. Knodel, and M. Lindvall. Save: Software architecture visualization and evaluation. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 323–324, March 2009.

[80] Keith Gallagher, Andrew Hatch, and Malcolm Munro. Software architecture visualization: An evaluation framework and its application. *IEEE Trans. Softw. Eng.*, 34(2):260–270, March 2008.

[81] Zohreh Sharafi. A systematic analysis of software architecture visualization techniques. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 254–257, Washington, DC, USA, 2011. IEEE Computer Society.

[82] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *FSE*, 1995.

[83] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Toward a catalogue of architectural bad smells. In *QoSA '09: Proc. 5th Int'l Conf. on Quality of Software Architectures*, 2009.

[84] Joshua Garcia, Daniel Popescu, George Edwards, and Medvidovic Nenad. Identifying Architectural Bad Smells. In *13th European Conference on Software Maintenance and Reengineering*, 2009.

[85] Joshua Garcia. *A unified framework for studying architectural decay of software systems*. PhD thesis, University of Southern California, 2014.

[86] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and A Le Meur. Decor: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, 36(1):20–36, 2010.

[87] John B. Tran and Richard C. Holt. Forward and Reverse Repair of Software Architecture. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 12–, Mississauga, Ontario, Canada, 1999. IBM Press.

[88] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, 8(1):89–120, 2001.

[89] Jan Philipps and Bernhard Rumpe. Refinement of information flow architectures. In *Formal Engineering Methods., 1997. Proceedings., First IEEE International Conference on*, pages 203–212. IEEE, 1997.

[90] Tom Mens and Tom Tourwé. A survey of software refactoring. *Software Engineering, IEEE Transactions on*, 30(2):126–139, 2004.

[91] Igor Ivkovic and Kostas Kontogiannis. A Framework for Software Architecture Refactoring Using Model Transformations and Semantic Annotations. In *Proceedings of the Conference on Software Maintenance and Reengineering*, CSMR '06, pages 135–144, Washington, DC, USA, 2006. IEEE Computer Society.

[92] E. Kouroshfar. Studying the effect of co-change dispersion on software quality. In *ACM Student Research Competition, 35th International Conference on Software Engineering (ICSE)*, pages 1450–1452, San Francisco, CA, May 2013.

[93] Duc Le, Pooyan Behnamghader, Joshua Garcia, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. An empirical study of architectural change in open-source software systems. *To appear in the 12th Working Conference on Mining Software Repositories*, 2015.

[94] Rick Kazman, Yuanfang Cai, Ran Mo, Qiong Feng, Lu Xiao, Serge Haziyev, Volodymyr Fedak, and Andriy Shapochka. A case study in locating the architectural roots of technical debt. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 179–188. IEEE Press, 2015.

[95] Architectural decay prediction. https://seal.ics.uci.edu/projects/decayprediction/, 2016.

[96] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. Are developers aware of the architectural impact of their changes? In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 95–105, Piscataway, NJ, USA, 2017. IEEE Press.