Software Evolution

Na Meng, Tianyi Zhang, Miryung Kim

Virginia Tech and University of California, Los Angeles

Abstract. Miryung is in charge.

1 Introduction

2 page - explain the definition of software evolution (cite: belady and lehman, etc) - describe why this chapter focus on code changes, rather than other types of artefacts such as requirements, specifications, design documents, etc. - argue why software evolution is important (cite: code decay, eick et al.)

2 Concepts and Principles

4 page - explain a broad category of changes: corrective, adaptive, and perfective changes (cite kemerer and slaugher) - include a diagram about the process of software evolution with focus on changes: (A) applying program changes (B) inspecting program changes, and (C) debugging and testing program changes to overview the rest of sections. - introduce topics under each of the three perspectives on program changes.

3 An Organized Tour of Seminal Papers: I. Applying Program Changes

make the following text a little bit less like compiler optimization and small program transformation. start from three broad categories of changes. Program transformation is the process to transform a program to another program. A typical example is compilers. As shown in Figure 1, a Java compiler transforms Java source code into Java bytecode, an intermediate representation of the program. The bytecode can be either interpreted by a bytecode interpreter to directly execute on Java Virtual Machine (JVM), or further transformed by a Just-in-Time (JIT) compiler to machine code for faster execution.

Miryung to Na, could you please update the figure so that we can discuss program transformations in a broader sense?

Adapt the following text to meet the new outline / categorization. Also make the text not to just focus on automated transformation, but include other empirical studies on code changes, and how people make such changes manually.

Manually transforming programs is tedious and error-prone, because people make mistakes when manipulating code to enforce the explicit or implicit

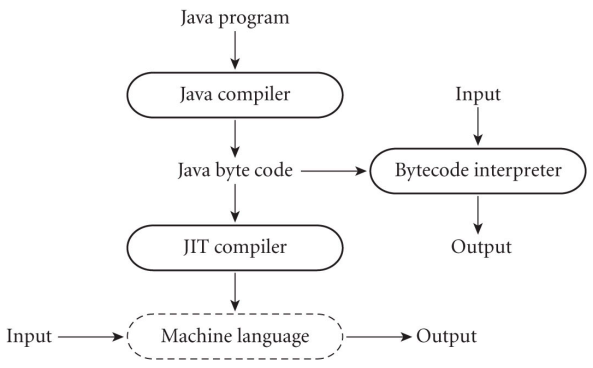


Fig. 1. Java compilation system

transformation rules. Various automatic systems have been built to transform programs in a mechanic way. Such systems are built for different purposes. For instance, a source-to-source compiler translates programs from one programming language to another language. Source code generation produces source code based on an ontological model, which defines the types, properties, and interrelationships of entities. Many systems guarantee that each transformed program is semantically equivalent to the original program, while other systems do not preserve such semantic equivalence while transforming code. In this section, we will first overview various methods, techniques, and tools to automate source-to-source program transformation (Section **??**), and then especially discuss refactoring—one kind of semantic-preserving transformation (Section **??**), and systematic editing—one kind of transformation that does not preserve semantics (Section 3.3).

3.1 Perfective Changes

Miryung: Write the following paragraph more broadly. refactoring is a special category of automated transformation: refactoring practices (kim et al., emerson murphy hill, ralph johnson)

Code refactoring is the process of restructuring source code without changing its external behaviors. It is usually applied to improve code readability or extensibility, or to reduce code complexity. Fowler et al. defined a catalog of refactorings that developers can apply to improve their codebase in different ways [**?**], although developers can also define and manually apply their own refactorings. Eclipse IDE also provides tool support to automate some of the refactorings mentioned in Fowler's catalog, such as *Extract Method*, *Pull up Method*, and *Move Field*. Researchers proposed various approaches to automate refactoring or to complete the refactoring tasks initiated by developers [**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**]. Start from Bill Griswold and Okdype, Ralph Johnson

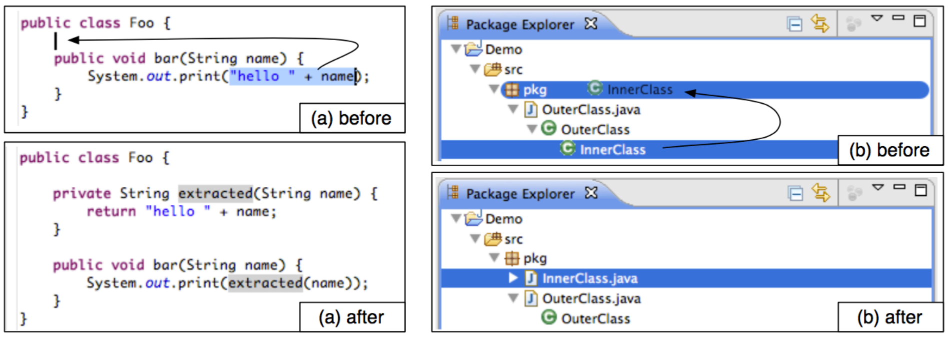
To ensure the semantic equivalence between programs that are before and after a predefined refactoring program transformation, automated tools always check some pre-conditions before applying the transformation, and may check some post-conditions after the transformation. In this section, we will explain two refactoring tools with more details: Drag-and-Drop Refactoring (DNDRefactoring) [?] and R3 [?].

