Software Evolution

Miryung Kim, Na Meng, Tianyi Zhang

Abstract. Software evolution plays an ever-increasing role in software development. Programmers rarely build software from scratch but often spend more time in modifying existing software to provide new features to customers and fix defects in existing software. Evolving software systems is often a time-consuming and error-prone process. This chapter overviews key concepts and principles in the area of software evolution and presents the fundamentals of state-of-the art methods, tools, and techniques for evolving software. The chapter first classifies the types of software changes into four types: *perfective* changes to expand the existing requirements of a system, *corrective* changes for resolving defects, *adaptive* changes to accommodate any modifications to the environments, and finally *preventive* changes to improve the maintainability of software. For each type of changes, the chapter overviews software evolution techniques from the perspective of three kinds of activities: (1) applying changes, (2) inspecting changes, and (3) validating changes. The chapter concludes with the discussion of open problems and research challenges for the future.

1 Introduction

Software evolution plays an ever-increasing role in software development. Programmers rarely build software from scratch but often spend more time in modifying existing software to provide new features to customers and fix defects in existing software. Evolving software systems is often a time-consuming and error-prone process. In fact, it is reported that 90% of the cost of a typical software system is incurred during the maintenance phase [119] and a primary focus in software engineering involves issues relating to upgrading, migrating, and evolving existing software systems.

The term, *software evolution* dates back to 1976 when Belady and Lehman first coined this term. Software evolution refers to the *dynamic behavior* of software systems, as they are maintained and enhanced over their lifetimes [23]. Software evolution is particularly important as systems in organizations become longer-lived. A key notion behind this seminal work by Belady and Lehman is the concept of software system *entropy*. The term entropy, with a formal definition in physics relating to the amount of energy in a closed thermodynamic system is used to broadly represent a measure of the cost required to change a system or correct its natural disorder. As such, this term has had significant appeal to software engineering researchers, since it suggests a set of reasons for software maintenance. Their original work in the 1970s involved studying 20 user-oriented releases of the IBM OS/360 operating systems software, and it

was the first empirical research to focus on the dynamic behavior of a relatively large and mature system (12 years old) at the time. Starting with the available data, they attempted to deduce the nature of consecutive releases of OS/360 and to postulate five *laws* of software evolution: (1) continuing change, (2) increasing complexity, (3) fundamental law of program evolution, (4) conservation of organizational stability, and (5) conservation of familiarity.

Later, many researchers have systematically studied software evolution by measuring concrete metrics about software over time. Notably, Eick et al. [49] quantified the symptoms of *code decay—software is harder to change than it should be* by measuring the extent to which each risk factor matters using a rich data set of 5ESS telephone switching system. For example, they measured the number of files changed in each modification request to monitor code decay progress over time. This empirical study has influenced a variety of research projects on mining software repositories.

Now that we accept the fact that software systems go through a *continuing life cycle of evolution* after the initial phase of requirement engineering, design, analysis, testing and validation, we describe an important aspect of software evolution—*software changes* in this chapter. To that end, we first introduce the categorization of software changes into four types in Section 2. We then discuss the techniques of evolving software from the perspectives of three kinds of activities: (1) change application, (2) change inspection, and (3) change validation. In the following three sections, we provide an organized tour of seminal papers focusing on the above-mentioned topics.

In Section 3, we first discuss empirical studies to summarize the characteristics of each change type and then overview tool support for applying software changes. For example, for the type of *corrective changes*, we present several studies on the nature and extent of bug fixes. We then discuss automated techniques for fixing bugs such as automated repair. Similarly, for the type of *preventative changes*, we present empirical studies on refactoring practices and then discuss automated techniques for applying refactorings. Regardless of change types, various approaches could reduce the manual effort of updating software through automation, including source-to-source program transformation, Programming by Demonstration (PbD), simultaneous editing, and systematic editing.

In Section 4, we overview research topics for inspecting software changes. Software engineers other than the change author often perform peer reviews by inspecting program changes, and provide feedback if they discover any suspicious software modifications. Therefore, we summarize modern code review processes and discuss techniques for comprehending code changes. This section also overviews a variety of program differencing techniques, refactoring reconstruction techniques, and code change search techniques that developers can use to investigate code changes.

In Section 5, we overview research techniques for validating software changes. After software modification is made, developers and testers may create new tests or reuse existing tests, run the modified software against the tests, and check whether the software executes as expected. Therefore, the activity of checking

the correctness of software changes involves failure-inducing change isolation, regression testing, and change impact analysis.

2 Concepts and Principles

Swanson initially identified three categories of software changes: corrective, adaptive, and perfective [179]. These categories were updated later and ISO/IEC 14764 instead presents four types of changes: corrective, adaptive, perfective, and preventive [11].

2.1 Corrective Change

Corrective change refers software modifications initiated by software defects. A defect can result from design errors, logic errors, and coding errors [9].

- Design errors: software design does not fully align with the requirements specification. The faulty design leads to a software system that either incompletely or incorrectly implements the requested computational functionality.
- Logic errors: a program behaves abnormally by terminating unexpectedly or producing wrong outputs. The abnormal behaviors are mainly due to flaws in software functionality implementations.
- Coding errors: although a program can function well, it takes excessively high runtime or memory overhead before responding to user requests. Such failures may be caused by loose coding, or the absence of *reasonable checks* on computations performed.

2.2 Adaptive Change

Adaptive change is a change introduced to accommodate any modifications in the environment of a software product. The term **environment** here refers to the totality of all conditions that influence the software product, including business rules, government policies, and software and hardware operating systems. For example, when a library or platform developer may evolve its APIs, the corresponding adaptation may be required for client applications to handle such environment change. As another example, when porting a mobile application from Android to iOS, mobile developers need to apply adaptive changes to translate the code from Java to Swift, so that the software is still compilable and executable on the new platform.

2.3 Perfective Change

Perfective change is the change undertaken to expand the existing requirements of a system [168]. When a software product becomes useful, users always expect to use it in new scenarios beyond the scope for which it was initially developed. Such requirement expansion causes changes to either enhance existing

system functionality or to add new features. For instance, an image processing system is originally developed to process JPEG files, and later goes through a series of perfective changes to handle other formats, such as PNG and SVG. The nature and characteristics of new feature additions is not necessarily easy to define and in fact understudied for that reason. In Section 3.3, we discuss a rather well-understood type of perfective changes, called *crosscutting concerns* and then present tool and language support for adding crosscutting concerns. Crosscutting concerns refer to the *secondary design decisions* such as logging, performance, error handling, and synchronization. Adding these secondary concerns often involves non-localized changes throughout the system, due to the *tyranny* of dominant design decisions already implemented in the system. Concerns that are added later may end up being scattered across many modules and thus tangled with one another.

2.4 Preventive Change

Preventive change is the change applied to prevent malfunctions or to improve the maintainability of software. According to Lehman's laws of software evolution [113], the long-term effect of corrective, adaptive, and perfective changes is deteriorating the software structure, while increasing entropy. Preventive changes are usually applied to address the problems. For instance, after developers fix some bugs and implement new features in an existing software product, the complexity of source code can increase to an unmanageable level. Through code *refactoring*—a series of behavior-preserving changes, developers can reduce code complexity, and increase the readability, reusability, and maintainability of software.

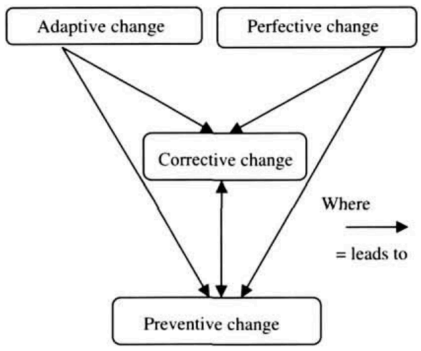


Fig. 1. Potential relation between software changes [168]

Figure 1 presents the potential relationships between different types of changes [168]. Specifically, both adaptive changes and perfective changes may lead to the other two types of changes, because developers may introduce bugs or worsen code

structures when adapting software to new environments or implementing new features.

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

We discuss the characteristics of *corrective*, *adaptive*, *perfective*, and *preventative* changes using empirical studies and the process and techniques for updating software, respectively in Sections 3.1, 3.2, 3.3, and 3.4. Next, regardless of change types, automation could reduce the manual effort of updating software. Therefore, we discuss the topic of automated program transformation and interactive editing techniques for reducing repetitive edits in Section 3.5.

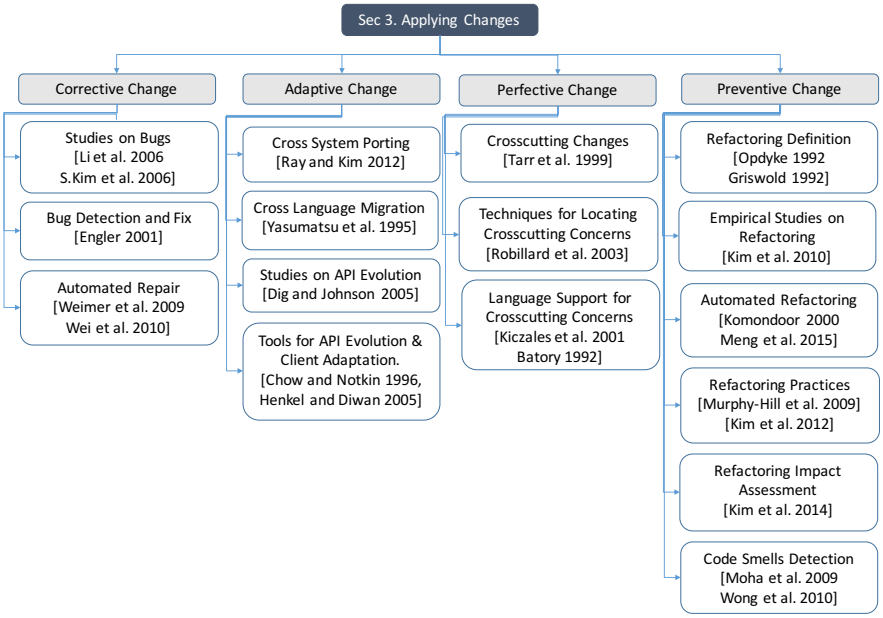


Fig. 2. Applying Changes Categorized by Change Type and Related Research Topics

3.1 Corrective Change

Corrective changes such as bug fixes are frequently applied by developers to eliminate defects in software. There are mainly two lines of research conducted: (1) empirical studies to characterize bugs and corresponding fixes, and (2) automatic approaches to detect and fix such bugs. There is no clear boundary between the two lines of research, because some prior projects first make observations about

particular kinds of bug fixes empirically and then subsequently leverage their observed characteristics to find more bugs and fix them. Below, we discuss a few representative examples of empirical studies with such flavor of characterizing and fixing bugs.

3.1.1 Empirical Studies of Bug Fixes. In this section, we discuss two representative studies on bug fixes. These studies are not the earliest, seminal works in this domain. Rather, the flavor and style of their studies are representative. Li et al. conducted is a large scale characterization of bugs by digging through bug reports in the wild and by quantifying the extent of each bug type [116]. S. Kim et al.'s *memory of bug fixes* [101] uses fine-grained bug fix histories to measure the extent of recurring, similar bug fixes and to assess the potential benefit of automating similar fixes based on change history.

Li et al. conducted an empirical study of bugs from two popular open source projects: Mozilla and Apache HTTP Server [116]. By manually examining 264 bug reports from the Mozilla Bugzilla database, and 209 bug reports from the Apache Bugzilla database, they investigated the root cause, impact, and software components of each software error that exhibited abnormal runtime behaviors. They observed three major root causes: *memory*, *concurrency*, and *semantics*. The memory bugs accounted for 16.3% in Mozilla and 12.2% in Apache. Among memory bugs, NULL pointer dereference was observed as a major cause, accounting for 37.2% in Mozilla and 41.7% in Apache. More importantly, semantic bugs were observed to be dominant, accounting for 81.1% in Mozilla and 86.7% in Apache. One possible reason is that most semantic bugs are specific to applications. A developer could easily introduce semantic bugs while coding, due to a lack of thorough understanding of software and its requirements. It is challenging to automatically detect or fix such semantic bugs, because diagnosing and resolving them may require a lot of domain-specific knowledge and such knowledge is inherently not generalizable across different systems and applications.

To understand the characteristics and frequency of project-specific bug fixes, Kim et al. conducted an empirical study on the bug fix history of five open source projects: ArgoUML, Columba, Eclipse, jEdit, and Scarab [101]. With keywords like "Fixed" or "Bugs", they retrieved code commits in software version history that are relevant to bug fixes, chopped each commit into contiguous code change blocks (i.e., hunks), and then clustered similar code changes. They observed that 19.3 to 40.3% bugs appeared repeatedly in version history, while 7.9 to 15.5% of bug-and-fix pairs appeared more than once. The results demonstrated that project-specific bug fix patterns occur frequently enough and for each bug-and-fix pair, it is possible to both detect similar bugs and provide fix suggestions. Their study also showed history-based bug detection could be complementary to static analysis-based bug detection—the bugs that can be detected by past bug fix histories do not overlap with the bugs that can be detected by a static bug finding tool, PMD [6].

3.1.2 Rule-based Bug Detection and Fixing Approaches. Rule-based bug detection approaches detect and fix bugs based on the assumption that bugs are *deviant program behaviors* that violate implicit programming rules. Then one may ask, where those implicit rules are coming from? Such rules can be written by the developers of bug-finding tools or can be refined based on empirical observation in the wild. For example, Engler et al. define a meta-language for users to easily specify temporal system rules such as "release locks after acquiring them" [52]. They also extend a compiler to interpret the rules and dynamically generate additional checks in the compiler. If any code snippet violates the specified rule(s), the approach reports the snippet as a software bug. Table 1 presents some exemplar system rule templates and instances. With this approach, developers can flexibly define their own rules to avoid some project-specific bugs, without worrying about how to implement checkers to enforce the rules. Engler et al.'s later work enables tool developers to tailor rule templates to a specific system and to check for contradictions and violations [53].

Table 1. Sample system rule templates and examples from [52]

Rule template	Example
"Never/always do X"	"Do not use floating point in the kernel"
"Do X rather than Y"	"Use memory mapped I/O rather than copying"
"Always do X before/after Y"	"Check user pointers before using them in the kernel"

As another example of rule-based bug detection is CP-Miner, an automatic approach to find copy-paste bugs in large-scale software [115]. CP-Miner is motivated by Chou et al.'s finding that, under the Linux drivers/i2o directory, 34 out of 35 errors were caused by copy-paste [33] and based on the insight that when developers copy and paste, they may forget to consistently rename identifiers. CP-Miner first identifies copy-paste code in a scalable way, and then detects bugs by checking for a specific rule, e.g., consistent renaming of identifiers.

3.1.3 Automated Repair. Automatic program repair generates candidate patches and checks correctness using compilation, testing, and/or specification.

One set of techniques uses *search-based repair* [66] or predefined repair templates to generate many candidate repairs for a bug, and then validates them using indicative workloads or test suites. For example, GenProg generates candidate patches by replicating, mutating, or deleting code *randomly* from the existing program [112, 197]. GenProg uses genetic programming (GP) to search for a program variant that retains required functionality but is not vulnerable to the defect in question. GP is a stochastic search method inspired by biological evolution that discovers computer programs tailored to a particular task. GP uses computational analogs of biological mutation and crossover to generate new program variations, in other words, program variants. A user-defined fitness function evaluates each variant. GenProg uses the input test cases to evaluate

the fitness, and individuals with high fitness are selected for continued evolution. This GP process is successful, when it produces a variant that passes all tests encoding the required behavior and does not fail those encoding the bug.

Another class of strategies in automatic software repair relies on *specifications* or *contracts* to guide sound patch generation. This provides confidence that the output is correct. For example, AutoFix-E generates simple bug fixes from manually prescribed contracts [196]. The key insights behind this approach are to rely on contracts present in the software to ensure that the proposed fixes are semantically sound. AutoFix-E takes an Eiffel class and generates test cases with some automated testing engine first. From the test runs, it extracts object states using boolean queries. By comparing the states of passing and failing runs, it then generates a fault profile—an indication of what went wrong in terms of an abstract object state. From the state transitions in passing runs, it generates a finite-state behavioral model, capturing the normal behavior in terms of control. Both control and state guide the generation of fix candidates, and only those fixes passing the regression test suite remain.

Some approaches are specialized for particular types of bugs only. For example, FixMeUp inserts missing security checks using inter-procedural analysis, but these additions are very specific and stylized for access-control related security bugs [176]. As another example, PAR [92] encodes ten common bug fix patterns from Eclipse JDT's version history to improve GenProg. However, the patterns are created manually.

3.2 Adaptive Change

Adaptive changes are applied to software, when its environment changes. In this section, we focus on three scenarios of adaptive changes: cross-system software porting, cross-language software migration, and software library upgrade (i.e., API evolution).

Consider an example of cross-system porting. When a software system is installed on a computer, the installation can depend on the configurations of the hardware, the software, and the device drivers for particular devices. To make the software to run on a different processor or an operating system, and to make it compatible with different drivers, we may need adaptive changes to adjust the software to the new environment. Consider another example of cross-language migration where you have software in Java that must be translated to C. Developers need to rewrite software and must also update language-specific libraries. Finally consider the example of API evolution. When the APIs of a library and a platform evolves, corresponding adaptations are often required for client applications to handle such API update. In extreme cases, e.g., when porting a Java desktop application to the iOS platform, developers need to rewrite everything from scratch, because both the programming language (i.e., Swift) and software libraries are different.

3.2.1 Cross-System Porting. Software forking—creating a variant product by copying and modifying an existing product—is often considered an ad hoc,

low cost alternative to principled product line development. To maintain such forked products, developers often need to port an existing feature or bug-fix from one product variant to another.

Empirical Studies on Cross-System Porting. OpenBSD, NetBSD, and FreeBSD have evolved from the same origin but have been maintained independently from one another. Many have studied the BSD family to investigate the extent and nature of cross-system porting. The studies found that (1) the information flow among the forked BSD family is decreasing according to change commit messages [55]; (2) 40% of lines of code were shared among the BSD family [206]; (3) in some modules such as device driver modules, there is a significant amount of adopted code [36]; and (4) contributors who port changes from other projects are highly active contributors according to textual analysis of change commit logs and mailing list communication logs [30].

More recently, Ray et al. comprehensively characterized the temporal, spatial, and developer dimensions of cross-system porting in the BSD family [156]. Their work computed the amount of edits that are ported from other projects as opposed to the amount of code duplication across projects, because not all code clones across different projects undergo similar changes during evolution, and similar changes are not confined to code clones. To identify ported edits, they first built a tool named as Repertoire that takes *diff* patches as input and compares the content and edit operations of the program patches. Repertoire was applied to total 18 years of NetBSD, OpenBSD and FreeBSD version history. Their study found that maintaining forked projects involves significant effort of porting patches from other projects—10% to 15% of patch content was ported from another project's patches. Cross-system porting is periodic and its rate does not necessarily decrease over time. A significant portion of active developers participate in porting changes from peer projects. Ported changes are less defect-prone than non-ported changes. A significant portion (26% to 59%) of active committers port changes but some do more porting work than others. While most ported changes migrate to peer projects in a relatively short amount of time, some changes take a very long time to propagate to other projects. Ported changes are localized within less than 20% of the modified files per release on average in all three BSD projects, indicating that porting is concentrated on a few sub systems.

3.2.2 Cross-Language Migration. When maintaining a legacy system that was written in an old programming language (e.g., 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 leverage new programming language features.

Cross-Language Program Translation. To translate code implementation from one language to another, researchers have built tools by hard coding the translation rules and implementing any missing functionality between languages. Yasumatsu et al. map compiled methods and contexts in Smalltalk to machine

code and stack frames respectively, and implement runtime replacement classes in correspondence with the Smalltalk execution model and runtime system [209]. Mossienko [132] and Sneed [174] automate COBOL-to-Java code migration by defining and implementing rules to generate Java classes, methods, and packages from COBOL programs. *mppSMT* automatically infers and applies Java-to-C# migration rules using a phrase-based statistical machine translation approach [139]. 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 sequence-to-sequence translation.

Mining Cross-Language API Rules. When migrating software to a different target language, API conversion poses a challenge for developers, because the diverse usage of API libraries induces an endless process of specifying API translation rules or identifying API mappings across different languages. Zhong et al. [216] and Nguyen et al. [138, 141] automatically mine API usage mappings between Java and C#. Zhong et al. align code based on similar names, and then construct the API transformation graphs for each pair of aligned statements [216]. StaMiner [138] mines API usage sequence mappings by conducting program dependency analysis [133] and representing API usage as a graph-based model [142].

3.2.3 Library Upgrade and API Evolution. Instead of building software from scratch, developers often use existing frameworks or third-party libraries to reuse well-implemented and tested functionality. Ideally, the APIs of libraries must remain stable such that library upgrades do not incur corresponding changes in client applications. In reality, however, APIs change their input and output signatures, change semantics, or are even deprecated, forcing client application developers to make corresponding adaptive changes in their applications.

Empirical Studies of API Evolution. Dig and Johnson manually investigated API changes using the change logs and release notes to study the types of library-side updates that break compatibility with existing client code, and discovered that 80% of such changes are refactorings [45]. Xing and Stroulia used UMLDiff to study API evolution and found that about 70% of structural changes are refactorings [204]. Yokomori et al. investigated the impact of library evolution on client code applications using component ranking measurements [211]. Padioleau et al. found that API changes in the Linux kernel led to subsequent changes on dependent drivers, and such collateral evolution could introduce bugs into previously mature code [149]. McDonelle et al. examined the relationship between API stability and the degree of adoption measured in propagation and lagging time in the Android Ecosystem [122]. Hou and Yao studied the Java API documentation and found that a stable architecture played an important role in supporting the smooth evolution of the AWT/Swing API [75]. In a large scale study of the Smalltalk development communities, Robbes et al. found that only

14% of deprecated methods produce non-trivial API change effects in at least one client-side project; however, these effects vary greatly in magnitude. On average, a single API deprecation resulted in 5 broken projects, while the largest caused 79 projects and 132 packages to break [162].

Tool Support for API Evolution and Client Adaptation. Several existing approaches semi-automate or automate client adaptations to cope with evolving libraries. Chow and Notkin [34] propose a method for changing client applications in response to library changes—a library maintainer annotates changed functions with rules that are used to generate tools that update client applications. Henkel and Diwan's CatchUp records and stores refactorings in an XML file that can be replayed to update client code [69]. However, its update support is limited to three refactorings: renaming operations (e.g. types, methods, fields), moving operations (e.g. classes to different packages, static members), or change operations (e.g. types, signatures). The key idea of CatchUp, *record-and-replay*, assumes that the adaptation changes in client code are exact or similar to the changes in the library side. Thus, it works well for replaying rename or move refactorings or supporting API usage adaptations via inheritance. However, CatchUp cannot suggest programmers how to manipulate the context of API usages in client code such as the surrounding control structure or the ordering between method-calls. Furthermore, CatchUp requires that library and client application developers use the same development environment to record API-level refactorings, limiting its adoption in practice. Xing and Stroulia's Diff-CatchU automatically recognizes API changes of the reused framework and suggests plausible replacements to the obsolete APIs based on the working examples of the framework codebase [205]. Dig et al.'s MolhadoRef uses recorded API-level refactorings to resolve merge conflicts that stem from refactorings; this technique can be used for adapting client applications in case of simple rename and move refactorings occurred in a library [46].

SemDiff [42] mines API usage changes from other client applications or the library itself. It defines an adaptation pattern as a frequent *replacement* of a method invocation. That is, if a method call to $A.m$ is changed to $B.n$ in several adaptations, $B.n$ is likely to be a correct replacement for the calls to $A.m$. As SemDiff models API usages in terms of method calls, it cannot support complex adaptations involving multiple objects and method calls that require the knowledge of the surrounding context of those calls. LibSync helps client applications migrate library API usages by learning migration patterns [140] with respect to a partial AST with containment and data dependences. Though it suggests what code locations to examine and shows example API updates, it is unable to transform code automatically. Cossette and Walker found that, while most broken code may be mended using one or more of these techniques, each is ineffective when used in isolation [38].

3.3 Perfective Change

Perfective change is the change undertaken to expand the existing requirements of a system. Not much research is done to characterize feature enhancement or addition. One possible reason is that the implementation logic is always domain and project-specific and that it is challenging for any automatic tool to predict what new feature to add and how that new feature must be implemented. Therefore, the nature and characteristics of feature additions are under-studied.

In this section, we discuss a rather well-understood type of perfective changes, called *crosscutting concerns* and techniques for implementing and managing crosscutting concerns. As programs evolve over time, they may suffer from the *the tyranny of dominant decomposition* [183]. They can be modularized in only one way at a time. Concerns that are added later may end up being scattered across many modules and tangled with one another. Logging, performance, error handling, and synchronization are canonical examples of such secondary design decisions that lead to non-localized changes.

Aspect-oriented programming languages provide language constructs to allow concerns to be updated in a modular fashion [91]. Other approaches instead leave the crosscutting concerns in a program, while providing mechanisms to document and manage related but dispersed code fragments. For example, Griswold's information transparency technique uses naming conventions, formatting styles, and ordering of code in a file to provide indications about crosscutting concern code that should change together [60].

3.3.1 Techniques for Locating Crosscutting Concerns. Several tools allow programmers to automatically or semi-automatically locate crosscutting concerns. Robillard et al. allow programmers to manually document crosscutting concerns using structural dependencies in code [163]. Similarly, the Concern Manipulation Environment allows programmers to locate and document different types of concerns [67]. Van Engelen et al. use clone detectors to locate crosscutting concerns [193]. Shepherd et al. locate concerns using natural language program analysis [170]. Breu et al. mine aspects from version history by grouping method-calls that are added together [28]. Dagenais et al. automatically infer and represent structural patterns among the participants of the same concern as rules in order to trace the concerns over program versions [41].

3.3.2 Language Support for Crosscutting Concerns. *Aspect-Oriented Programming* (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of crosscutting concerns [7]. Suppose developers want to add a new feature such as logging to log all executed functions. The logging logic is straightforward: printing the function's name at each function's entry. However, manually inserting the same implementation to each function body is tedious and error-prone. With AOP, developers only need to first define the logging logic as **an advice**, and then specify the place where to insert the advice (i.e., **pointcut**), such as the entry point of each function. An aspect weaver

will read the aspect-oriented code, and generate appropriate object-oriented code with the aspects integrated. In this way, AOP facilitates developers to efficiently introduce new program behaviors without cluttering the core implementation in the existing codebase. Many Java bytecode manipulation frameworks implement the AOP paradigm, like ASM [2], Javassist [5], and AspectJ [7], so that developers can easily modify program runtime behaviors without touching source code. The benefit of AOP during software evolution is that crosscutting concerns can be contained as a separate module such as an `aspect` with its `pointcut` and `advice` description, and thus reduces the developer effort in locating and updating all code fragments relevant to a particular secondary design decision such as logging, synchronization, database transaction, etc.

Feature Oriented Programming (FOP) is another paradigm for program generation in software product lines and for incremental development of programs [22]. FOP is closely related to AOP. Both deal with modules that encapsulate crosscuts of classes, and both express program extensions. In FOP, every software is considered as a composition of multiple features or layers. Each feature implements a certain program functionality, while features may interact with each other to collaboratively provide a larger functionality. A software product line (SPL) is a family of programs where each program is defined by a unique composition of features. Formally, FOP considers programs as *values* and program extensions as *functions* [108]. The benefit of FOP is similar to AOP in that secondary design decisions can be encapsulated as a separate feature and can be composed later with other features using program synthesis, making it easier to add a new feature at a later time during software evolution. Further discussion of program generation techniques for software product lines is described elsewhere in Chapter cross reference a chapter on the product line.

3.4 Preventive Change

As a software system is enhanced, modified, and adapted to new requirements, the code becomes more complex and drifts away from its original design, thereby lowering the quality of the software. *Refactoring* [10, 61, 127, 145] copes with increasing software complexity by transforming a program from one representation to another while preserving the program's external behavior (functionality and semantics). Mens et al. present a survey of refactoring research and describe a refactoring process, consisting of the following activities [127]:

1. Identifying where to apply what refactoring(s).
2. Checking that the refactoring to apply preserves program behaviors.
3. Refactoring the code.
4. Assessing the effect of applied refactoring on software quality (e.g., complexity and readability).
5. Maintaining the consistency between refactored code and other related software artifacts, like documentation, tests, and issue tracking records.

Section 3.4.1 describes the definition of refactoring and example transformations. Section 3.4.2 describes empirical studies on refactoring. Section 3.4.3

describes tool support for automated refactoring. Section 3.4.4 describes several studies of modern refactoring practices and the limitations of current refactoring support. Section 3.4.5 describes techniques for assessing the impact of refactoring. Section 3.4.6 describes techniques for identifying opportunities for refactoring.

3.4.1 Definition of Refactoring Operations. Griswold's dissertation [61] discusses one of the first refactoring operations that automate repetitive, error-prone, non-local transformations. Griswold supports a number of restructuring operations: replacing an expression with a variable that has its value, swapping the formal parameters in a procedure's interface and the respective arguments in its calls, etc. It is important to note that many of these refactoring operations are systematic in the sense that they involve repetitive non-local transformations.

Opdyke's dissertation [145] distinguishes the notion of low-level refactorings from high-level refactorings. High-level refactorings (i.e., composite refactorings) reflect more complex behavior-preserving transformations while low-level refactorings are primitive operations such as creating, deleting, or changing a program entity or moving a member variable. Opdyke describes three kinds of complex refactorings in detail: (1) creating an abstract superclass, (2) subclassing and simplifying conditionals, and (3) capturing aggregations and components. All three refactorings are systematic in the sense that they contain multiple similar transformations at a code level. For example, creating an abstract superclass involves moving multiple variables and functions common to more than one sibling classes to their common superclass. Subclassing and simplifying conditionals consists of creating several classes, each of which is in charge of evaluating a different conditional. Capturing aggregations and components usually involves moving multiple members from a component to an aggregate object.

While refactoring is defined as behavior-preserving code transformations in the academic literature [127], the de-facto definition of refactoring in practice seems to be very different from such rigorous definition. Fowler catalogs 72 types of structural changes in object oriented programs but these transformations do not necessarily guarantee behavior preservation [10]. In fact, Fowler recommends developers to write test code first, since these refactorings may change a program's behavior. Murphy-Hill et al. analyzed refactoring logs and found that developers often interleave refactorings with other behavior-modifying transformations [135], indicating that pure refactoring revisions are rare. Johnson's refactoring definition is aligned with these findings—*refactoring improves behavior in some aspects but does not necessarily preserve behavior in all aspects* [84].

3.4.2 Empirical Studies of Refactoring. There are contradicting beliefs on refactoring benefits. On one hand, some believe that refactoring improves software quality and maintainability and a lack of refactoring incurs *technical debt* to be repaid in the future in terms of increased maintenance cost [29]. On the other hand, some believe that refactoring do not provide immediate benefits unlike bug fixes and new features during software evolution.

Supporting the view that refactoring provides benefits during software evolution, researchers found empirical evidence that bug fix time decreases after refactoring and defect density decreases after refactoring. More specifically, Carriere et al. found that the productivity measure manifested by the average time taken to resolve tickets decreases after re-architecting the system [31]. Ratzinger et al. developed defect prediction models based on software evolution attributes and found that refactoring related features and defects have an inverse correlation [155]—if the number of refactorings increases in the preceding time period, the number of defects decreases.

Supporting the opposite view that refactoring may even incur additional bugs, researchers found that code churns are correlated with defect density and that refactorings are correlated with bugs. More specifically, Purushothaman and Perry found that nearly 10% of changes involved only a single line of code, which has less than a 4% chance to result in error, while a change of 500 lines or more has nearly a 50% chance of causing at least one defect [152]. This result may indicate that large commits, which tend to include refactorings, have a higher chance of inducing bugs. Weißgerber and Diehl found that refactorings often occur together with other types of changes and that refactorings are followed by an increasing number of bugs [198]. Kim et al. investigated the spatial and temporal relationship between API refactorings and bug fixes using a K-revision sliding window and by reasoning about the method-level location of refactorings and bug fixes. They found that the number of bug fixes increases after API refactorings [93].

One reason why refactoring could be potentially error-prone is that refactoring often requires coordinated edits across different parts of a system, which could be difficult for programmers to locate all relevant locations and apply coordinated edits consistently. Several researchers found such evidence from open source project histories—Kim et.al. found the exceptions to systematic change patterns, which often arise from the failure to complete coordinated refactorings [95, 96] cause bugs. Görg and Weißgerber detect errors caused by incomplete refactorings by relating API-level refactorings to the corresponding class hierarchy [59]. Nagappan and Ball found that code churn—the number of added, deleted, and modified lines of code—is correlated with defect density [136]—since refactoring often introduces a large amount of structural changes to the system, some question the benefit of refactoring.

3.4.3 Automated Refactoring. The Eclipse IDE provides automatic support for a variety of refactorings, including *rename*, *move*, and *extract* method. With such support, developers do not need to worry about how to check for preconditions or postconditions before manually applying a certain refactoring. Instead, they can simply select the refactoring command from a menu (e.g., *extract method*), and provide necessary information to accomplish the refactoring (e.g., *the name of a new method*). The Eclipse refactoring engine takes care of the precondition check, program transformation, and post-condition check.