Miryugn to Na: we should discuss a survey of refactoring papers by Mens Miryung to Na: DND refactoring does not seem like a seminal paper. Reduce the space allocation of this example section to a half page as a recent technique after discussing some seminal papers first.

Drag-and-Drop Refactoring. Prior research identified at least three dominant usability problems when using automated refactoring tools [?,?,?,?,?,?]. First, programmers have trouble identifying opportunities for using the tools. Second, programmers have difficulty invoking the right refactoring from a lengthy menu of available refactorings. Third, programmers find it complicated to properly configure refactoring dialogs. DNDRefactoring was proposed to facilitate automated refactoring by allowing programmers to apply refactorings through direct manipulation on program elements, including variables, expressions, statements, and methods, in the IDE. In this way, developers do not need menus or dialogs to specify what refactoring to apply or how to apply those refactorings. Instead, DNDRefactoring can automatically infer such information by monitoring the selection and drag-and-drop operations of developers.

Figure 2 presents two scenarios where DNDRefactoring automates refactorings based on the gestures of developers in Eclipse IDE. As shown in Figure 2 (a), when developers select a code snippet in an existing method `bar(...)`, and then drag-and-drop the snippet to a location outside `bar(...)`, DNDRefactoring automates the *Extract Method* refactoring by creating a new method `extracted(...)`, which takes in one parameter: `name`. In this process, no menu or configuration dialog is needed, because the tool can infer developers' refactoring intent, and decide all detailed information required for the refactoring, such as the method name, the method body, and the code location where to place the new method. Figure 2 (b) shows another scenario. When developers manually select the `InnerClass`, a class declared inside another class `OuterClass`, and then drag-and-drop the class to a package `pkg`, DNDRefactoring infers that developers want to extract the type and create a new Java file. Therefore, it generates a Java file InnerClass.java to hold all implementation of `InnerClass`.

Although DNDRefactoring demonstrated effectiveness in simplifying the application of automated refactorings, it still suffers from several limitations. First, not every refactoring can be conducted in a drag-and-drop manner. DNDRefactoring effectively supports move- and extract-based refactorings, but does not support *Rename* refactoring. If developers want to rename a variable or a class, they cannot express that intent via selecting some text and moving it around with the cursor. Second, even for move- or extract- based refactorings, DNDRefactoring mainly works when the drag source and drop target elements are shown in the same screen. It does not work when these elements are located too far away to be presented in the same visual editor simultaneously. Third, DNDRefactoring does not allow users to freely configure refactoring details. For example,



```
public class Foo {
    |
    public void bar(String name) {
        System.out.print("hello " + name);
    }
}
```

```
public class Foo {

    private String extracted(String name) {
        return "hello " + name;
    }

    public void bar(String name) {
        System.out.print(extracted(name));
    }
}
```

Fig. 2. DNDRefactoring: Drag-and-drop gestures in (a) Java editor for Extract Method refactoring, and (b) Package Explorer for Extract Type to New File refactoring [?].

when users want to specify a meaningful name of an extracted method, with DNDRefactoring, they have no way to customize the information.

Miryung to Na. R3 does not seem like a seminal paper. Similarly, I will reduce the space to a half page

R3. Compared with DNDRefactoring which improves the usability of Eclipse Refactoring with better UIs, R3 presents an alternative refactoring engine that works 10 times faster than Eclipse Refactoring. Specifically, R3 provides refactoring scripts as short Java methods, which enables users to easily define new refactorings. To speed up the application of refactorings, R3 first parses Java sources files, and builds a main-memory, non-persistent database to encode Java entity declarations (e.g., packages, class, methods), their containment relationships, and language features such inheritance and modifiers. By converting program Abstract Syntax Tree (AST) transformations to database queries and manipulations, R3 significantly reduces the runtime overhead of automating refactorings.

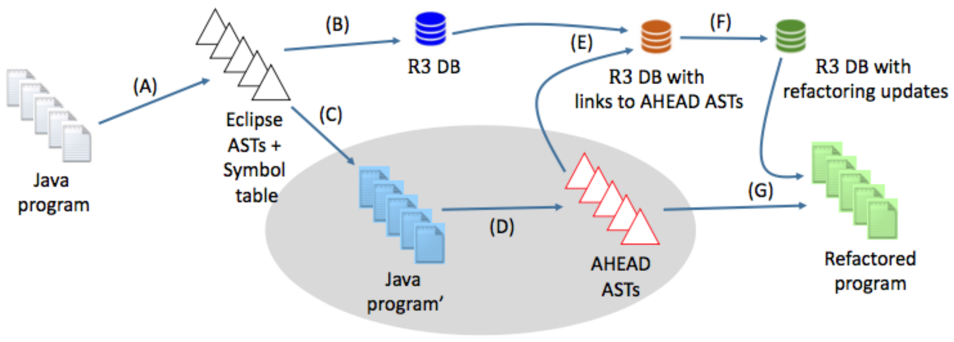

Java program
(A)
Eclipse ASTs + Symbol table
(B)
R3 DB
(C)
Java program'
(D)
AHEAD ASTs
(E)
R3 DB with links to AHEAD ASTs
(F)
R3 DB with refactoring updates
(G)
Refactored program

Fig. 3. R3 pipeline [?].