During refactoring automation, Opdyke suggests to ensure behavior preservation by specifying *refactoring preconditions* [145]. For instance, when conducting a *create_method_function* refactoring, before inserting a member function F to a class C, developers should specify and check for five preconditions: (1) the function is not already defined locally. (2) The signature matches that of any inherited function with the same name. (3) The signature of corresponding functions in subclasses match it. (4) If there is an inherited function with the same name, either the inherited function is not referenced on instances of C and its subclasses, or the new function is semantically equivalent to the function it replaces. (5) F will compile as a member of C. If any precondition is not satisfied, the refactoring should not be applied to the program. These five conditions in Opdyke's dissertation is represented using first order logic.

Clone removal refactorings factorizes the common parts of similar code by parameterizing their differences using a *strategy* design pattern or a *form template method* refactoring [18, 74, 87, 106, 181]. These tools insert customized calls in each original location to use newly created methods. Juillerat et al. automate *introduce exit label* and *introduce return object* refactorings [87]. However, for variable and expression variations, they define extra methods to mask the differences [18]. Hotta et al. use program dependence analysis to handle gapped clones—trivial differences inside code clones that are safe to factor out such that they can apply the *form template method* refactoring to the code [74]. Krishnan et al. use PDGs of two programs to identify a maximum common subgraph so that the differences between the two programs are minimized and fewer parameters are introduced [106]. RASE is an advanced clone removal refactoring technique that (1) extracts common code; (2) creates new types and methods as needed; (3) parameterizes differences in types, methods, variables, and expressions; and (4) inserts return objects and exit labels based on control and data flow by combining multiple kinds of clone removal transformations [123]. Such clone removal refactoring could lead to an increase in the total size of code because it creates numerous simple methods.

Komondoor et al. extract methods based on the user-selected or tool-selected statements in one method [103, 104]. The *extract method* refactoring in the Eclipse IDE requires contiguous statements, whereas their approach handles non-contiguous statements. Program dependence analysis identifies the relation between selected and unselected statements and determines whether the non-contiguous code can be moved together to form extractable contiguous code. Komondoor et al. apply *introduce exit label* refactoring to handle exiting jumps in selected statements [104]. Tsantalis et al. extend the techniques by requiring developers to specify a variable of interest at a specific point only [190]. They use a block-based slicing technique to suggest a program slice to isolate the computation of the given variable. These automated procedure extraction approaches are focused on extracting code from a single method only. Therefore, they do not handle extracting common code from multiple methods and resolving the differences between them.

3.4.4 Real-World Refactoring Practices. Several studies investigated refactoring practices in industry and also examined the current challenges and risks associated with refactoring. Kim et al. conducted a survey with professional developers at Microsoft [98, 99]. They sent a survey invitation to 1290 engineers whose commit messages include a keyword "refactoring" in the version histories of five MS products. 328 of them responded to the survey. More than half of the participants said they carry out refactorings in the context of bug fixes or feature additions, and these changes are generally not semantics-preserving. When they asked about their own definition of refactoring, 46% of participants did not mention preservation of semantics, behavior, or functionality at all. 53% reported that refactorings that they perform do not match the types and capability of transformations supported by existing refactoring engines.

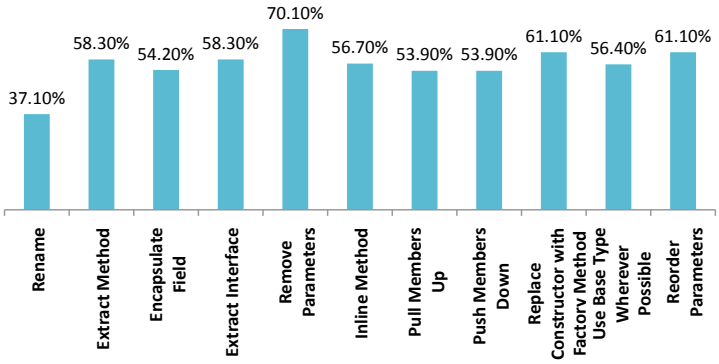


Fig. 3. The percentage of survey participants who know individual refactoring types but do those refactorings manually. [99]

In the same study, when developers are asked *"what percentage of your refactoring is done manually as opposed to using automated refactoring tools?"*, developers answered they do 86% of refactoring manually on average. Figure 3 shows the percentages of developers who usually apply individual refactoring types manually despite the awareness of automated refactoring tool support. Vakilian et al. [192] and Murphy et al. [134] also find that programmers do not use automated refactoring despite their awareness of the availability of automated refactorings. Murphy-Hill manually inspected source code produced by 12 developers and found that developers only used refactoring tools for 10% of refactorings for which tools were available [135]. For the question, *"based on your experience, what are the risks involved in refactorings?"*, developers reported regression bugs, code churn, merge conflicts, time taken from other tasks, the difficulty of doing code reviews after refactoring, and the risk of over-engineering. 77% think that refactoring comes with a risk of introducing subtle bugs and functionality regression [98].

In a separate study of refactoring tool use, Murphy-Hill et al. gave developers specific examples of when they did not use refactoring tools, but could have [135] and asked why. One reason was that developers started a refactoring manually, but only partway through realized that the change was a refactoring that the IDE offered—by then, it was too late. Another complaint was that refactoring tools disrupted their workflow, forcing them to use a tool when they wanted to focus on code.

3.4.5 Quantitative Assessment of Refactoring Impact. While several prior research efforts have conceptually advanced the benefit of refactoring through metaphors, few empirical studies assessed refactoring impact quantitatively. Sullivan et al. first linked software modularity with option theories [178]. A module provides an option to substitute it with a better one without symmetric obligations, and investing in refactoring activities can be seen as purchasing *options* for future adaptability, which will produce benefits when changes happen and the module can be replaced easily. Baldwin and Clark argued that the modularization of a system can generate tremendous value in an industry, given that this strategy creates valuable options for module improvement [20]. Ward Cunningham drew the comparison between debt and a lack of refactoring: a quick and dirty implementation leaves *technical debt* that incur *penalties* in terms of increased maintenance costs [40]. While these projects advanced conceptual understanding of refactoring impact, they did not quantify the benefits of refactoring.

Kim et al. studied how refactoring impacts inter-module dependencies and defects using the quantitative analysis of Windows 7 version history [99]. Their study finds the top 5% of preferentially refactored modules experience higher reduction in the number of inter-module dependencies and several complexity measures but increase size more than the bottom 95%. Based on the hypothesis that measuring the impact of refactoring requires multi-dimensional assessment, they investigated the impact of refactoring on various metrics: churn, complexity, organization and people, cohesiveness of ownership, test coverage and defects.

MacCormack et al. defined modularity metrics and used these metrics to study evolution of Mozilla and Linux. They found that the redesign of Mozilla resulted in an architecture that was significantly more modular than that of its predecessor. Their study monitored design structure changes in terms of modularity metrics without identifying the modules where refactoring changes are made [118]. Kataoka et al. proposed a refactoring evaluation method that compares software before and after refactoring in terms of coupling metrics [88]. Kolb et al. performed a case study on the design and implementation of existing software and found that refactoring improves software with respect to maintainability and reusability [102]. Moser et al. conducted a case study in an industrial, agile environment and found that refactoring enhances quality and reusability related metrics [131]. Tahvildari et al. suggested using a catalogue of object-oriented metrics to estimate refactoring impact, including complexity metrics, coupling metrics, and cohesion metrics [180].

3.4.6 Code Smells Detection. Fowler describes the concept of *bad smell* as a heuristic for identifying redesign and refactoring opportunities [10]. Example bad smells include code clone and feature envy. Several techniques automatically identify bad smells that indicate needs of refactorings [187–189].

Garcia et al. propose several architecture-level bad smells [57]. Moha et al. present the Decor tool and domain specific language (DSL) to automate the construction of design defect detection algorithms [130].

Tsantalis and Chatzigeorgiou's technique identifies *extract method* refactoring opportunities using static slicing [188]. Detection of some specific bad smells such as code duplication has also been extensively researched. Higo et al. propose the Aries tool to identify possible refactoring candidates based on the number of assigned variables, the number of referred variables, and dispersion in the class hierarchy [71]. A refactoring can be suggested if the metrics for the clones satisfy certain predefined values. Koni-N'Sapu provides refactoring suggestions based on the location of clones with respect to a class hierarchy [105]. Balazinska et al. suggest clone refactoring opportunities based on the differences between the cloned methods and the context of attributes, methods, and classes containing clones [19]. Kataoka et al. use Daikon to infer program invariants at runtime, and suggest candidate refactorings using inferred invariants [89]. If Daikon observes that one parameter of a method is always constant, it then suggests a *remove parameter* refactoring. *Breakaway* automatically identifies detailed structural correspondences between two abstract syntax trees to help programmers generalize two pieces of similar code [39].

Gueheneuc et al. detect inter-class design defects [63] and Marinescu identifies design flaws using software metrics [121]. Izurieta and Bieman detect accumulation of non design-pattern related code [77]. Guo et al. define domain-specific code smells [65] and investigate the consequence of technical debt [64]. Tsantalis et al. rank clones that have been repetitively or simultaneously changed in the past to suggest refactorings [191]. Wang et al. extract features from code to reflect program context, code smell, and evolution history, and then use a machine learning technique to rank clones for refactorings [195].

Among the above tools, we briefly present a few concrete examples of four design smells from Decor [130]. In `XERCES`, method `handleIncludeElement (XMLAttributes)` of the `org.apache.xerces.xinclude.XIncludeHandler` class is a typical example of *Spaghetti Code*—classes without structure that declare long methods without parameters. A good example of *Blob* (a large controller class that depends on data stored in surrounding data classes) is class `com.aelitis.azureus.core.dht.control.impl.DHTControlImpl` in AZUREUS. This class declares 54 fields and 80 methods for 2,965 lines of code. Functional decomposition may occur if developers with little knowledge of object-orientation implement an object-oriented system. An interesting example of *Functional Decomposition* is class `org.argouml.uml.cognitive.critics.Init` in ARGOUML, in particular because the name of the class includes a suspicious term, *init* that suggests a functional programming. The *Swiss Army Knife* code smell is a complex class that offers a high number of services, (i.e., interfaces). Class `org.apache.-`

`xerces.impl.dtd.DTDGrammar` is a striking example of Swiss Army Knife in XERCES, implementing four different sets of services with 71 fields and 93 methods for 1,146 lines of code.

Clio detects modularity violations based on the assumptions that multiple types of bad smells are instances of modularity violations that can be uniformly detected by reasoning about modularity hierarchy in conjunction with change locations [201]. They define *modularity violations* as recurring discrepancies between which modules should change together and which modules actually change together according to version histories. For example, when code clones change frequently together, Clio will detect this problem because the co-change pattern deviates from the designed modular structure. Second, by taking version histories as input, Clio detects violations that happened most recently and frequently, instead of bad smells detected in a single version without regard to the program's evolution context. Ratzinger et al. also detect bad smells by examining change couplings but their approach leaves it to developers to identify design violations from visualization of change coupling [154].

3.5 Automatic Change Application

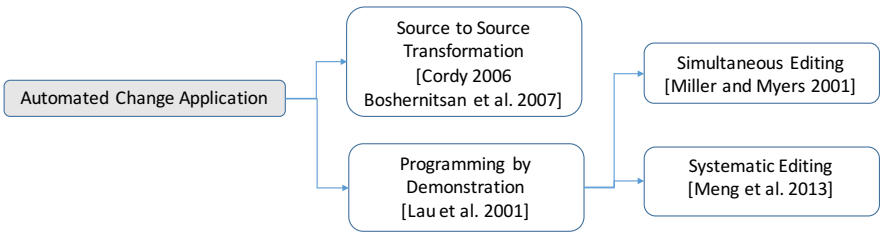


Fig. 4. Automated Change Application and Related Research Topics

Regardless of change types, various approaches are proposed to automatically suggest program changes or reduce the manual effort of updating software. In this section, we discuss automated change application techniques including source-to-source program transformation, Programming by Demonstration (PbD), simultaneous editing, and systematic editing.

3.5.1 Source Transformation and Languages and Tools. Source transformation tools allow programmers to author their change intent in a formal syntax and automatically update a program using the change script. Most source transformation tools automate repetitive and error-prone program updates. The most ubiquitous and the least sophisticated approach to program transformation is text substitution. More sophisticated systems use program structure information. For example, A* [107] and TAWK [62] expose syntax trees and primitive

data structures. Stratego/XT is based on algebraic data types and term pattern matching [194]. These tools are difficult to use as they require programmers to understand low-level program representations. TXL attempts to hide these low-level details by using an extended syntax of the underlying programming language [35]. Boshernitsan et al.'s iXJ enables programmers to perform systematic code transformations easily by providing a visual language and a tool for describing and prototyping source transformations. Their user study shows that iXj's visual language is aligned with programmers' mental model of code changing tasks [26]. Coccinelle [148] allows programmers to safely apply crosscutting updates to Linux device drivers. We describe two seminal approaches with more details.

Example: TXL TXL is a programming language and rapid prototyping system specifically designed to support structural source transformation. TXL's source transformation paradigm consists of parsing the input text into a structure tree, transforming the tree to create a new structure tree, and unparsing the new tree to a new output text. Source text structures to be transformed are described using an unrestricted ambiguous context free grammar in extended Backus-Nauer (BNF) form. Source transformations are described by example, using a set of context sensitive structural transformation rules from which an application strategy is automatically inferred.

Each transformation rule specifies a *target type* to be transformed, a *pattern* (an example of the particular instance of the type that we are interested in replacing), and a *replacement* (an example of the result we want when we find such an instance). In particular, the pattern is an actual source text example expressed in terms of tokens (terminal symbols) and variables (non-terminal types). When the pattern is matched, variable names are bound to the corresponding instances of their types in the match. Transformation rules can be composed like function compositions.

TXL programs normally consist of three parts, a context-free base grammar for the language to be manipulated, a set of context-free grammatical overrides (extensions or changes) to the base grammar, and a rooted set of source transformation rules to implement transformation of the extensions to the base language, as shown in Figure 5. This TXL program overrides the grammar of statements to allow a new statement form. The transformation rule `main` transforms the new form of a statement `V+=E` to an old statement `V:= V+(E)`. In other words, if there are two statements `foo+=bar` and `baz+=boo` they will be transformed to `foo:= foo+(bar)` and `baz:=baz+(boo)` at the source code level.

Example: iXj. iXj's pattern language consists of a *selection pattern* and a *transformation action*. iXj's transformation language allows grouping of code elements using a wild-card symbol `*`. Figure 6 shows an example selection pattern and a transformation pattern.

To reduce the burden of learning the iXj pattern language syntax, iXj's visual editor scaffolds this process through from-example construction and iterative refinement; When a programmer selects an example code fragment to change, iXj

```
% Trivial coalesced addition dialect of Pascal
% Based on standard Pascal grammar
include "Pascal.Grm"
% Overrides to allow new statement forms
redefine statement
       ...
    |  [reference] += [expression]
end redefine
% Transform new forms to old
rule main
    replace [statement]
      V [reference] += E [expression]
    by
      V := V + (E)
end rule
```

Fig. 5. A simple exemplar TXL file based on [8]

Selection pattern:
`* expression instance of java.util.Vector (:obj).removeElement(:method)(* expressions(:args))`
Match calls to the removeElement() method where the obj expression is a subtype of java.util.Vector.
Transformation action:
`$obj$.remove($obj$.indexOf($args$))`
Replace these calls with with calls to the remove() method whose argument is the index of an element to remove.

Fig. 6. *Example iXj transformation*

automatically generates an initial pattern from the code selection and visualizes all code fragments matched by the initial pattern. The initial pattern is presented in a pattern editor, and a programmer can modify it interactively and see the corresponding matches in the editor. A programmer may edit the transformation action and see the preview of program updates interactively.

3.5.2 Programming by Demonstration. Programming by Demonstration is also called Programming by Example (PbE). It is an end-user development technique for teaching a computer or a robot new behaviors by demonstrating the task to transfer directly instead of manually programming the task. Approaches were built to generate programs based on the text-editing actions demonstrated or text change examples provided by users [109,111,143,200]. For instance, TELS records editing actions such as search-and-replace, and generalizes them into a program that transforms input to output [200]. It leverages heuristics to match actions against each other to detect any loop in the user-demonstrated program.

SMARTedit is a representative early effort of applying PbD to text editing. It automates repetitive text-editing tasks by learning programs to perform them using techniques drawn from machine learning [111]. 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. Below, we describe two seminal PBD approaches applied to software engineering to automate repetitive program changes.