Figure 3 presents the R3 pipeline, which has a series of stages (A)-(G) that map a target Java program (JDT project) on the left to a refactored program on the right. Stage (A) parses a Java program into ASTs with an Eclipse ASTParser. Stage (B) visits each generated AST to gather all Java entity-relevant information, and to save the information in R3 DB. To prepare program refactorings based on R3 DB, stage (C) generalizes the original program by replacing concrete entity information with abstract symbolic names, with each name corresponding to one concrete entity identifier kept in R3 DB. Stage (D) uses AHEAD [?] to further parse the generalized program representation and create AHEAD ASTs. In Stage (E), tuples in R3 DB are doubly-linked to their AHEAD AST nodes, so that each pretty-printer of an AST node can reference the corresponding R3 tuple and vice versa.

Stage (F) execute R3 refactorings. Unlike classical refactoring engines that modify Abstract Syntax Trees (ASTs), R3 refactorings modify only the database. For instance, if developers want to move a method from one class to another class, R3 does not directly move the method AST. Instead, it simply updates the method's database entry to have the new receiver type. With such information updates to the database, R3 can avoid repetitive AST manipulations. Finally, according to the updated entity information in R3 DB, stage (G) pretty-prints the source code, producing the resulting refactored program.

In addition to converting expensive AST manipulations to cheap database updates, R3 also precomputes the value of many properties of Java entities and saves those values in its database. Since these values can be used int the precondition checks of many refactoring tasks, keeping them in the database can avoid repetitive value computations, which can further reduce runtime overhead. R3 can well handle move- and rename-based refactorings, but does not support extract-based refactorings, because it does not store or modify statement-level or expression-level information in the database.

3.2 Corrective Changes

- discuss literature on bug fixes more empirical studies by S. Kim et al and T. Nguyen et al specialize bug fix techniques by Y.Y Zhou, S. Lu

3.3 Additive Changes

discuss literature on adding features, (systematic editing is a specialized technique for automated feature addition) source to source transformation

add some literature on cross cutting changes. P. Tarr

Source to Source Transformation TXL is a source transformation language designed to translate or manipulate programming languages [?]. Given a context-free grammar to describe program syntax, and a set of transformation rules to manipulate the syntax, TXL automatically transforms programs of the syntactic structure by applying those rules. With TXL, developers to not need to build programming language translators by coding every line of implementation. Instead, the TXL program transformation engine can automatically translate code once developers specify all needed grammars and rules. Researchers built tools to perform various code translation tasks using TXL [?,?,?,?]. For instance, Hassan et al. migrated web applications between different web development frameworks like ASP and NSP [?], while El-Ramly et al. converted Java programs to C# [?].

TXL is a representative technique. So add some example rule representation here.

mppSMT is the most recent code translation tool, which automatically infers and applies Java-to-C# migration rules using a phrase-based statistical machine translation approach [?]. It encodes both Java and C# source files into sequences of syntactic symbols, called syntaxemes, and then relies on the syntaxemes to align code and to train a sequence-to-sequence translation model. Unlike TXL which requires users to specify mapping rules, mppSMT automatically infers those rules from the corresponding implementations in both languages. However, this approach depends on the highly similar program syntaxes between Java and C#. It also requires tool builders to manually define rules to encode program syntactic components to syntaxemes.

Systematic Editing The following is moved from program synthesis section. I think we start broadly in systematic editing techniques and go deeper in sydit and lase perhaps reduce the space allocation on systematic editing to 2 pages. Right now, there's too much emphasis on this?

Simultaneous editing repetitively applies source code changes that are interactively demonstrated by users [?]. When users apply their edits in one program context, the tool replicates the *exact lexical* edits to other code fragments, or transforms code accordingly. For instance, Linked Editing requires users to first specify the similar code snippets which they want to modify in the same way [?]. As users interactively edit one of these snippets, Linked Editing simultaneously applies the identical edits to other snippets. CloneTracker takes the output of a clone detector as input and creates a descriptor for each clone [?]. With such descriptors, CloneTracker tracks clones across program versions and identifies any modification to those clones. Similar to Linked Editing, CloneTracker also echoes edits in one clone to other counterparts upon a developer's request. Clever is another clone management system that tracks code clone groups and detects any inconsistent change applied to clones within the same group [?]. If a clone misses the updates applied to the other clones in the same group, Clever automatically suggests the missing update to that clone.

Systematic editing is the process of applying similar, but not necessarily identical, program changes to multiple code locations. Prior work shows that programmers apply systematic edits to either add features, fix bugs, or refactor code [?,?,?]. Manually applying similar but different edits to multiple code locations is tedious and error-prone for two reasons. First, developers may forget to apply systematic edits to all program contexts where the edits are needed, committing errors of omission. Second, developers may apply edits inconsistently and thus introduce new bugs. To improve programmer productivity and software quality, several approaches [?,?,?] have been proposed to infer the general program transformation from one or more code change examples provided by developers, and then apply the transformation to other program contexts in need of similar changes. In this section, we will explain one approach with more details—SYDIT [?], and briefly discuss another approach: LASE [?].