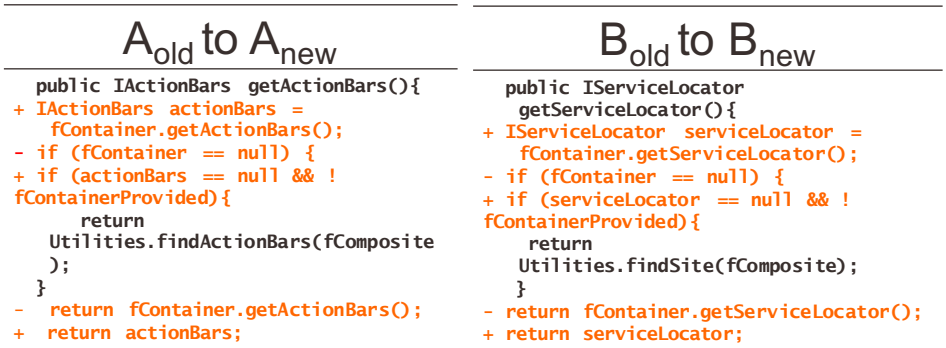


Fig. 7. An example of non-contiguous, abstract edits that can be applied using LASE [125]

Simultaneous Editing. Simultaneous editing repetitively applies source code changes that are interactively demonstrated by users [129]. When users apply their edits in one program context, the tool replicates the *exact lexical* edits to other code fragments, or transforms code accordingly. Linked Editing requires users to first specify the similar code snippets which they want to modify in the same way [185]. As users interactively edit one of these snippets, Linked Editing simultaneously applies the identical edits to other snippets.

Systematic Editing. Systematic editing is the process of applying similar, but not necessarily identical, program changes to multiple code locations. High-level changes are often systematic—consisting of related transformations at a code level. In particular, crosscutting concerns, refactoring, and API update mentioned in Sections 3.3, 3.2, and 3.4 are common kinds of systematic changes, because making these changes during software evolution involves tedious effort of locating individual change locations and applying similar but not identical changes. Several approaches have been proposed to infer the general program transformation from one or more code change examples provided by developers [124, 125, 164], and apply the transformation to other program contexts in need of similar changes. Specifically, LASE requires developers to provide multiple similarly changed code examples in Java (at least two) [125]. By extracting the commonality between demonstrated changes and abstracting the changes in terms of identifier usage and control- or data-dependency constraints in edit contexts, LASE creates a general program transformation, which can both detect code locations that should be changed similarly, and suggest customized code changes for each candidate location. For example, in Figure 7, LASE can take the change example on from A_{old} to A_{new} as input and apply to the code on B_{old} to generate B_{new}. Such change is similar but customized to the code on the right.

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

Section 4.1 presents the brief history of software inspection and discusses emerging themes from modern code review practices. Sections 4.1.1 to 4.1.5 discuss various methods that help developers better comprehend software changes, including *change decomposition*, *refactoring reconstruction*, *conflict* and *interference* detection, *related change search*, and *inconsistent change detection*. Section 4.2 describes various program differencing techniques that serve as a basis for analyzing software changes. Section 4.3 describes complementary techniques that record software changes during programming sessions.

4.1 Software Inspection and Modern Code Review Practices

To improve software quality during software evolution, developers often perform *code reviews* to manually examine software changes. Michael Fagan from IBM

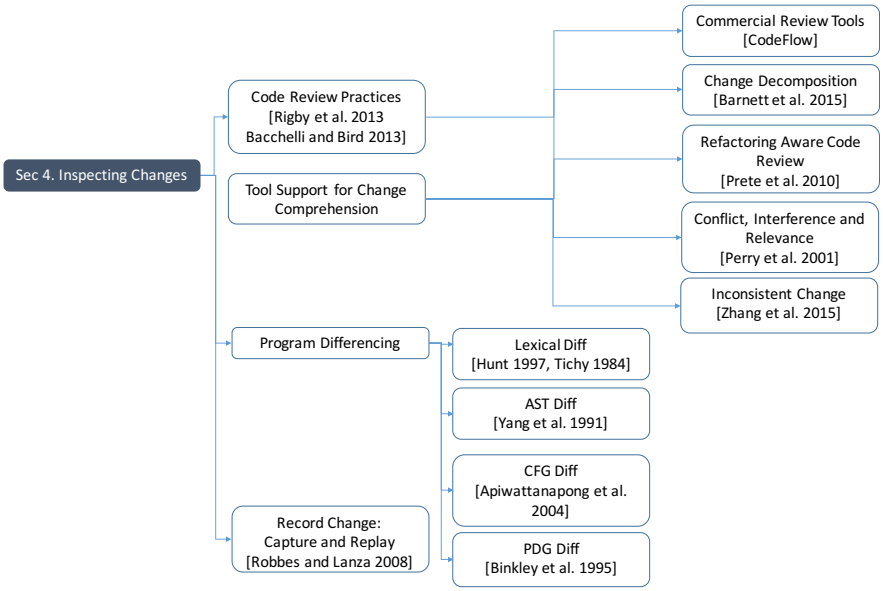


Fig. 8. Change Inspection and Related Research Topics

first introduced "code inspections", in a seminal paper in 1976 [54]. Code inspections are performed at the end of major software development phases, with the aim of finding overlooked defects before moving to the next phase. Software artifacts are circulated a few days in advance and then reviewed and discussed in a series of meetings. The review meetings include the author of an artifact, other developers to assess the artifact, and a meeting chair to moderate the discussion, and a secretary to record the discussion. Over the years, code inspections have been proved a valuable method to improve software quality. However, the cumbersome and time-consuming nature of this process hinders its universal adoption in practice [83].

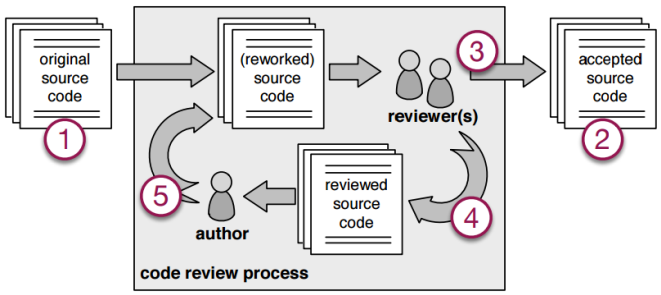


Fig. 9. Modern Code Review Process [24]

To avoid the inefficiencies in code inspections, most open-source and industrial projects adopt a lightweight, flexible code review process, which we refer to as *modern code reviews*. Figure 9 shows the workflow of modern code reviews. The *author* first submits the *original source code* for review. The *reviewers* then decide whether the submitted code meets the quality acceptance criteria. If not, reviewers can annotate the source code with review comments and send back the *reviewed source code*. The author then revises the code to address reviewers' comments and send it back for further reviews. This process continues till all reviewers accept the revised code.

In contrast to formal code inspections (Fagan style), modern code reviews occur more regularly and informally on program changes. Rigby et al. conducted the first case study about modern code review practices in an open-source software (OSS), Apache HTTP server, using archived code review records in email discussions and version control histories [160]. They described modern code reviews as "early, frequent reviews of small, independent, complete contributions conducted asynchronously by a potentially large, but actually small, group of self-selected experts." As code reviews are practiced in software projects with different settings, cultures, and policies, Rigby and Bird further investigated code review practices using a diverse set of open-source and industrial projects [159]. Despite differences among projects, they found that many characteristics of modern code reviews have converged to similar values, indicating general principles of modern code review practices. We summarize these convergent code review practices as the following.

- *Modern code reviews occur early, quickly, and frequently.* Traditional code inspections happen after finishing a major software component and often last for several weeks. In contrast, modern code reviews happen more frequently and quickly when software changes are committed. For example, the Apache project has review intervals between a few hours to a day. Most reviews are picked up within a few hours among all projects, indicating that reviewers are regularly watching and performing code reviews [159].
- *Modern code reviews often examine small program changes.* During code reviews, the median size of software change varies from 11 to 32 changed lines. The change size is larger in industrial projects, e.g, 44 lines in Android, 78 lines in Chrome, but still much smaller than code inspections, e.g., 263 lines in Lucent. Such small changes facilitate developers to constantly review changes and thus keep up-to-date with the activities of their peers.
- *Modern code reviews are conducted by a small group of self-selected reviewers.* In OSS projects, no reviews are assigned and developers can select the changes of interest to review. Program changes and review discussions are broadcast to a large group of stakeholders but only a small number of developers periodically participate in code reviews. In industrial projects, reviews are assigned in a mixed manner—the author adds a group of reviewer candidates and individuals from the group then select changes based on their interest and expertise. On average, two reviewers find an optimal number of defects [159].

- *Modern code reviews are often tool-based.* There is a clear trend towards utilizing review tools to support review tasks and communication. Back in 2008, code reviews in OSS projects were often email-based due to a lack of tool support [160]. In 2013 study, some OSS projects and all industrial projects that they studied used a review tool [159]. More recently, popular OSS hosting services such as GitHub and BitBucket have integrated lightweight review tools to assign reviewers, enter comments, and record discussions. Compared with email-based reviews and traditional software inspections, tool-based reviews provide the benefits of traceability.
- *Although the initial purpose of code review is to find defects, recent studies find that the practices and actual outcomes are less about finding defects than expected.* A study of code reviews at Microsoft found that only a small portion of review comments were related to defects, which were mainly about small, low-level logical issues [17]. Rather, code review provides a spectrum of benefits to software teams, such as knowledge transfer, team awareness, and improved solutions with better practices and readability.

4.1.1 Commercial Code Review Tools. There is a proliferation of review tools, e.g., Phabricator,[1] Gerrit,[2] CodeFlow,[3] Crucible,[4] and Review Board.[5] We illustrate CodeFlow, a collaborative code review tool at Microsoft. Other review tools share similar functionality as CodeFlow.

To create a review task, a developer uploads changed files with a short description to CodeFlow. Reviewers are then notified via email and they can examine the software change in CodeFlow. Figure 10 shows the desktop window of CodeFlow. It includes a list of changed files under review (A), the reviewers and their status (B), the highlighted diff in a changed file (C), a summary of all review comments and their status (D), and the iterations of a review (E). If a reviewer would like to provide feedback, she can select a change and enter a comment which is overlayed with the selected change (F). The author and other reviewers can follow up the discussion by entering comments in the same thread. Typically, after receiving feedback, the author may revise the change accordingly and submit the updated change for additional feedback, which constitutes another review cycle and is termed as an *iteration*. In Figure 10-E, there are five iterations. CodeFlow assigns a status label to each review comment to keep track of the progress. The initial status is "Active" and can be changed to "Pending", "Resolved", "Won't Fix", and "Closed" by anyone. Once a reviewer is satisfied with the updated changes, she can indicate this by setting their status to "Signed Off". After enough reviewers signed off—sign-off policies vary by team—the author can commit the changes to the source repository.

[1] http://phabricator.org
[2] http://code.google.com/p/gerrit/
[3] http://visualstudioextensions.vlasovstudio.com/2012/01/06/codeflow-code-review-tool-for-visual-studio/
[4] https://www.atlassian.com/software/crucible
[5] https://www.reviewboard.org/

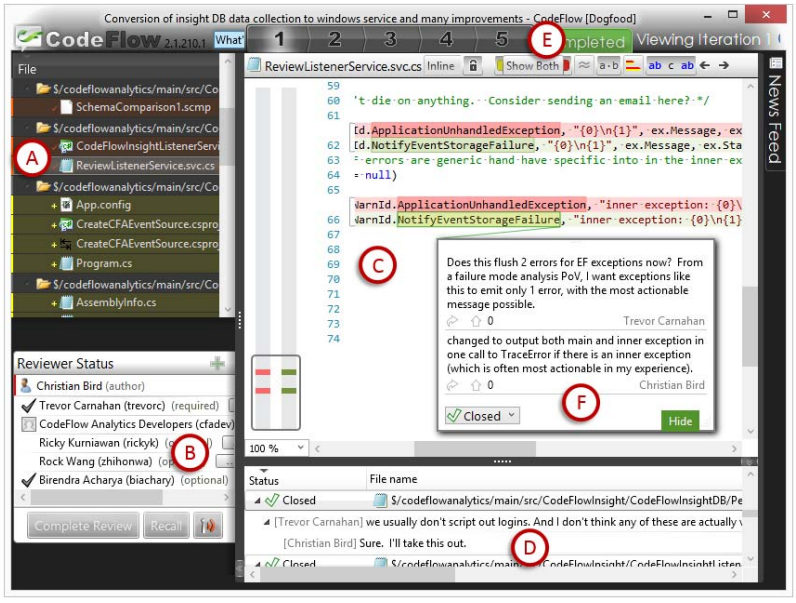


Fig. 10. Example of Code Review using CodeFlow [27]

Commercial code review tools facilitate management of code reviews but do not provide deep support for change comprehension. According to Bachhelli et al. [17], understanding program changes and their contexts remains a key challenge in modern code review. Many interviewees acknowledged that it is difficult to understand the rationale behind specific changes. All commercial review tools show the highlighted *textual, line-level diff* of a changed file. However, when the code changes are distributed across multiple files, developers find it difficult to inspect code changes [48]. This obliges reviewers to read changed lines file by file, even when those cross-file changes are done systematically to address the same issue.

4.1.2 Change Decomposition. Prior studies also observe that developers often package program changes of multiple tasks to a single code review [70,90,135]. Such large, unrelated changes often lead to difficulty in inspection, since reviewers have to mentally "untangle" them to figure out which subset addresses which issue. Reviewers indicated that they can better understand small, cohesive changes rather than large, tangled ones [160]. 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.*"[6] Among change decomposition techniques [21, 182], we discuss a representative technique called ClusterChanges.

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

ClusterChanges is a lightweight static analysis technique for decomposing large changes [21]. The insight is that program changes that address the same issue can be related via implicit dependency such as *def-use* relationship. For example, if a method definition is changed in one location and its call-sites 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.

- **Def-use relation**. If the definition of a method or a 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 field defined within the change-set are changed, these changes are considered related.
- **Enclosing relation**. Program changes in the same method are considered related, under the assumption that (1) program changes to the same method are often related, and (2) 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 of *miscellaneous changes*.

4.1.3 Refactoring Aware Code Review. Identifying which refactorings happened between two program versions is an important research problem, because inferred refactorings can help developers understand software modifications made by other developers during peer code reviews. Reconstructed refactorings can be used to update client applications that are broken due to refactorings in library components. Furthermore, they can be used to study the effect of refactorings on software quality empirically when the documentation about past refactorings is unavailable in software project histories.

Refactoring reconstruction techniques compare the old and new program versions and identify corresponding entities based on their name similarity and structure similarity [43, 44, 120, 199, 217]. Then based on how basic entities and relations changed from one version to the next, concrete refactoring type and locations are inferred. For example, Xing et al.'s approach [202] UMLDiff extracts class models from two versions of a program, traverses the two models, and identifies corresponding entities based on their name similarity and structure similarity (i.e., similarity in type declaration and uses, field accesses, and

method calls). Xing et al. later presented an extended approach to refactoring reconstruction based on change-facts *queries* [203]. They first extract facts regarding design-level entities and relations from each individual source code version. These facts are then pairwise compared to determine how the basic entities and relations have changed from one version to the next. Finally, queries corresponding to well-known refactoring types are applied to the change-facts database to find concrete refactoring instances. Among these refactoring reconstruction techniques, we introduce a representative example of refactoring reconstruction, called RefFinder in details [94, 151].

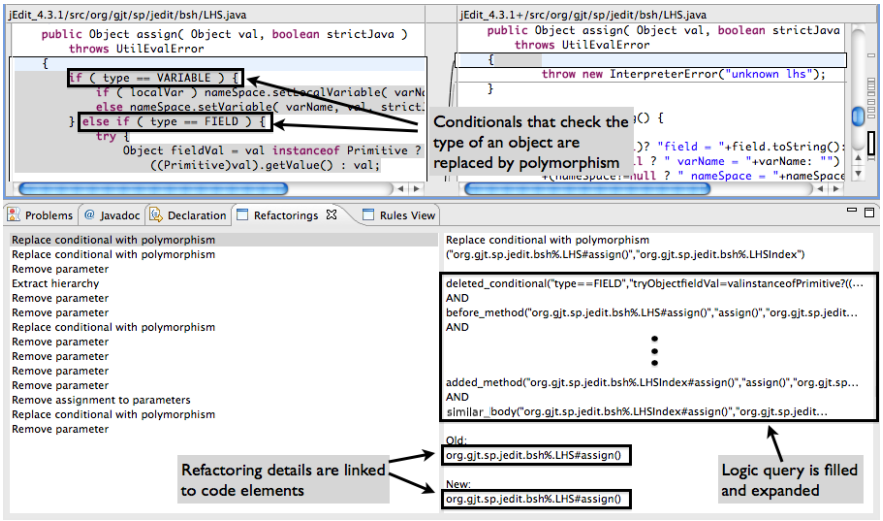


Fig. 11. RefFinder infers a *replace conditionals with polymorphism* refactoring from change facts *deleted_conditional*, *after_subtype*, *before_method*, *added_method* and *similar_body*. [94]

Example: RefFinder. *RefFinder* is a logic-query based approach for inferring various types of refactorings in Fowler's catalog [151]. It first encodes each refactoring type as a structural constraint on the program before and after the refactoring in a template logic rule. It then compares the syntax tree of each version to compute change facts such as `added_subtype`, at the level of code elements (packages, types, methods, and fields), structural dependencies (subtyping, overriding, method-calls, and field-accesses), and control constructs (while, if-statements, and try-catch blocks). It determines a refactoring inference order to find atomic refactorings before composite refactorings.

For example, consider an *extract superclass* refactoring that extracts common functionality in different classes into a superclass. It finds each *pull-up-method* refactoring and then tests if they combine to an *extract superclass* refac-

toring. For each refactoring rule, it converts the antecedent of the rule to a logic query and invokes the query on the change-fact database. If the query returns the constant bindings for logic variables, it creates a new logic fact for the found refactoring instance and *writes* it to the fact-base. For example, by invoking a query `pull_up_method(?method, ?class, ?superclass)` ∧ `added_type(?superclass)`, it finds a concrete instance of *extract superclass* refactoring. Figure 12 illustrates an example refactoring reconstruction process.

pull_up_method	You have methods with identical results on subclasses; move them to the superclass.
template	deleted_method(m1, n, t1) ∧ after_subtype(t2, t1) ∧ added_method(m1, n, t2) ⇒ pull_up_method(n, t1, t2)
logic rules	pull_up_method(m1, t1, t2) ∧ added_type(t2) ⇒ extract_superclass(t1,t2)

code example	+public class Customer{
	+ chargeFor(start:Date, end:Date) { ... } ...}
	-public class RegularCustomer{
	+public class RegularCustomer extends Customer{
	- chargeFor(start:Date, end:Date){ ... } ...}
	+public class PreferredCustomer extends Customer{
	- chargeFor(start:Date, end:Date){ ... } // deleted ... }

found refactorings	pull_up_method("chargeFor", "RegularCustomer", "Customer") pull_up_method("chargeFor", "PreferredCustomer", "Customer") extract_superclass("RegularCustomer", "Customer") extract_superclass("PreferredCustomer", "Customer")

Fig. 12. Reconstruction of *Extract Superclass* Refactoring

This approach has two advantages over other approaches. First, it analyzes the body of methods including changes to the control structure within method bodies. Thus, it can handle the detection of refactorings such as *replacing conditional code with polymorphism*. Second, it handles composite refactorings, since the approach reasons about which constituent refactorings must be detected first and reason about how those constituent refactorigs are knit together to detect higher-level, composite refactorings. It supports 63 out of 72 refactoring types in Fowler's catalog. As shown in Figure 11, RefFinder visualizes the reconstructed refactorings as a list. The panel on the right summarizes the key details of the selected refactoring and allows the developer quickly navigate to the associated code fragments.

4.1.4 Change Conflicts, Interference, and Relevance. As development teams become distributed, and the size of the system is often too large to be handled by a few developers, multiple developers often work on the same module at the same time. In addition, the market-pressure to develop new features or products makes parallel development no longer an option. A study on a subsystem of Lucent 5ESS telephone found that 12.5% of all changes are made by different developers to the same files within 24 hours, showing a high degree of parallel updates [150]. A subsequent study found that even though only 3% of the changes made within 24 hours by different developers physically overlapped each other's changes at a textual level but there was a high degree of semantic

interference among parallel changes at a data flow analysis level (about 43% of revisions made within one week). They also discovered a significant correlation between files with a high degree of parallel development and the number of defects [169].

Most version control systems are only able to detect most simple types of conflicting changes—changes made on top of other changes [126]. To detect changes that indirectly conflict with each other, some define the notion of *semantic interference* using program slicing on program dependence graphs, and integrate non-interfering versions only if there is no overlap between program slices [73]. As another example, some define semantic interference as the overlap between the data-dependence based impact sets of parallel updates [169].

4.1.5 Detecting and Preventing Inconsistent Changes to Clones. Code cloning often requires similar but not identical changes to multiple parts of the system [97] and cloning is an important source of bugs. In 65% of the ported code, at least one identifier is renamed, and in 27% cases at least one statement is inserted, modified, or deleted [114]. An incorrect adaptation of ported code often leads to porting errors [82]. Interviews with developers confirm that inconsistencies in clones are indeed bugs and report that *"nearly every second, unintentional inconsistent changes to clones lead to a fault."* [86]. Several techniques find inconsistent changes to similar code fragments by tracking copy-paste code and by comparing the corresponding code and its surrounding contexts [78, 81, 82, 114, 157]. Below, we present a representative technique, called Critics.

Example: Critics. Critics allows reviewers to interactively detect inconsistent changes through template-based code search and anomaly detection [215]. 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.

Figure 13 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 13). 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 in Figure 13) visualizes the change template of the selected

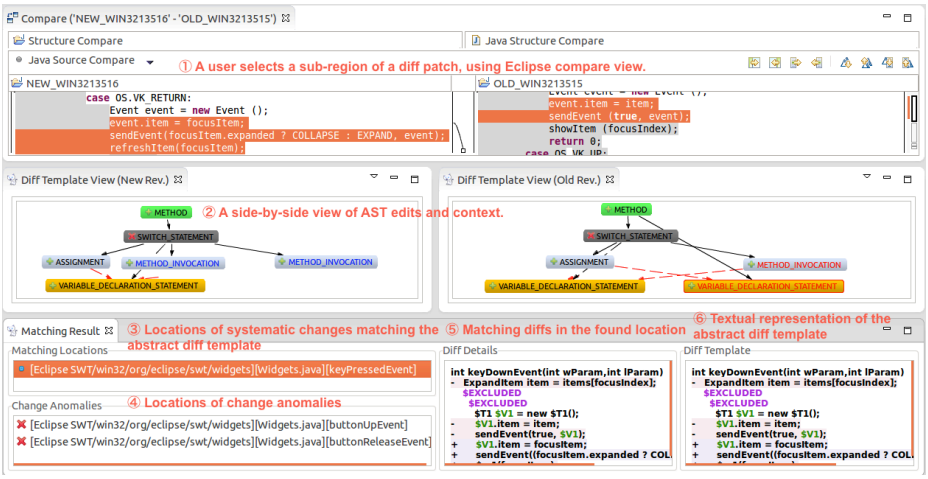


Fig. 13. A screen snapshot of CRITICS's Eclipse plugin and its features

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 13) shows the change template in a unified format. The Matching Result View summarizes the consistent changes as *similar changes* (see ③ in Figure 13) and inconsistent ones as *anomalies* (see ④ in Figure 13).

4.2 Program Differencing

Program differencing serves as a basis for analyzing software changes between program versions. The program differencing problem is a dual problem of code matching, and is defined as follows.

Suppose that a program P' is created by modifying P. Determine the difference Δ between P and P'. For a code fragment $c' \in P'$, determine whether $c' \in \Delta$. If not, find c''s corresponding origin c in P.

A code fragment in the new version either contributes to the difference or comes from the old version. If the code fragment has a corresponding origin in the old version, it means that it does not contribute to the difference. Thus, finding the delta between two versions is the same problem as finding corresponding code fragments between two versions.

Suppose that a programmer inserts if-else statements in the beginning of the method `m_A` and reorders several statements in the method `m_B` without changing semantics (see Figure 14). An intuitively correct matching technique should produce [(p0-c0), (p1-c2), (p2-c3), (p4-c4), (p4-c6), (p5-c7), (p6-c9), (p7-c8), (p8-c10), (p9-c11)] and identify that c1 and c5 are added.

Matching code across program versions poses several challenges. First, previous studies indicate that programmers often disagree about the origin of code

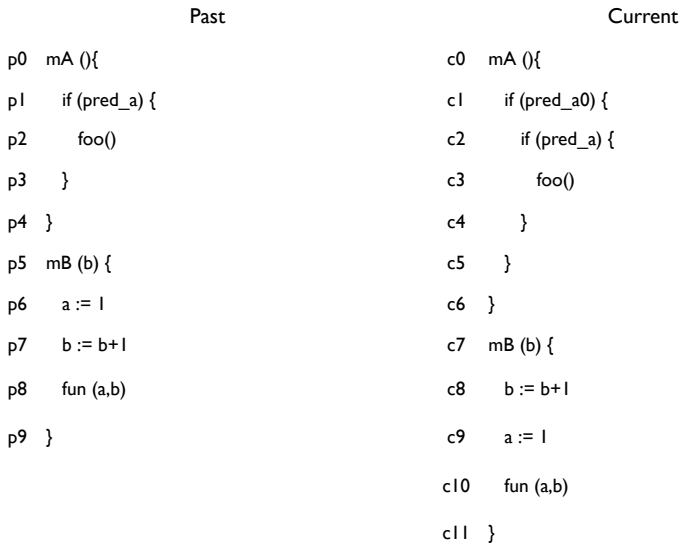


Fig. 14. Example code change

elements; low inter-rater agreement suggests that there may be no ground truth in code matching [100]. Second, renaming, merging, and splitting of code elements that are discussed in the context of refactoring reconstruction in Section 4.1.3 make the matching problem non-trivial. Suppose that a file `PElmtMatch` changed its name to `PMatching`; a procedure `matchBlck` is split into two procedures `matchDBlck` and `matchCBlck`; and a procedure `matchAST` changed its name to `matchAbstractSyntaxTree`. The intuitively correct matching technique should produce [(`PElmtMatch, PMatching`), (`matchBlck, matchDBlck`), (`matchBlck, matchCBlck`), and (`matchAST, matchAbstractSyntaxTree`)], while simple name-based matching will consider `PMatching`, `matchDBlck`, `matchCBlck`, and `matchAbstractSyntaxTree` added and consider `PElmtMatch`, `matchBlck`, and `matchAST` deleted.

Existing code matching techniques usually employ syntactic and textual similarity measures to match code. They can be characterized by the choices of (1) an underlying program representation, (2) matching granularity, (3) matching multiplicity, and (4) matching heuristics. Below, we categorize program differencing techniques with respect to internal program representations, and we discuss seminal papers for each representation.

4.2.1 String and Lexical Matching. When a program is represented as a string, the best match between two strings is computed by finding the longest common subsequence (LCS) [16]. The LCS problem is built on the assumption that (1) available operations are addition and deletion, and (2) matched pairs cannot cross one another. Thus, the longest common subsequence does not necessarily include all possible matches when available edit operations include

copy, paste, and move. Tichy's *bdiff* [184] extended the LCS problem by relaxing the two assumptions above: permitting crossing block moves and not requiring one-to-one correspondence.

The line-level LCS implementation, *diff* [76] is fast, reliable, and readily available. Thus, it has served as a basis for popular version control systems such as CVS. Many evolution analyses are based on *diff* because they use version control system data as input. For example, identification of fix-inducing code snippets is based on line tracking *(file name:: function name:: line number)* backward from the moment that a bug is fixed [173]

The longest common subsequence algorithm is a dynamic programming algorithm with $O(mn)$ in time and space, when m is the line size of the past program and the n is the line size of the current program. The goal of LCS-based diff is to report the minimum number of line changes necessary to convert one file to another. It consists of two phases: (1) computing the length of LCS and (2) reading out the longest common subsequence using a backtrace algorithm. Applying LCS to the example in Figure 14 will produce the line matching of [(p0-c0), (p1-c1), (p2-c3), (p3-c5), (p4-c6), (p5-c7), (p6-c9), (p8-c10), (p9-c11)]. Due to the assumption of no crossing matches, LCS does not find (p7-c8). In addition, because the matching is done at the line level and LCS does not consider the syntactic structure of code, it produces a line-level match such as (p3-c5) that do not observe the matching block parentheses rule.

4.2.2 Syntax Tree Matching. For software version merging, Yang [207] developed an AST differencing algorithm. Given a pair of functions (f_T, f_R), the algorithm creates two abstract syntax trees T and R and attempts to match the two tree roots. Once the two roots match, the algorithm aligns T's subtrees $t_1, t_2, ..., t_i$ and R's subtrees $r_1, r_2, ...r_j$ using the LCS algorithm and maps subtrees recursively. This type of tree matching respects the parent-child relationship as well as the order between sibling nodes, but is very sensitive to changes in nested blocks and control structures because tree roots must be matched for every level. Because the algorithm respects parent-child relationships when matching code, all matches are observe the syntactic boundary of code and the matching block parentheses rule. Similar to LCS, because Yang's algorithm aligns subtrees at the current level by LCS, it cannot find crossing matches caused by code reordering. Furthermore, the algorithm is very sensitive to tree level changes or insertion of new control structures in the middle, because Yang's algorithm performs top-down AST matching.

As another example, Change Distiller [56] uses an improved version of Chawathe et al.'s hierarchically structured data comparison algorithm [32]. Change Distiller takes two abstract syntax trees as input and computes basic tree edit operations such as *insert, delete, move* or *update* of tree nodes. It uses *bi-gram string similarity* to match source code statements such as method invocations and uses *subtree similarity* to match source code structures such as if-statements. After identifying tree edit operations, Change Distiller maps each tree-edit to an atomic AST-level change type.

4.2.3 Control Flow Graph Matching. Laski and Szermer [110] first developed an algorithm that computes one-to-one correspondences between CFG nodes in two programs. This algorithm reduces a CFG to a series of single-entry, single-exit subgraphs called hammocks and matches a sequence of hammock nodes using a depth first search (DFS). Once a pair of corresponding hammock nodes is found, the hammock nodes are recursively expanded in order to find correspondences within the matched hammocks.

Jdiff [14] extends Laski and Szermer's (LS) algorithm to compare Java programs based on an enhanced control flow graph (ECFG). *Jdiff* is similar to the LS algorithm in the sense that hammocks are recursively expanded and compared, but is different in three ways: First, while the LS algorithm compares hammock nodes by the name of a start node in the hammock, *Jdiff* checks whether the ratio of unchanged-matched pairs in the hammock is greater than a chosen threshold in order to allow for flexible matches. Second, while the LS algorithm uses DFS to match hammock nodes, *Jdiff* only uses DFS up to a certain look-ahead depth to improve its performance. Third, while the LS algorithm requires hammock node matches at the same nested level, *Jdiff* can match hammock nodes at a different nested level; thus, *Jdiff* is more robust to addition of while loops or if-statements at the beginning of a code segment. *Jdiff* has been used for regression test selection [146] and dynamic change impact analysis [15]. Figure 15 shows the code example and corresponding extended control flow graph representations in Java. Because their representation and matching algorithm is designed to account for dynamic dispatching and exception handling, it can detect changes in the method body of `m3 (A a)`, even though it did not have any textual edits: (1) `a.m1()` calls the method definition `B.m()` for the receiver object of type B and (2) when the exception type `E3` is thrown, it is caught by the catch block `E1` instead of the catch block `E2`.

CFG-like representations are commonly used in regression test selection research. Rothermel and Harrold [165] traverse two CFGs in parallel and identify a node with unmatched edges, which indicates changes in code. In other words, their algorithm stops parallel traversal as soon as it detects changes in a graph structure; thus, this algorithm does not produce deep structural matches between CFGs. However, traversing graphs in parallel is still sufficient for the regression testing problem because it conservatively identifies affected test cases. In practice, regression test selection algorithms [68, 146] require that syntactically changed classes and interfaces are given as input to the CFG matching algorithm.

4.2.4 Program Dependence Graph Matching. There are several program differencing algorithms based on a program dependence graph [25, 72, 79].