SYDIT. When developers want to change multiple code locations in similar but not identical ways, SYDIT requires developers (1) to modify one of these code locations and present it as a code change example, and (2) to manually specify the other locations to change similarly. By inferring a program transformation from the given example, SYDIT can apply the transformation to those locations and automate systematic editing accordingly.

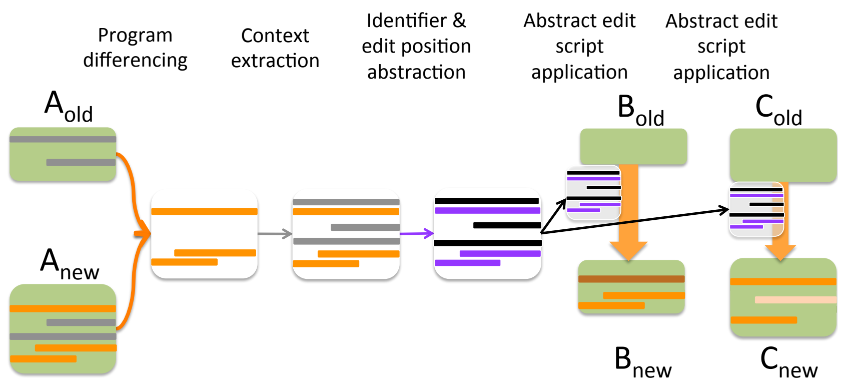

Program differencing
Context extraction
Identifier & edit position abstraction
Abstract edit script application
Abstract edit script application
A_{old}
A_{new}
B_{old}
C_{old}
B_{new}
C_{new}

Fig. 4. SYDIT overview.

As shown in Figure 4, suppose developers want to apply systematic edits to three methods: A, B, and C. Without any tool support, developers have to manually apply these similar but different edits to all three locations. With SYDIT, developers only need to (1) modify one location (e.g., A), and (2) manually specify the other locations to change (e.g., B and C). With the user input, SYDIT takes four steps to generalize and apply a program transformation. In the first step (program differencing), given the old and new version of method A, SYDIT first leverages ChangeDistiller [?] to compare the ASTs of both versions, and to extract changes as an AST edit script. The edit script may involve four types of statement-level edit operations: insert, delete, update, and move. To generalize the concrete edit script to a program transformation that is applicable to other program contexts, SYDIT needs to abstract the edit context and concrete identifiers. In step 2, SYDIT abstracts the edit context by identifying all unchanged code that is either control- or data-dependent on by the edited code, because such unchanged code manifests the semantic constraints those applied edits put on the program context. In step 3, SYDIT also abstracts the concrete identifiers used in the demonstrated edit so that the general program transformation is also applicable to code snippets that use different identifiers. Besides, SYDIT abstracts all edit operations' positions with respect to the extracted unchanged code, so that the inferred program transformation describes each edit operation within the edit-relevant context. In this way, SYDIT derives an abstract, context-aware edit script from a given code change example.

In step 4, SYDIT concretizes the inferred program transformation to given code locations B and C, and applies the customized edits to suggest new versions of code. This step is the reverse process of the steps 1-3 mentioned above. With more details, given a code location to change (e.g., B), SYDIT first looks for any matching for the abstract edit context. If no matching is found, it indicates that similar changes should not be applied to the selected location, because the semantic constraints embedded in the abstract context are not satisfied. If one matching is found, SYDIT further concretizes the identifier usage and edit location information in the abstract script, creating a customized edit script applicable to the specific location. Finally, by modifying the code's AST based on the script, SYDIT produces a revised program for developers to review.

Although SYDIT can automate systematic editing in specified code locations, it always relies on users to manually pick those locations. In reality, finding code locations may be more challenging than applying the edits, especially when the codebase is large, and developers lack expertise of the project. Additionally, one code change example may not precisely characterize all program contexts which should be changed similarly. If the single example contains some edit-relevant context that is unique to the code location, SYDIT has no clue about how to only generalize the edit-relevant context which is commonly shared among all code locations.

LASE. To facilitate systematic editing when developers cannot manually identify all code locations that should be changed similarly, LASE requires developers to provide at least two code change examples, from which it infers a general program transformation, and then leverages the transformation to both find other edit locations and suggest customized edits.

As shown in Figure 5, given two changed examples: A and B, LASE infers a general program transformation for each example, and then extracts the largest commonality between them in terms of edit operations, edit-relevant contexts, and identifier usage. In this way, LASE filters out any location-specific edit information, and only generalizes the program transformation that occurs multiple times. With such inferred program transformation, LASE attempts to establish matching between the abstract edit context and every method in the whole codebase. If a method contains a matching to the given context (e.g., C), LASE recommends the method as a candidate edit location, and suggests customized edits according to the matching information. If a method does not contain any matching to the given context (e.g., D), LASE considers the method as an irrelevant method to the systematic editing task.