Horwitz [72] presents a semantic differencing algorithm that operates on a program representation graph (PRG) which combines features of program dependence graphs and static single assignment forms. In her definition, semantic equivalence between two programs $P1$ and $P2$ means that, for all states σ such that $P1$ and $P2$ halt, the sequence of values produced at $c1$ is identical to the

Program P

```
public class A {
     void m1() {...}
}

public class B extends A {

     void m2() {...}
}

public class E1 extends Exception {}
public class E2 extends E1 {}
public class E3 extends E2 {}

public class D {
     void m3(A a) {
          a.m1();
          try {
             throw new E3();
          }
          catch (E2 e) {...}
          catch (E1 e) {...}
     }
}
```

Program P'

```
public class A {
     void m1() {...}
}

public class B extends A {
     void m1() {...}
     void m2() {...}
}

public class E1 extends Exception {}
public class E2 extends E1 {}
public class E3 extends E1 {}

public class D {
     void m3(A a) {
          a.m1()
          try {
               throw new E3();
          }
          catch(E2 e) {...}
          catch(E1 e) {...}
     }
}
```

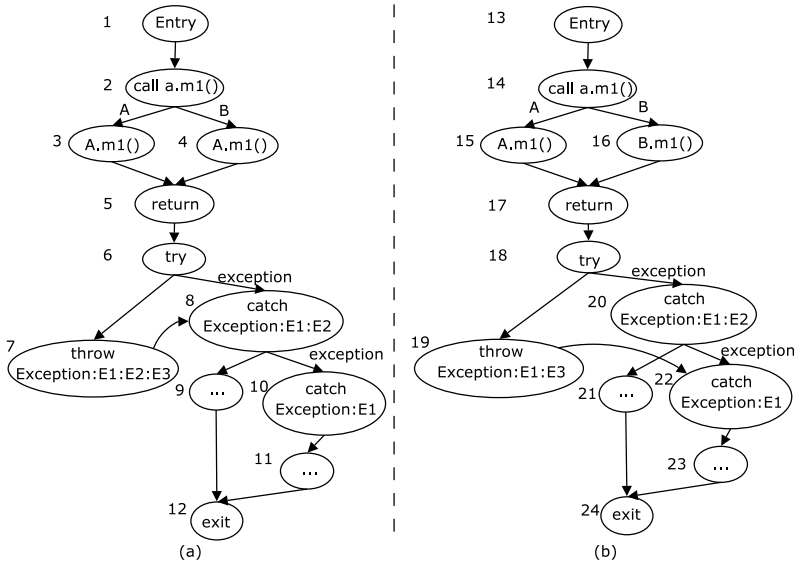


Fig. 15. JDiff Change Example and CFG representations [15]

sequence of values produced at $c2$ where $c1$ and $c2$ are corresponding locations. Horwitz uses Yang's algorithm [208] to partition the vertices into a group of semantically equivalent vertices based on three properties, (1) the equivalence of their operators, (2) the equivalence of their inputs, (3) the equivalence of the predicates controlling their evaluation. The partitioning algorithm starts with an initial partition based on the operators used in the vertices. Then by following flow dependence edges, it refines the initial partition if the successors of the same group are not in the same group. Similarly, it further refines the partition by following control dependence edges. If two vertices in the same partition are textually different, they are considered to have only a *textual change*. If two vertices are in different partitions, they have a *semantic change*. After the partitioning phase, the algorithm finds correspondences between $P1$'s vertices and $P2$'s vertices that minimize the number of semantically or textually changed components of $P2$. In general, PDG-based algorithms are not applicable to popular modern program languages because they can run only on a limited subset of C-like languages without global variables, pointers, arrays, or procedures.

4.2.5 Related Topics: Model Differencing and Clone Detection. A clone detector is simply an implementation of an arbitrary equivalence function. The equivalence function defined by each clone detector depends on a program representation and a comparison algorithm. Most clone detectors are heavily dependent on (1) hash functions to improve performance, (2) parametrization to allow flexible matches, and (3) thresholds to remove spurious matches. A clone detector can be considered as a many-to-many matcher based solely on content similarity heuristics.

In addition to these, several differencing algorithms compare model elements [47,144,177,202]. For example, UMLdiff [202] matches methods and classes between two program versions based on their name. However, these techniques assume that no code elements share the same name in a program and thus use name similarity to produce one-to-one code element matches. Some have developed a general, meta-model based, configurable program differencing framework [4, 167]. For example, SiDiff [167, 186] allows tool developers to configure various matching algorithms such as identity-based matching, structure-based matching, and signature-based matching by defining how different types of elements need to be compared and by defining the weights for computing an overall similarity measure.

4.3 Recording Changes: Edit Capture and Replay

Recorded change operations can be used to help programmers reason about software changes. Several editors or integrated development environment (IDE) extensions capture and replay keystrokes, editing operations, and high-level update commands to use the recorded change information for intelligent version merging, studies of programmers' activities, and automatic updates of client applications. When recorded change operations are used for helping programmers

reason about software changes, this approach's limitation depends on the granularity of recorded changes. If an editor records only keystrokes and basic edit operations such as cut and paste, it is a programmer's responsibility to raise the abstraction level by grouping keystrokes. If an IDE records only high-level change commands such as refactorings, programmers cannot retrieve a complete change history. In general, capturing change operations to help programmers reason about software change is *impractical* as this approach constrains programmers to use a particular IDE. Below, we discuss a few examples of recording change operations from IDEs:

Spyware is a representative example in this line of work [161]. It is a smalltalk IDE extension to capture AST-level change operations (creation, addition, removal and property change of an AST node) as well as refactorings. It captures refactorings during development sessions in an IDE rather than trying to infer refactorings from two program versions. Spyware is used to study when and how programmers perform refactorings, but such edit-capture-replay could be used for performing refactoring-aware version merging [46] or updating client applications due to API evolution [69].

5 An Organized Tour of Seminal Papers: III. Change Validation

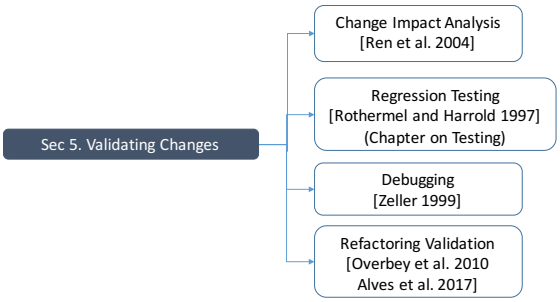


Fig. 16. Change Validation and Related Research Topics

After making software changes, developers must validate the correctness of updated software. Validation and verification is a vast area of research. In this section, we focus on techniques that aim to identify faults introduced due to software changes. As Chapter cross reference to testing discusses the history and seminal work on regression testing in details, we refer the interested readers to that chapter instead. Section 5.1 discusses Change Impact Analysis, which aims to determine the impact of source code edits on programs under test. Section 5.2 discusses how to localize program changes responsible for test failures. Section 5.3 discusses the techniques that are specifically designed to validate refactoring

edits under the assumption that software's external behavior should not change after refactoring.

5.1 Change Impact Analysis

Change Impact Analysis consists of a collection of techniques for determining the effects of source code modifications, and can improve programmer productivity by: (i) allowing programmers to experiment with different edits, observe the code fragments that they affect, and use this information to determine which edit to select and/or how to augment test suites, (ii) reducing the amount of time and effort needed in running regression tests, by determining that some tests are guaranteed not to be affected by a given set of changes, and (iii) reducing the amount of time and effort spent in debugging, by determining a safe approximation of the changes responsible for a given tests failure.

In this section, we discuss the seminal change impact analysis work, called Chianti that serve the both purposes of affected test identification and isolation of failure-inducing deltas. It uses a two-phase approach in Figure 17 [158].

In the first phase, to identify which test cases a developer must rerun on the new version to ensure that all potential regression faults are identified, Chianti takes the old and new program versions P_o and P_n and an existing test suite T as inputs, and identify a set of atomic program changes at the level of methods, fields, and subtyping relationships. It then computes the profile of the test suite T on P_o in terms of dynamic call graphs and selects $T' \subset T$ that guarantees the same regression fault revealing capability between T and T'.

In the second phase, Chianti then first runs the selected test cases T' from the first phase on the new program version P_n and computes the profile of T' on P_n in terms of dynamic call graphs. It then uses both the atomic change set information together with dynamic call graphs to identify which subset of the delta between P_o and P_n led to the behavior differences for each failed test on P_n.

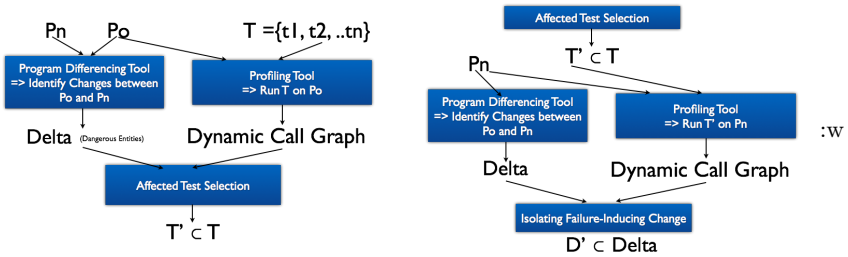


Fig. 17. Chianti Change Impact Analysis: identifying affected tests (left) and identifying affecting change (right) [158]

To represent atomic changes, Chianti compares the syntax tree of the old and new program versions and decomposes the edits into atomic changes at a

method and field level. Changes are then categorized as added classes (AC), deleted classes (DC), added methods (AM), deleted methods (DM), changed methods (CM), added fields (AF), deleted fields (DF), and lookup (i.e., dynamic dispatch) changes (LC). The LC atomic change category models changes to the dynamic dispatch behavior of instance methods. In particular, an LC change `LC(Y, X.m())` models the fact that a call to method `X.m()` on an object of type `Y` results in the selection of a different method call target.

For example, Figure 18 shows a software change example and corresponding lists of atomic changes inferred from AST-level comparison. An arrow from an atomic change $A1$ to an atomic change $A2$ indicates that $A2$ is dependent on $A1$. For example, the addition of the call `B.bar()` in method `B.foo()` is the method body change `CM(B.foo())` represented as ⑧. This change `8` requires the declaration of method `B.bar()` to exist first, i.e., `AM(B.bar())` represented as ⑥. This dependence is represented as an arrow from ⑥ to ⑧.

Phase I reports **affected tests**—a subset of regression tests relevant to edits. It identifies a test if its dynamic call graph on the old version contains a node that corresponds to a changed method (CM) or deleted method (DM) or or if the call graph contains an edge that corresponds to a lookup change (LC). Figure 18 also shows the dynamic call graph of each test for the old version (left) and the new version (right). Using the call graphs on the left, it is easy to see that: (i) `test1` is not affected, (ii) `test2` is affected because its call graph contains a node for `B.foo()`, which corresponds to ⑧, and (iii) `test3` is affected because its call graph contains an edge corresponding to a dispatch to method `A.foo()` on an object of type `C`, which corresponds to ④.

Phase II then reports **affecting changes**—a subset of changes relevant to the execution of affected tests in the new version. For example, we can compute the affecting changes for `test2` as follows. The call graph for `test2` in the edited version of the program contains methods `B.foo()` and `B.bar()`. These nodes correspond to ⑧ and ⑨ respectively. Atomic change ⑧ requires ⑥ and ⑨ requires ⑥ and ⑦. Therefore, the atomic changes affecting test2 are ⑥, ⑦, ⑧, and ⑨. Informally, this means that we can automatically determine that `test2` is affected by the addition of field `B.y`, the addition of method `B.bar()`, and the change to method `B.foo()`, but not on any of the other source code changes.

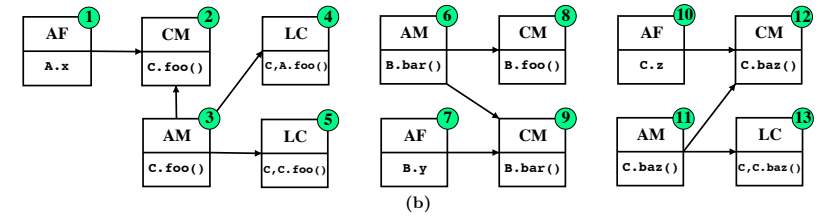
5.2 Debugging Changes

The problem of simplifying and isolating failure-inducing input is a long standing problem in software engineering. *Delta Debugging (DD)* addresses this problem by repetitively running a program with different sub-configurations (subsets) of the input to systematically isolate failure-inducing inputs [212, 213]. DD splits the original input into two halves using a binary search-like strategy and re-runs them. DD requires a test oracle function $test(c)$ that takes an input configuration c and checks whether running a program with c leads to a failure. If one of the two halves fails, DD recursively applies the same procedure for only that failure-inducing input configuration. On the other hand, if both halves pass, DD tries different sub-configurations by mixing fine-grained sub-configurations

```
class A {
  public A(){ }
  public void foo(){ }
  public int x;
}
class B extends A {
  public B(){ }
  public void foo(){ B.bar(); }
  public static void bar(){ y = 17; }
  public static int y;
}
class C extends A {
  public C(){ }
  public void foo(){ x = 18; }
  public void baz(){ z = 19; }
  public int z;
}
```

```
class Tests {
  public static void test1(){
    A a = new A();
    a.foo();
  }
  public static void test2(){
    A a = new B();
    a.foo();
  }
  public static void test3(){
    A a = new C();
    a.foo();
  }
}
```

(a)



(b)

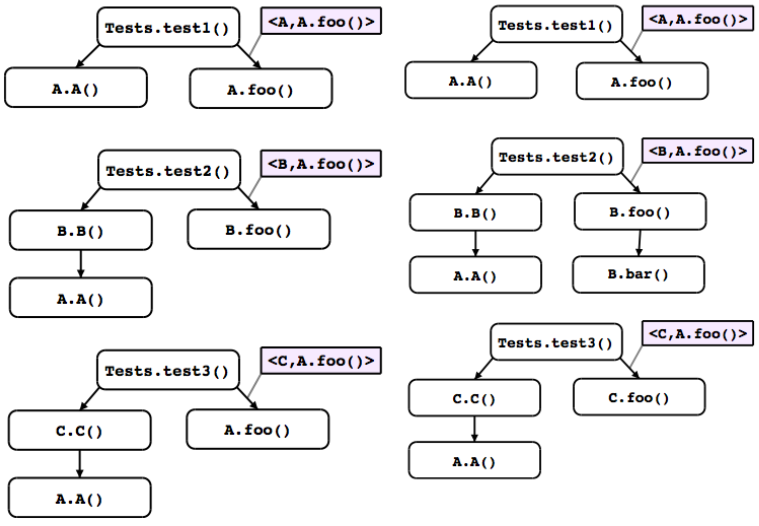


Fig. 18. Chianti change impact analysis

with larger sub-configurations (computed as the complement from the current configuration).

Under the assumption that failure is *monotone*—where C is a super set of all configurations, if a larger configuration c is successful, then any of its smaller sub-configurations c' does not fail, i.e., $\forall c \subset C \; (\; test(c) = \checkmark \rightarrow \forall c' \subset c \; (test(c') \neq \text{✗}))$, DD returns a minimal failure-inducing configuration.

This idea of Delta Debugging was applied to isolate failure-inducing changes. It considers all line-level changes between the old and new program version as the candidate set without considering compilation dependences among those changes. In Zeller's seminal paper, *"yesterday, my program worked, but today, it does not, why?"* Zeller demonstrates the application of DD to isolate program edits responsible for regression failures [212]. DDD 3.1.2, released in December, 1998, exhibited a nasty behavioral change: When invoked with a the name of a non-existing file, DDD 3.1.2 dumped core, while its predecessor DDD 3.1.1 simply gave an error message. The DDD configuration management archive lists 116 logical changes between the 3.1.1 and 3.1.2 releases. These changes were split into 344 textual changes to the DDD source. After only 12 test runs and 58 minutes, the failure-inducing change was found:

```
diff -r1.30 -r1.30.4.1 ddd/gdbinit.C
295,296c296
<
< --- >
string classpath =
getenv("CLASSPATH") != 0 ? getenv("CLASSPATH") : ".";
string classpath = source view->class path();
```

When called with an argument that is not a file name, DDD 3.1.1 checks whether it is a Java class; so DDD consults its environment for the class lookup path. As an "improvement", DDD 3.1.2 uses a dedicated method for this purpose. Unfortunately, the source view pointer used is initialized only later, resulting in a core dump.

Spectra-based fault localization. Spectrum-based fault localization techniques such as Tarantula [85] statistically compute suspiciousness scores for statements based on execution traces of both passed and failed test cases, and rank potential faulty statements based on the derived suspiciousness scores. Researchers have also introduced more suspiciousness computation measures to the realm of fault localization for localizing faulty statements [117,137] and also developed various automated tool-sets which embodies different spectrum-based fault localization techniques [1, 80]. However, such spectrum-based fault localization techniques are not scalable to large evolving software systems, as they compute spectra on all statements in each program version and do not leverage information about program edits between the old and new versions.

To address this problem, FaultTracer [214] combines Chianti-style change impact analysis and Tarantula-style fault localization. To present a ranked list of potential failure-inducing edits, FaultTracer applies a set of spectrum-based

ranking techniques to the affecting changes determined by Chianti-style change impact analysis. It uses a new enhanced call graph representation to measure test spectrum information directly for field-level edits and to improve upon the existing Chianti algorithm. The experimental results show that FaultTracer outperforms Chianti in selecting affected tests (slightly better) as well as in determining affecting changes (with an improvement of approximately 20%). By ranking the affecting changes using spectrum-based profile, it places a real regression fault within a few atomic changes, significantly reducing developers effort in inspecting potential failure-inducing changes.

5.3 Refactoring Validation

Unlike other types of changes, refactoring validation is a special category of change validation. By definition, refactoring must guarantee behavior preservation and thus the old version's behavior could be compared against the new version's behavior for behavior preservation. Regression testing is the most used strategy for checking refactoring correctness. However, a recent study finds that test suites are often inadequate [153] and developers may hesitate to initiate or perform refactoring tasks due to inadequate test coverage [98]. Soares et al. [175] design and implement SafeRefactor that uses randomly generated test suites for detecting refactoring anomalies.