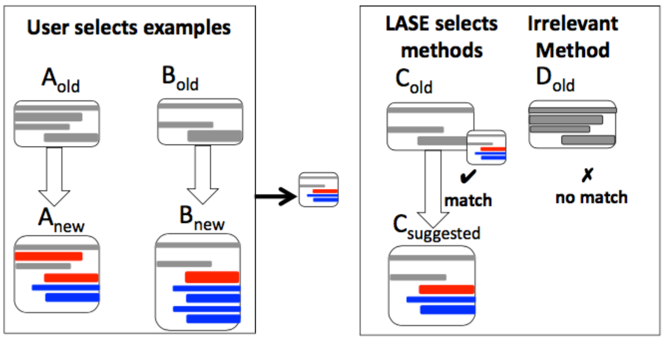


Fig. 5. LASE overview [?].

3.4 Programming by Demonstration (PbD).

Condense PBD to 1 page or less It is also called programming by example (PbE) [?]. This is an end-user development technique for teaching a computer or a robot new behaviors by demonstrating the task to transfer directly instead of programming it through machine commands.

Various approaches were built to generate programs based on the text-editing actions demonstrated or text change examples provided by users [?,?,?,?]. For instance, TELS records editing actions, such as search-and-replace, and generalizes them into a program that transforms input to output [?]. It leverages heuristics to match actions against each other to detect any loop in the user-demonstrated program. Similarly, SMARTedit [?] automates repetitive text-editing tasks by learning programs to perform them using techniques drawn from machine learning. SMARTedit represents a text-editing program as a series of functions that alter the state of the text editor (i.e., the contents of the file, or the cursor position). Like macro recording systems, SMARTedit learns the program by observing a user performing her task. However, unlike macro recorders, SMARTedit examines the context in which the user's actions are performed and learns programs that work correctly in new contexts.

Program synthesis is the task of generating a program in a domain-specific language from a given specification using some search techniques [?]. For instance, Gulwani defined a string manipulation language that supports restricted forms of regular expressions, conditionals, and loops, and also defined an algorithm to synthesize a program in this language from input-output string examples [?]. The algorithm was implemented as an interactive add-in for Microsoft Excel spreadsheet system. Each time when users manipulate a string and provide both the before- and after- versions of the manipulation, the algorithm treats the input-output example as a constraint that a desired program should satisfy, and searches for all candidate programs that can produce the output given the input. As users provide more input-output examples, the algorithm filters out the candidates that partially satisfy some of the constraints, until finding the program which satisfies all input-output constraints. Such techniques were also used in other problem domains, such as table transformation in Excel spreadsheets [?], geometry construction [?], data structure transformations [?], and hierarchical structured data manipulation [?].

3.5 Porting programs to different languages

Split the following text into different languages vs. platforms

Developers translate code from one language to another language for various reasons. For instance, when maintaining a legacy system that was written in Fortran decades ago, programmers may migrate the system to a mainstream general purpose language, such as Java, to facilitate the maintenance of existing codebase and to extend the system by leveraging new features of the popular language. When building phone apps, Mobile developers may port a mobile application from one platform (e.g., Android) to another (e.g. iOS) by translating code from Java to Swift.

Most researchers and engineers built cross-language program transformation tools by hard coding the translation rules and implementing any missing equivalent functionality between languages [?,?,?,?,?]. For instance, SPiCE translates Smalltalk programs to C [?]. The two languages are different in two aspects. First, the execution model of Smalltalk, which creates activation records as objects, is very different from that of C. Second, Smalltalk and C have very different approaches to storage management. To overcome the challenges, Yasumatsu et al. created runtime replacement classes implementing the same functionality of Smalltalk classes that are inherently part of the Smalltalk execution model. They also provided semi-conservative real-time compacting garbage collection that works without language support. Mossienko [?] and Sneed [?] automated COBOL-to-Java code migration by defining and implementing rules to generate Java classes, methods, and packages from COBOL programs. Although some of these tools can perform very complicated code translations tasks, it always costs a lot of time and effort to manually build such tools from scratch between any two languages. HTML5 is a markup language used for structuring and presenting content on the World Wide Web [?]. To simplify cross-platform mobile software development, PhoneGap [?] was built to automatically translate HTML5 implementation to Android or iOS native code.

3.6 Porting programs to different platforms

4 An Organized Tour of Seminal Papers: II. Inspecting Program Changes

4.1 Code Reviews Practices

since code reviews is a common context where change inspection happens (peter rigby) code flow, collaborators in industry. generally starting from code review practices

Miryung to Tianyi: make the introduciton a little bit broad Peer code review is a quality assurance mechanism that requires software developers to manually inspect each other's source code to find programming mistakes overlooked in the initial development phase. In 1976, Fagan formalized code review as a highly structured inspection process with multiple stages (e.g., *overview*, *preparation*, *inspection*, *rework*, *follow-up*) and multiple participants (e.g., *moderator*, *designer*, *implementor*, and *tester*) [**?**]. This inspection process is performed at the end of each software development phase (e.g., design, implementing, testing), in which software developers attend a series of group meetings and review design documentations and source code line by line. Such careful and thorough inspection process has been proven effective in terms of finding bugs, but the cumbersome and time-consuming nature of this process hinders its universal adoption in practice.

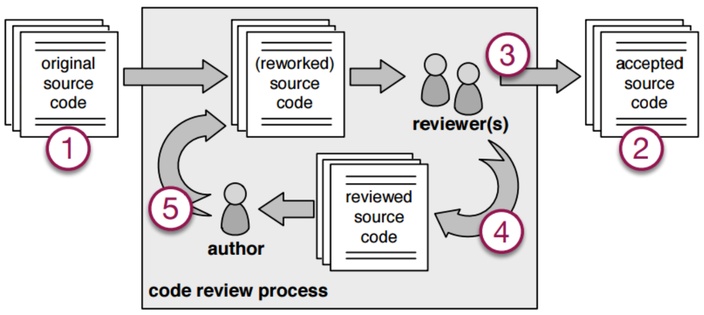


Fig. 6. The modern code review process (based on [**?**])

Nowadays many organizations adopt lightweight code review processes with less overhead than formal code inspections. In contrast to formal code review, modern code review occurs more regularly (often on a daily basis) and does not require a dedicated code review team with particular members. In particular, there is a clear trend towards utilizing code review tools to support code review tasks, instead of organizing and attending group meetings. Figure 6 shows the overall process of a code review task. The *author* first submits the *original source code* for review. The *reviewers* then decide whether the submitted code meets the project's quality acceptance criteria. If not, reviewers can add review comments to the source code and send the *reviewed source code* back to the author. The author then revises the source code to address reviewers' comments and then send it back for further review. This process continues till all reviewers accept the revised source code. Modern code review is widely practiced both in open-source and industrial contexts and is recognized as a valuable tool to remove common vulnerabilities, performance bugs, and style issues.

Tianyi: Could you please expand upon the following a little more. I would like this to be about 2 pages in total with some references to recent work such as a short 4 page paper by MSR on code reviews in ICSE 2016. Recent studies have investigated the common practices and challenges in modern code review. Rigby et al. conduct a case study in an open-source software project, Apache HTTP server, using archived code review records in email discussions and version control repositories [?]. They find that, in contrast to traditional software inspection (Fagan style), modern code review in open-source projects often involves frequent review tasks of small yet complete program contributions, conducted asynchronously by a small group of self-selected experts. Rigby and Bird conduct a similar investigation on a more diverse set of software projects, including two two Google-led projects, Android and Chromium OS, three Microsoft projects, Bing, Office, and MS SQL, and internal projects at Advanced Micro Devices (AMD) [?]. They find several convergent code review practices that may indicate general principles of modern code review. For example, modern code review often involves two reviewers and is performed regularly and quickly. They also observe the benefits of sharing knowledge across the development team via peer code review. In order to understand the expectations, outcomes, and challenges of modern code review, Bacchelli and Bird observe, interview, and survey software developers at Microsoft and manually classify hundreds of review comments across several Microsoft teams [?]. They find that, although the top motivation of code review is finding defects, the actual outcomes are less about finding defects than expected: only a small portion of review comments are related to software defects, which mainly cover small, low-level logical issues. A key challenge reported by reviewers is the lack of tool support for code change comprehension during peer code review. Tao et al. also study the challenges that developers face when they comprehend code changes and find that modern code review tools must support the capability to divide a large chunk of code changes into small, cohesive groups and to filter non-essential changes [?].

The following text should be placed under code review practices.

Prior work also observes that developers often submit program changes from multiple programming tasks (e.g., bug fixing, refactorings, feature additions) to a single code review. Reviewers sometimes use “chunky changes” or “code bombs” to describe such large, unrelated changes that are bundled in a single review. Such changes often lead to difficulty in change comprehension, since reviewers have to mentally “untangle” them to figure out which subset of changes addresses which issue first [?,?,?]. Reviewers have indicated that they can better understand small, cohesive changes rather than large, tangled ones [?]. For example, a code reviewer commented on Gson revision 1154 saying “I would have preferred to have two different commits: one for adding the new `getFieldNamingPolicy` method, and another for allowing overriding of primitives.”[1] This motivates the need of decoupling composite changes in a large code review.

[1] `https://code.google.com/p/google-gson/source/detail?r=1154`

4.2 Techniques for Code Change Comprehension

Tianyi: Make one of the commercial code review tools a little bit more concrete and go deeper with screen shots. the following text seems like it was just copied from the intro of critics paper.

Code review is effective only when reviewers are able to understand the changes being made. When the information required to inspect code changes is distributed across multiple files, developers find it difficult to inspect code changes [?]. For example, when an API gets modified in the latest release, all call sites using this API must be updated correctly. Such edits tend to be systematic—involving similar but not identical edits to multiple locations.

For example, take a snapshot of CodeFlow or Code Collaborator, describe the tool's feature in more detail. target length should be 1 page including a screenshot.