Formal verification is an alternative for avoiding refactoring anomalies [127]. Some propose rules for guaranteeing semantic preservation [37], use graph rewriting for specifying refactorings [128], or present a collection of refactoring specifications, which guarantee the correctness by construction [147]. However, these approaches focus on improving the correctness of automated refactoring through formal specifications only. Assuming that developers may apply refactoring manually rather, Schaeffer et al. validate refactoring edits by comparing data and control dependences between two program versions [166].

RefDistiller is a static analysis approach [12, 13] to support the inspection of manual refactorings. It combines two techniques. First, it applies predefined templates to identify potential missed edits during manual refactoring. Second, it leverages an automated refactoring engine to identify extra edits that might be incorrect, helping to determine the root cause of detected refactoring anomalies. GhostFactor [58] checks the correctness of manual refactoring, similar to RefDistiller. Another approach by Ge and Murphy-Hill [50] helps reviewers by identifying applied refactorings and letting developers examine them in isolation by separating pure refactorings.

6 Future Directions and Open Problems

Software maintenance is challenging and time-consuming. Albeit various research and existing tool support, the global cost of debugging software has risen up to $312 billion annually [3]. The cost of software maintenance is rising dramatically and has been estimated as more than 90% of the total cost for software [51].

Software evolution research still has a long future ahead, because there are still challenges and problems that cost developers a lot of time and manual effort. In this section, we highlight some key issues in change comprehension and suggestion.

6.1 Change Comprehension

Understanding software changes made by other people is a difficult task, because it requires not only the domain knowledge of the software under maintenance, but also the comprehension of change intent, and the interpretation of mappings between the program semantics of applied changes and those intent. Existing change comprehension tools discussed in Section 4.1 and program differencing tools discussed in Section 4.2 mainly present the textual or syntactical differences between the before- and after- versions of software changes. Current large-scale empirical studies on code changes discussed in Sections 3.1, 3.2, 3.3 and 3.4 also mainly focus on textual or syntactical notion of software changes. However, there is no tool support to automatically summarize the semantics of applied changes, or further infer developers' intent behind the changes.

The new advanced change comprehension tools must assist software professionals in two aspects. First, by summarizing software changes with a natural language description, these tools must produce more meaningful commit messages when developers check in their program changes to software version control systems (e.g., SVN, Git) to facilitate other people (e.g., colleagues and researchers) to mine, comprehend, and analyze applied changes more precisely [70]. Second, the generated change summary must provide a second opinion to developers of the changes, and enable them to easily check whether the summarized change description matches their actual intent. If there is a mismatch, developers should carefully examine the applied changes and decide whether the changes reflect realize their original intent.

To design and implement such advanced change comprehension tools, researchers must address several challenges.

1. How should we correlate changes applied in source code, configuration files, and databases to present all relevant changes and their relationships as a whole? For instance, how can we explain why a configuration file is changed together with a function's code body? How are the changes in a database schema correspond to source code changes?
2. How should we map concrete code changes or abstract change patterns to natural language descriptions? For instance, when complicated code changes are applied to improve a program's performance, how can we detect or reveal that intent? How should we differentiate between different types of changes when inferring change intent or producing natural language descriptions accordingly?
3. When developers apply multiple kinds of changes together, such as refactoring some code to facilitate feature addition, we can we identify the boundary between the different types of changes? How can we summarize the changes

in a meaningful way so that both types of changes are identified, and the connection between them is characterized clearly?

To solve these challenges, we may need to invent new program analysis techniques to correlate changes, new change interpretation approaches to characterize different types of changes, and new text mining and natural language processing techniques to map changes to natural language descriptions.

6.2 Change Suggestion

Compared with understanding software changes, applying changes is even more challenging, and can cause serious problems if changes are wrongly applied. Empirical studies showed that 15-70% of the bug fixes applied during software maintenance were incorrect in their first release [171,210], which indicates a desperate need for more sophisticated change suggestion tools. Below we discuss some of the limitations of existing automatic tool support, and also suggest potential future directions.

Corrective Change Suggestion. Although various bug fix and program repair tools discussed in Section 3.1 detect different kinds of bugs or even suggest bug fixes, the suggested fixes are usually relatively simple. They may focus on single-line bug fixes, multiple if-condition updates, missing APIs to invoke, or similar code changes that are likely to be applied to similar code snippets. However, no existing approach can suggest a whole missing `if`-statement or `while`-loop, neither can they suggest bug fixes that require declaring a new method and inserting the invocation to the new method in appropriate code locations.

Adaptive Change Suggestion.Existing adaptive change support tools discussed in Section 3.2 allow developers to migrate programs between specific previously known platforms (e.g., desktop and cloud). However, it is not easy to extend these tools when a new platform becomes available and people need to migrate programs from existing platforms to the new one. Although cross-platform software development tools can significantly reduce the necessity of platform-to-platform migration tools, these tools are limited to the platforms for which they are originally built. When a new platform becomes available, these tools will undergo significant modifications to support the new platform. In the future, we need extensible program migration frameworks, which will automatically infer program migration transformations from the concrete migration changes manually applied by developers, and then apply the inferred transformations to automate other migration tasks for different target platforms. With such frameworks, developers will not need to manually apply repetitive migration changes.

Perfective Change Suggestion. There are some programming paradigms developed (e.g., AOP and FOP discussed in Section 3.3), which facilitate developers to apply perfective changes to enhance or extend any existing software. However, there is no tool support to automatically suggest what perfective changes to apply and where to apply those changes. The main challenge of creating such tools is that unlike other types of changes, perfective changes usually aim to introduce new features instead of modifying existing features. Without any hint

provided by developers, it is almost impossible for any tool to predict what new features to add to the software. However, when developers know what new features they want to add but do not know how to implement those features, some advanced tools can be helpful by automatically searching for relevant open source projects, identifying relevant code implementation for the queried features, or even providing customized change suggestion to implement the features and to integrate the features into existing software.

Preventive Change Suggestion. Although various refactoring tools discussed in Section 3.4 can automatically refactor code, all the supported refactorings are limited to predefined behavior-preserving program transformations. It is not easy to extend existing refactoring tools to automate new refactorings, especially when the program transformation involves modifications of multiple software entities (i.e., classes, methods, and fields). Some future tools should be designed and implemented to facilitate the extensions of refactoring capabilities. There are also some refactoring tools that suggest refactoring opportunities based on code smells. For instance, if there are many code clones in a codebase, existing tools can suggest a clone removal refactoring to reduce duplicated code. In reality, nevertheless, most of the time developers apply refactorings only when they want to apply bug fixes or add new features, which means that refactorings are more likely to be motivated by other kinds of changes instead of code smells and change history [172]. In the future, with the better change comprehension tools mentioned above, we may be able to identify the trends of developers' change intent in the past, and observe how refactorings were applied in combination with other types of changes. Furthermore, with the observed trends, new tools must be built to predict developers' change intent in future, and then suggest refactorings accordingly to prepare for the upcoming changes.

6.3 Change Validation

In terms of change validation discussed in Section 5, there is disproportionately more work being done in the area of validating refactoring (i.e., *preventative changes*), compared to other types of changes such as *adaptive* and *perfective* changes. Similarly, in the absence of adequate existing tests which helped to discover defects in the first place, it is not easy to validate whether *corrective changes* are applied correctly to fix the defects.

The reason why is that, with the exception of refactoring that has a canonical, straightforward definition of *behavior preserving modifications*, when it comes to other types of software changes, it is difficult to define the updated semantics of software systems. For example, when a developer adds a new feature, how can we know the desired semantics of the updated software?

This problem naturally brings up the needs of having the correct specifications of updated software and having easier means to write such specifications in the context of software changes. Therefore, new tools must be built to guide developers in writing software specifications for the changed parts of the systems. In particular, we see a new opportunity for tool support suggests the template for updated specifications by recognizing the type and pattern of program changes

to guide developers in writing updated specifications—Are there common specification patterns for each common type of software changes? Can we then suggest which specifications to write based on common types of program modifications such as API evolution? Such tool support must not require developers to write specifications from scratch but rather guide developers on which specific parts of software require new, updated specifications, which parts of software may need additional tests, and how to leverage those written specifications effectively to guide the remaining areas for writing better specifications. We envision that, with such tool support for reducing the effort of writing specifications for updated software, researchers can build change validation techniques that actively leverage those specifications. Such effort will contribute to expansion of change-type specific debugging and testing technologies.

Appendix

The following text box shows selected, recommended readings for understanding the area of software evolution.

Key References:

T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *ASE '04: Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pages 2–13, Washington, DC, USA, 2004. IEEE Computer Society.

A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*, pages 712–721. IEEE Press, 2013.

J. R. Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.

J. Henkel and A. Diwan. Catchup!: Capturing and replaying refactorings to support api evolution. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 274–283, New York, NY, USA, 2005. ACM.

M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 50:1–50:11, New York, NY, USA, 2012. ACM.

M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 187–196, New York, NY, USA, 2005. ACM.

K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1 –10. IEEE Press, September 2010.

X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 432–448, New York, NY, USA, 2004. ACM.

P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

A. Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE-7: Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–267, London, UK, 1999. Springer-Verlag.

References

1. http://pleuma.cc.gatech.edu/aristotle/Tools/tarantula/index.html.
2. ASM. http://asm.ow2.org.
3. Cambridge University Study States Software Bugs Cost Economy $312 Billion Per Year. http://markets.financialcontent.com/stocks/news/read/23147130/Cambridge_University_Study_States_Software_Bugs_Cost_Economy_$312_Billion_Per_Year.
4. *Eclipse EMF Compare Project description: http://www.eclipse.org/emft/projects/compare.*
5. Javassist. http://jboss-javassist.github.io/javassist/.
6. Pmd: http://pmd.sourceforge.net/.
7. The AspectJ Project. https://eclipse.org/aspectj/.
8. The Guided Tour of TXL. https://www.txl.ca/tour/tour1.html.
9. *Software Maintenance and Computers (Ieee Computer Society Press Tutorial).* Ieee Computer Society, 1990.
10. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
11. Iso/iec 14764:2006: Software engineering software life cycle processes maintenance. Technical report, ISO/IEC, 2006.
12. E. L. G. Alves, M. Song, and M. Kim. Refdistiller: A refactoring aware code review tool for inspecting manual refactoring edits. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 751–754, New York, NY, USA, 2014. ACM.
13. E. L. G. Alves, M. Song, T. Massoni, P. D. L. Machado, and M. Kim. Refactoring inspection support for manual refactoring edits. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2017.
14. T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *ASE '04: Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pages 2–13, Washington, DC, USA, 2004. IEEE Computer Society.
15. T. Apiwattanapong, A. Orso, and M. J. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 432–441, New York, NY, USA, 2005. ACM.
16. A. Apostolico and Z. Galil, editors. *Pattern matching algorithms.* Oxford University Press, Oxford, UK, 1997. program differencing LCS.
17. A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*, pages 712–721. IEEE Press, 2013.
18. M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial redesign of java software systems based on clone analysis. In *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, page 326, Washington, DC, USA, 1999. IEEE Computer Society.
19. M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings Seventh Working Conference on Reverse Engineering*, pages 98–107, 2000.
20. C. Y. Baldwin and K. B. Clark. *Design Rules: The Power of Modularity.* MIT Press, Cambridge, MA, USA, 1999.
21. M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proceedings of the*

37th International Conference on Software Engineering-Volume 1, pages 134–144. IEEE Press, 2015.
22. D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.*, 1(4):355–398, Oct. 1992.
23. L. A. Belady and M. M. Lehman. A model of large program development. *IBM Syst. J.*, 15(3):225–252, Sept. 1976.
24. M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th working conference on mining software repositories*, pages 202–211. ACM, 2014.
25. D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, 4(1):3–35, 1995.
26. M. Boshernitsan, S. L. Graham, and M. A. Hearst. Aligning development tools with the way programmers think about code changes. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 567–576, New York, NY, USA, 2007. ACM.
27. A. Bosu, M. Greiler, and C. Bird. Characteristics of useful code reviews: An empirical study at microsoft. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 146–156. IEEE, 2015.
28. S. Breu and T. Zimmermann. Mining aspects from version history. In *International Conference on Automated Software Engineering*, pages 221–230, 2006.
29. N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 47–52, New York, NY, USA, 2010. ACM.
30. G. Canfora, L. Cerulo, M. Cimitile, and M. Di Penta. Social interactions around cross-system bug fixings: the case of freebsd and openbsd. In *Proceeding of the 8th working conference on Mining software repositories*, MSR '11, pages 143–152, New York, NY, USA, 2011. ACM.
31. J. Carriere, R. Kazman, and I. Ozkaya. A cost-benefit framework for making architectural decisions in a business context. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 149–157, New York, NY, USA, 2010. ACM.
32. S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 493–504, New York, NY, USA, 1996. ACM.
33. A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, New York, NY, USA, 2001. ACM.
34. K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 359, Washington, DC, USA, 1996. IEEE Computer Society.
35. J. R. Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

36. J. R. Cordy. Exploring large-scale system similarity. using incremental clone detection and live scatterplots. In *ICPC 2011, 19th International Conference on Program Comprehension (to appear)*, 2011.
37. M. Cornélio, A. Cavalcanti, and A. Sampaio. Sound refactorings. *Science of Computer Programming*, 75(3):106–133, 2010.
38. B. E. Cossette and R. J. Walker. Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In *FSE '12 Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, New York, NY, USA, 2012. ACM.
39. R. Cottrell, J. J. C. Chang, R. J. Walker, and J. Denzinger. Determining detailed structural correspondence for generalization tasks. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 165–174, New York, NY, USA, 2007. ACM.
40. W. Cunningham. The wycash portfolio management system. In *OOPSLA '92: Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)*, pages 29–30, New York, NY, USA, 1992. ACM.
41. B. Dagenais, S. Breu, F. W. Warr, and M. P. Robillard. Inferring structural patterns for concern traceability in evolving software. In *ASE '07: Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 254–263, New York, NY, USA, 2007. ACM.
42. B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 481–490, New York, NY, USA, 2008. ACM.
43. S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 166–177, New York, NY, USA, 2000. ACM.
44. D. Dig and R. Johnson. Automated detection of refactorings in evolving components. In *ECOOP '06: Proceedings of European Conference on Object-Oriented Programming*, pages 404–428. Springer, 2006.
45. D. Dig and R. Johnson. How do apis evolve? a story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, 2006.
46. D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 427–436, 2007.
47. A. Duley, C. Spandikow, and M. Kim. Vdiff: a program differencing algorithm for verilog hardware description language. *Automated Software Engineering*, 19:459–490, 2012.
48. A. Dunsmore, M. Roper, and M. Wood. Object-oriented inspection in the face of delocalisation. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 467–476, New York, NY, USA, 2000. ACM. code inspection, code review, object-oriented, delocalized.
49. S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Trans. Softw. Eng.*, 27(1):1–12, Jan. 2001.
50. X. S. EmersonMurphy-Hill. Towards refactoring-aware code review. In *CHASE' 14: 7th International Workshop on Cooperative and Human Aspects of Software Engineering, co-located with 2014 ACM and IEEE 36th International Conference on Software Engineering*, 2014.

51. F. P. Engelbertink and H. H. Vogt. How to save on software maintenance costs. *Omnext white paper*, 2010.
52. D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, Berkeley, CA, USA, 2000. USENIX Association.
53. D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.
54. M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Syst. J.*, 38(2-3):258–287, 1999. code inspection, checklist.
55. M. Fischer, J. Oberleitner, J. Ratzinger, and H. Gall. Mining evolution data of a product family. In *MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, New York, NY, USA, 2005. ACM.
56. B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling—tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):18, November 2007.
57. J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *CSMR '09: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, pages 255–258, Washington, DC, USA, 2009. IEEE Computer Society.
58. X. Ge and E. Murphy-Hill. Manual refactoring changes with automated refactoring validation. In *Software Engineering (ICSE 2014) 36th International Conference on*. IEEE, 2014.
59. C. Görg and P. Weißgerber. Error detection by refactoring reconstruction. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.
60. W. Griswold. Coping with crosscutting software changes using information transparency. In *Reflection 2001: The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 250–265. Springer, 2001.
61. W. G. Griswold. *Program Restructuring As an Aid to Software Maintenance*. PhD thesis, Seattle, WA, USA, 1992. UMI Order No. GAX92-03258.
62. W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *WPC '96: Proceedings of the 4th International Workshop on Program Comprehension*, page 144, Washington, DC, USA, 1996. IEEE Computer Society.
63. Y.-G. Guéhéneuc and H. Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, TOOLS '01, pages 296–, Washington, DC, USA, 2001. IEEE Computer Society.
64. Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. B. Da Silva, A. L. M. Santos, and C. Siebra. Tracking technical debt - an exploratory case study. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 528 –531, september.
65. Y. Guo, C. Seaman, N. Zazworka, and F. Shull. Domain-specific tailoring of code smells: an empirical study. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 167–170, New York, NY, USA, 2010. ACM.