Unfortunately, popular code review tools—Facebook's PHABRICATOR,[2] Google's GERRIT,[3] and Microsoft's CODEFLOW,[4]—all compute program differences per file. This obliges reviewers to read changed lines file by file, even when those cross-file changes are done systematically to address the same issue. Therefore, reviewers are left to manually inspect individual edits to answer questions such as "what other code locations are changed similar to this change?" and "are there any other locations that are similar to this code but are not updated?"

Interactive Code Review using Code Patterns Right now there's too much focus on Critics. Reduce the length to be 1.5 pages including a screen snapshot.

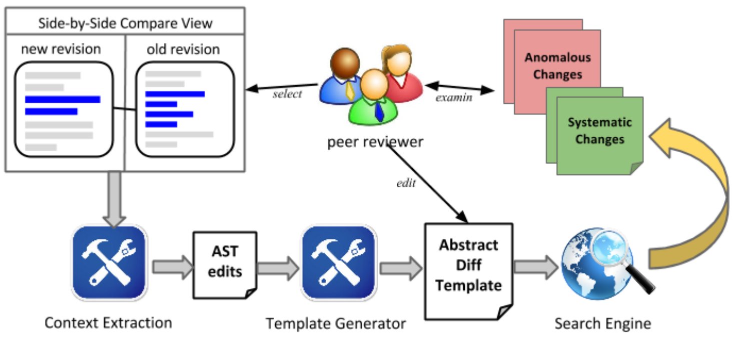
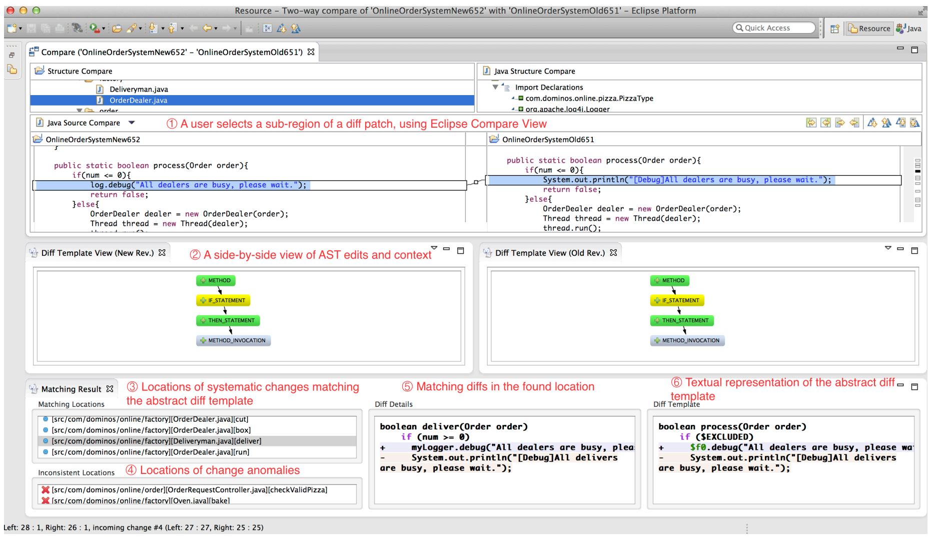


Fig. 7. The workflow of CRITICS

[2] http://phabricator.org
[3] http://code.google.com/p/gerrit/
[4] http://visualstudioextensions.vlasovstudio.com/2012/01/06/codeflow-code-review-tool-for-visual-studio/

To address this issue, Zhang et al. present CRITICS, a novel approach that allows reviewers to interactively inspect such systematic changes during peer code review [?]. Figure 7 describes the interactive workflow of CRITICS. Given a specified change that a reviewer would like to inspect, CRITICS creates a change template from the selected change, which serves as the pattern for searching similar changes. CRITICS includes *change context* in the template—unchanged, surrounding program statements that are relevant to the selected change. CRITICS models the template as Abstract Syntax Tree (AST) edits and allows reviewers to iteratively customize the template by parameterizing its content and by excluding certain statements. CRITICS then matches the customized template against the rest of the codebase to summarize similar changes and locate potential inconsistent or missing changes. Reviewers can incrementally refine the template and progressively search for similar changes until they are satisfied with the inspection results. This interactive feature allows reviewers with little knowledge of a codebase to flexibly explore the program changes with a desired pattern.


Resource – Two-way compare of 'OnlineOrderSystemNew652' with 'OnlineOrderSystemOld651' – Eclipse Platform
Quick Access
Resource
Java
Compare ('OnlineOrderSystemNew652' – 'OnlineOrderSystemOld651')
Structure Compare
Deliveryman.java
OrderDealer.java
order
Java Structure Compare
Import Declarations
com.dominos.online.pizza.PizzaType
org.apache.log4j.Logger
Java Source Compare
① A user selects a sub-region of a diff patch, using Eclipse Compare View
OnlineOrderSystemNew652
OnlineOrderSystemOld651
Diff Template View (New Rev.)
② A side-by-side view of AST edits and context
Diff Template View (Old Rev.)
METHOD
IF_STATEMENT
THEN_STATEMENT
METHOD_INVOCATION
Matching Result
③ Locations of systematic changes matching the abstract diff template
⑤ Matching diffs in the found location
⑥ Textual representation of the abstract diff template
Matching Locations
Inconsistent Locations
④ Locations of change anomalies
Diff Details
Diff Template

```
    public static boolean process(Order order){
        if(num <= 0){
            log.debug("All dealers are busy, please wait.");
            return false;
        }else{
            OrderDealer dealer = new OrderDealer(order);
            Thread thread = new Thread(dealer);
```

```
    public static boolean process(Order order){
        if(num <= 0){
            System.out.println("[Debug]All dealers are busy, please wait.");
            return false;
        }else{
            OrderDealer dealer = new OrderDealer(order);
            Thread thread = new Thread(dealer);
            thread.run();
```

Matching Locations

- [src/com/dominos/online/factory][OrderDealer.java][cut]
- [src/com/dominos/online/factory][OrderDealer.java][box]
- [src/com/dominos/online/factory][Deliveryman.java][deliver]
- [src/com/dominos/online/factory][OrderDealer.java][run]

Inconsistent Locations

- [src/com/dominos/online/order][OrderRequestController.java][checkValidPizza]
- [src/com/dominos/online/factory][Oven.java][bake]

```
boolean deliver(Order order)
    if (num >= 0)
+       myLogger.debug("All dealers are busy, plea
-       System.out.println("[Debug]All delivers
are busy, please wait.");
```

```
boolean process(Order order)
    if ($EXCLUDED)
+       $f0.debug("All dealers are busy, please wa
-       System.out.println("[Debug]All delivers
are busy, please wait.");
```

Left: 28 : 1, Right: 26 : 1, incoming change #4 (Left: 27 : 27, Right: 25 : 25)

Fig. 8. A screen snapshot of CRITICS's Eclipse plugin and its features

CRITICS is implemented as an Eclipse plugin.[5] Figure 8 shows a screenshot of CRITICS plugin. CRITICS is integrated with the **Compare View** in Eclipse, which displays line-level differences per file (see ① in Figure 8). A user can specify a program change she wants to inspect by selecting the corresponding code region in the Eclipse Compare View. The **Diff Template View** (see ②

[5] CRITICS's tool and evaluation dataset are available online `https://sites.google.com/a/utexas.edu/critics/`

in Figure 8) visualizes the change template of the selected change in a side-by-side view. Reviewers can parameterize concrete identifiers and exclude certain program statements by clicking on the corresponding node in the Diff Template View. **Textual Diff Template View** (see ⑥ in Figure 8) shows the change template in a unified format. The **Matching Result View** summarizes the consistent changes as *similar changes* (see ③ in Figure 8) and inconsistent ones as *anomalies* (see ④ in Figure 8).

the evaluation and quotes are too much for this chapter. so I removed them

Change Decomposition To address this issue, Barnett et al. present ClusterChanges, a lightweight static analysis technique for decomposing large changes. The insight is that program changes that address the same issue can be related via implicit dependency information such as *def-use* relationship. For example, if a method definition is changed in one location and its callsites are changed in two other locations, these three changes are likely to be related and should be reviewed together. Given a code review task, ClusterChanges first collects the set of definitions for types, fields, methods, and local variables in the corresponding project under review. Then ClusterChanges scans the project for all uses (i.e., references to a definition) of the defined code elements. For instance, any occurrence of a type, field, or method either inside a method or a field initialization is considered to be a use. Based on the extracted def-use information, ClusterChanges identifies three relationships between program changes.

Remove a bullet list and integrate into text.

- **Def-use relation**. If the definition of a method or a class field is changed, all the uses should also be updated. The change in the definition and the corresponding changes in its references are considered related.
- **Use-use relation**. If two or more uses of a method or a class field defined within the change-set are changed, these changes are considered related.
- **Enclosing relation**. Program changes in the same method are considered related because a) based on observation, program changes to the same method are often related, (b) reviewers often inspect methods atomically rather than reviewing different changed regions in the same method separately.

Given these relations, ClusterChanges creates a partition over the set of program changes by computing a transitive closure of related changes. On the other hand, if a change is not related to any other changes, it will be put into a specific partition, *miscellaneous changes*.

Expand the following technique a little bit more. Give similar weights to ClusterChange. Independently from ClusterChanges, Tao et al. present a similar change decomposition technique that leverages more sophisticated heuristics, other than the def-use analysis only [?]. Tao et al. cluster changes based on the following heuristics.

4.3 Program Differencing

program differencing (going to back history of differencing, diff, AST diff, CFG diff, PDG diff.— mention its symmetric problem of clone detection) go deeper for a few techniques that help with change comprehension: kim et al. critics, chris bird at MS

5 An Organized Tour of Seminal Papers: III. Debugging and Testing Program Changes

5.1 Delta Debugging: Finding Code Changes Causing Errors

2 pages

5.2 Regression Testing

2 pages

Change Impact Analysis 2 pages

6 Future Directions and Open Problems

4 pages

Acknowledgments. The heading should be treated as a subsubsection heading and should not be assigned a number.

7 The References Section

Appendix