66. M. Harman. The current state and future of search based software engineering. In *International Conference on Software Engineering*, pages 342–357, 2007.
67. W. Harrison, H. Ossher, S. Sutton, and P. Tarr. Concern modeling in the concern manipulation environment. In *Proceedings of the 2005 Workshop on Modeling and Analysis of Concerns in Software*, pages 1–5. ACM Press New York, NY, USA, 2005.
68. M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 312–326, New York, NY, USA, 2001. ACM.
69. J. Henkel and A. Diwan. Catchup!: Capturing and replaying refactorings to support api evolution. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 274–283, New York, NY, USA, 2005. ACM.
70. K. Herzig and A. Zeller. The impact of tangled code changes. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 121–130. IEEE, 2013.
71. Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Refactoring support based on code clone analysis. In *PROFES '04: Proceedings of 5th International Conference on Product Focused Software Process Improvement, Kausai Science City, Japan, April 5-8, 2004*, pages 220–233, 2004.
72. S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 1990. ACM.
73. S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, 1989.
74. K. Hotta, Y. Higo, and S. Kusumoto. Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 53–62. IEEE, 2012.
75. D. Hou and X. Yao. Exploring the intent behind api evolution: A case study. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pages 131–140, Washington, DC, USA, 2011. IEEE Computer Society.
76. J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, May 1977.
77. C. Izurieta and J. M. Bieman. How software designs decay: A pilot study of pattern evolution. In *ESEM*, pages 449–451, 2007.
78. P. Jablonski and D. Hou. Cren: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the ide. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, eclipse '07, pages 16–20, New York, NY, USA, 2007. ACM.
79. D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, pages 243–252, Washington, DC, USA, 1994. IEEE Computer Society.
80. T. Janssen, R. Abreu, and A. Gemund. Zoltar: A toolset for automatic fault localization. In *Proc. of ASE*, pages 662–664. IEEE Computer Society, 2009.
81. L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE '07: Proceedings of the 29th International*

Conference on Software Engineering, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.
82. L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 55–64, New York, NY, USA, 2007. ACM.
83. P. M. Johnson. Reengineering inspection. *Communications of the ACM*, 41(2):49–52, 1998.
84. R. Johnson. Beyond behavior preservation. Microsoft Faculty Summit 2011, Invited Talk, July 2011.
85. J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.
86. E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, Washington, DC, USA, 2009. IEEE Computer Society.
87. N. Juillerat and B. Hirsbrunner. Toward an implementation of the" form template method" refactoring. In *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*, pages 81–90. IEEE, 2007.
88. Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proceedings of the International Conference on Software Maintenance (ICSM 2002)*, pages 576–585. IEEE Computer Society, 2002.
89. Y. Kataoka, D. Notkin, M. D. Ernst, and W. G. Griswold. Automated support for program refactoring using invariants. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pages 736–, Washington, DC, USA, 2001. IEEE Computer Society.
90. D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 351–360, New York, NY, USA, 2011. ACM.
91. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.
92. D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *IEEE/ACM International Conference on Software Engineering*, 2013.
93. M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of refactorings during software evolution. In *ICSE' 11: Proceedings of the 2011 ACM and IEEE 33rd International Conference on Software Engineering*, 2011.
94. M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *FSE '10: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 371–372, New York, NY, USA, 2010. ACM.
95. M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 309–319, Washington, DC, USA, 2009. IEEE Computer Society.
96. M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE '07: Proceedings of the 29th Inter-*

national Conference on Software Engineering, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society.

97. M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 187–196, New York, NY, USA, 2005. ACM.
98. M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 50:1–50:11, New York, NY, USA, 2012. ACM.
99. M. Kim, T. Zimmermann, and N. Nagappan. An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Trans. Softw. Eng.*, 40(7):633–649, July 2014.
100. S. Kim, K. Pan, and J. E. James Whitehead. When functions change their names: Automatic detection of origin relationships. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 143–152, Washington, DC, USA, 2005. IEEE Computer Society.
101. S. Kim, K. Pan, and E. E. J. Whitehead, Jr. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 35–45, New York, NY, USA, 2006. ACM.
102. R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. Refactoring a legacy component for reuse in a software product line: a case study: Practice articles. *J. Softw. Maint. Evol.*, 18:109–132, March 2006.
103. R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 155–169, New York, NY, USA, 2000. ACM Press.
104. R. Komondoor and S. Horwitz. Effective, automatic procedure extraction. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 33, Washington, DC, USA, 2003. IEEE Computer Society.
105. G. G. Koni-N'Sapu. A scenario based approach for refactoring duplicated code in object-oriented systems. Master's thesis, University of Bern, June 2001.
106. G. P. Krishnan and N. Tsantalis. Refactoring clones: An optimization problem. *ICSM*, 0:360–363, 2013.
107. D. A. Ladd and J. C. Ramming. A*: A language for implementing language processors. *IEEE Transactions on Software Engineering*, 21(11):894–901, 1995.
108. R. Lammel, J. Saraiva, and J. Visser, editors. *Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers*, volume 7680 of *Lecture Notes in Computer Science*. Springer, 2013.
109. J. Landauer and M. Hirakawa. Visual awk: a model for text processing by demonstration. In *Proceedings of the 11th International IEEE Symposium on Visual Languages*, VL '95, pages 267–, Washington, DC, USA, 1995. IEEE Computer Society.
110. J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. In *ICSM 1992: Proceedings of International Conference on Software Maintenance*, 1992.

111. T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. *Learning repetitive text-editing procedures with SMARTedit*, pages 209–226. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.
112. C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, pages 3–13, 2012.
113. M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *J. Syst. Softw.*, 1:213–221, Sept. 1984.
114. Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: a tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.
115. Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, March 2006.
116. Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 25–33, New York, NY, USA, 2006. ACM.
117. D. Lo, L. Jiang, A. Budi, et al. Comprehensive evaluation of association measures for fault localization. In *ICSM*, pages 1–10. IEEE, 2010.
118. A. MacCormack, J. Rusnak, and C. Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015–1030, 2006.
119. N. H. Madhavji, F. J. C. Ramil, and D. E. Perry. *Software evolution and feedback: theory and practice*. Hoboken, NJ: John Wiley & Sons, 2006.
120. G. Malpohl, J. J. Hunt, and W. F. Tichy. Renaming detection. *Automated Software Engineering*, 10(2):183–202, 2000.
121. R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 350–359, Washington, DC, USA, 2004. IEEE Computer Society.
122. T. McDonnell, B. Ray, and M. Kim. An empirical study of api stability and adoption in the android ecosystem. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 70–79, 2013.
123. N. Meng, L. Hua, M. Kim, and K. S. McKinley. Does automated refactoring obviate systematic editing? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 392–402, Piscataway, NJ, USA, 2015. IEEE Press.
124. N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 329–342, New York, NY, USA, 2011. ACM.
125. N. Meng, M. Kim, and K. S. McKinley. Lase: locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 502–511, Piscataway, NJ, USA, 2013. IEEE Press.
126. T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.
127. T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, Feb. 2004.

128. T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.
129. R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 161–174, Berkeley, CA, USA, 2001. USENIX Association.
130. N. Moha, Y.-G. Guhneuc, A.-F. L. Meur, and L. Duchien. A domain analysis to specify design defects and generate detection algorithms. In J. L. Fiadeiro and P. Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2008.
131. R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi. Does refactoring improve reusability? In *ICSR*, pages 287–297, 2006.
132. M. Mossienko. Automated Cobol to Java recycling. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, 2003.
133. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997.
134. G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the eclipse IDE? volume 23, pages 76–83, Los Alamitos, CA, USA, July 2006. IEEE Computer Society Press.
135. E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, Jan 2012.
136. N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 284–292. ACM, 2005.
137. L. Naish, H. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM TOSEM*, 20(3):11, 2011.
138. A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Statistical learning approach for mining api usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 457–468. ACM, 2014.
139. A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.
140. H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen. A graph-based approach to api usage adaptation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 302–321, New York, NY, USA, 2010. ACM.
141. T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen. Exploring api embedding for api usages and applications. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 438–449, Piscataway, NJ, USA, 2017. IEEE Press.
142. T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 383–392, New York, NY, USA, 2009. ACM.
143. R. Nix. Editing by example. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 186–195, New York, NY, USA, 1984. ACM.

144. D. Ohst, M. Welle, and U. Kelter. Difference tools for analysis and design documents. In *ICSM '03*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.
145. W. F. Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.
146. A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth International Symposium on Foundations of Software Engineering*, pages 241–251, New York, NY, USA, 2004. ACM.
147. J. L. Overbey, M. J. Fotzler, A. J. Kasza, and R. E. Johnson. A collection of refactoring specifications for fortran 95. In *ACM SIGPLAN Fortran Forum*, volume 29, pages 11–25. ACM, 2010.
148. Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 247–260, New York, NY, USA, 2008. ACM.
149. Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in linux device drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 59–71, New York, NY, USA, 2006. ACM.
150. D. E. Perry, H. P. Siy, and L. G. Votta. Parallel changes in large-scale software development: an observational case study. *ACM Trans. Softw. Eng. Methodol.*, 10(3):308–337, 2001.
151. K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1 –10. IEEE Press, September 2010.
152. R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005.
153. N. Rachatasumrit and M. Kim. An empirical investigation into the impact of refactoring on regression testing. In *ICSM '12: the 28th IEEE International Conference on Software Maintenance*, page 10. IEEE Society, 2012.
154. J. Ratzinger, M. Fischer, and H. Gall. Improving evolvability through refactoring. In *Proc. 2nd*, pages 1–5, May 2005.
155. J. Ratzinger, T. Sigmund, and H. C. Gall. On the relation of refactorings and software defect prediction. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 35–38, New York, NY, USA, 2008. ACM.
156. B. Ray and M. Kim. A case study of cross-system porting in forked projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 53:1–53:11, New York, NY, USA, 2012. ACM.
157. B. Ray, M. Kim, S. Person, and N. Rungta. Detecting and characterizing semantic inconsistencies in ported code. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 367–377, Nov 2013.
158. X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 432–448, New York, NY, USA, 2004. ACM.

159. P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212. ACM, 2013.
160. P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: a case study of the apache server. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 541–550, New York, NY, USA, 2008. ACM.
161. R. Robbes and M. Lanza. Spyware: a change-aware development toolset. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 847–850, New York, NY, USA, 2008. ACM.
162. R. Robbes, M. Lungu, and D. Röthlisberger. How do developers react to api deprecation?: the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 56:1–56:11, New York, NY, USA, 2012. ACM.
163. M. P. Robillard and G. C. Murphy. Feat: a tool for locating, describing, and analyzing concerns in source code. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 822–823, Washington, DC, USA, 2003. IEEE Computer Society.
164. R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 404–415, Piscataway, NJ, USA, 2017. IEEE Press.
165. G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, 1997.
166. M. Schaefer and O. de Moor. Specifying and implementing refactorings. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 286–301, New York, NY, USA, 2010. ACM.
167. M. Schmidt and T. Gloetzner. Constructing difference tools for models using the sidiff framework. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 947–948, New York, NY, USA, 2008. ACM.
168. C. B. Seaman. Software maintenance: Concepts and practice. *J. Softw. Maint. Evol.*, 20(6):463–466, Nov. 2008.
169. D. Shao, S. Khurshid, and D. Perry. Evaluation of semantic interference detection in parallel changes: an exploratory experiment. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 74–83, Oct. 2007.
170. D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, pages 212–224, New York, NY, USA, 2007. ACM.
171. S. Sidiroglou, S. Ioannidis, and A. D. Keromytis. Band-aid patching. In *Proceedings of the 3rd Workshop on on Hot Topics in System Dependability*, HotDep'07, Berkeley, CA, USA, 2007. USENIX Association.
172. D. Silva, N. Tsantalis, and M. T. Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 858–870, New York, NY, USA, 2016. ACM.
173. J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

174. H. M. Sneed. Migrating from COBOL to Java. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, 2010.
175. G. Soares. Making program refactoring safer. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 521–522, 2010.
176. S. Son, K. S. McKinley, and V. Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *NDSS*, 2013.
177. M. Soto and J. Münch. Process model difference analysis for supporting process evolution. *Lecture Notes in Computer Science, Springer Berlin*, Volume 4257/2006:123–134, 2006.
178. K. Sullivan, P. Chalasani, and V. Sazawal. Software design as an investment activity: A real options perspective. Technical report, 1998.
179. E. B. Swanson. The dimensions of maintenance. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 492–497, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.
180. L. Tahvildari and K. Kontogiannis. A metric-based approach to enhance design quality through meta-pattern transformations. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, CSMR '03, pages 183–, Washington, DC, USA, 2003. IEEE Computer Society.
181. R. Tairas and J. Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Information and Software Technology*, 54(12):1297–1307, 2012.
182. Y. Tao and S. Kim. Partitioning composite code changes to facilitate code review. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 180–190. IEEE, 2015.
183. P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
184. W. F. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, 1984.
185. M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 173–180, Washington, DC, USA, 2004. IEEE Computer Society.
186. C. Treude, S. Berlik, S. Wenzel, and U. Kelter. Difference computation of large models. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 295–304, New York, NY, USA, 2007. ACM.
187. N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *CSMR '08: Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, pages 329–331, Washington, DC, USA, 2008. IEEE Computer Society.
188. N. Tsantalis and A. Chatzigeorgiou. Identification of extract method refactoring opportunities. In *CSMR '09: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, pages 119–128, Washington, DC, USA, 2009. IEEE Computer Society.
189. N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Trans. Softw. Eng.*, 35(3):347–367, 2009.

190. N. Tsantalis and A. Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782, 2011.
191. N. Tsantalis and A. Chatzigeorgiou. Ranking refactoring suggestions based on historical volatility. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 25–34, March 2011.
192. M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 233–243, June 2012.
193. R. van Engelen. On the use of clone detection for identifying crosscutting concern code. *IEEE Transactions on Software Engineering*, 31(10):804–818, 2005. Student Member-Magiel Bruntink and Member-Arie van Deursen and Member-Tom Tourwe.
194. E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. *Domain-Specific Program Generation*, 3016:216–238, 2004.
195. W. Wang and M. W. Godfrey. Recommending clones for refactoring using design, context, and history. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 331–340, Sept 2014.
196. Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 61–72, New York, NY, USA, 2010. ACM.
197. W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.
198. P. Weißgerber and S. Diehl. Are refactorings less error-prone than other changes? In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 112–118, New York, NY, USA, 2006. ACM.
199. P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.
200. Wikipedia. Comparison of bsd operating systems — Wikipedia, the free encyclopedia, 2012.
201. S. Wong, Y. Cai, M. Kim, and M. Dalton. Detecting software modularity violations. In *ICSE' 11: Proceedings of the 2011 ACM and IEEE 33rd International Conference on Software Engineering*, 2011.
202. Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65, New York, NY, USA, 2005. ACM.
203. Z. Xing and E. Stroulia. Refactoring detection based on umldiff change-facts queries. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 263–274, Washington, DC, USA, 2006. IEEE Computer Society.
204. Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported - an eclipse case study. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 458–468, Washington, DC, USA, 2006. IEEE Computer Society.

205. Z. Xing and E. Stroulia. Api-evolution support with diff-catchup. *IEEE Trans. Softw. Eng.*, 33(12):818–836, 2007.
206. T. Yamamoto, M. Matsushita, T. Kamiya, and K. Inoue. Measuring similarity of large software systems based on source code correspondence. In *Proceedings of 2005 Product Focused Software Process Improvement*, pages 530–544, 2005.
207. W. Yang. Identifying syntactic differences between two programs. *Software – Practice & Experience*, 21(7):739–755, 1991.
208. W. Yang, S. Horwitz, and T. Reps. Detecting program components with equivalent behaviors. Technical Report CS-TR-1989-840, University of Wisconsin, Madison, 1989.
209. K. Yasumatsu and N. Doi. SPiCE: a system for translating Smalltalk programs into a C environment. *IEEE Transactions on Software Engineering*, 1995.
210. Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 26–36, New York, NY, USA, 2011. ACM.
211. R. Yokomori, H. P. Siy, M. Noro, and K. Inoue. Assessing the impact of framework changes using component ranking. In *ICSM*, pages 189–198. IEEE, 2009.
212. A. Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE-7: Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–267, London, UK, 1999. Springer-Verlag.
213. A. Zeller. Automated debugging: Are we close? *IEEE Computer*, 34(11):26–31, 2001.
214. L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *Proc. of ICSM*, pages 23–32. IEEE, 2011.
215. T. Zhang, M. Song, J. Pinedo, and M. Kim. Interactive code review for systematic changes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 111–122. IEEE Press, 2015.
216. H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 195–204. ACM, 2010.
217. L. Zou and M. W. Godfrey. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.