A Systematic Review of the Literature of the Techniques to Perform Transformations in Software Engineering

Uma revisão sistemática da literatura das técnicas para realizar transformações na engenharia de software

DOI: 10.34117/bjdv6n7-361

Recebimento dos originais: 15/06/2020
Aceitação para publicação: 15/07/2020

Reudismam Rolim de Sousa
Doutor em Ciência da Computação pela Universidade Federal de Campina Grande (UFCG)
Instituição: Universidade Federal Rural do Semi-Árido
Endereço: Departamento de Engenharias e Tecnologia. Universidade Federal Rural do Semi-Árido (UFERSA), Rodovia BR-226, KM 405, s/n - São Geraldo, Pau dos Ferros – RN, Brasil, 59900-000.
E-mail: reudismam.sousa@ufersa.edu.br

ABSTRACT
Along with software evolution, developers may do repetitive edits. These edits can be identical or similar to different codebase locations, which may occur as developers add features, refactor, or fix a bug. Since some of these edits are not present in Integrated Development Environments (IDEs), they are often performed manually, which is time-consuming and error-prone. In order to help developers to apply repetitive edits, some techniques were proposed. In this work, we present a systematic review of the literature of the techniques to do transformations in software engineering. As a result, this systematic review returned 51 works ranging from the domains programming-by-examples, linked editing, API usage, bug fixing, complex refactoring, and complex transformations, which can be used to help tools' designer in the proposition of new approaches.

keywords: Systematic Review of the Literature, Transformations, Software Engineering.

RESUMO
Junto com a evolução do software, os desenvolvedores podem fazer edições repetitivas. Essas edições podem ser idênticas ou semelhantes a locais diferentes da base de código, o que pode ocorrer quando os desenvolvedores adicionam recursos, refatoram ou corrigem um bug. Como algumas dessas edições não estão presentes nos Ambientes de Desenvolvimento Integrado (IDEs), elas geralmente são executadas manualmente, o que consome tempo e é propenso a erros. Para ajudar os desenvolvedores a aplicar edições repetitivas, algumas técnicas foram propostas. Neste trabalho, apresentamos uma revisão sistemática da literatura das técnicas para realizar transformações na engenharia de software. Como resultado, esta revisão sistemática retornou 51 trabalhos, desde os domínios programação por exemplos, edição vinculada, uso de API, correção de erros, refatoração complexa e transformações complexas, que podem ser usadas para ajudar o designer de ferramentas na proposição de novas abordagens.

Palavras-chave: Revisão sistemática da literatura, transformações, engenharia de software.
1 INTRODUÇÃO

During software evolution, developers may do repetitive edits [Negara et al. 2014, Nguyen et al. 2013a, Ray et al. 2015, Kim and Meng 2014, Toomim et al. 2004]. These edits are identical or similar to different codebase locations, which may occur as developers add features, refactor, or fix a bug. An example of them is an API replacement where developers must find old API usages and replace it by the new one. Since some repetitive edits are recurring, IDEs automate, in the refactoring suites, common transformations (e.g., rename refactoring). A transformation generalizes developer concrete edits in an abstraction to be applied to other codebase locations.

However, developers still do many transformations absent in IDEs. These edits are often done manually, which is time-consuming and error-prone. To help developers to apply repetitive edits, some techniques were proposed. Nguyen et al. (2013b) present a recommender to automate repetitive edits based on properties of these edits. Meng et al. (2013) propose a technique that uses two or more examples of edited methods to learn a transformation that locates and edits target methods of the repetitive edit. Raychev et al. (2013) propose an approach that uses refactorings from IDEs to compose complex refactorings.

Researchers also reviewed techniques for repetitive edits. Monperrus (2015) writes a bibliography on automatic software repair. Zibran and Roy (2012) survey clone management. Park et al. (2012) present an empirical study on supplementary bug fixing. Mens and Tourwé (2004) and Pakdeetrakulwong et al. (2014) present techniques to recommend source code edits. These works focus on specific transformations, and many of them are not conducted in a systematic way. Furthermore, Kim et al. (2014) study techniques to automatize repetitive edits, but this study is restricted to a few pre-selected papers.

In this work, we present a systematic review of techniques to do repetitive edits. These edits could occur horizontally (i.e., across commits/ revisions) or vertically (i.e., on a single commit/revision).

Our goal is to characterize techniques to transform software code on perspective of repetitive edits. Formally, we have:

**Goal:** analyze techniques to transform source code for the purpose of characterizing them with respect to technique properties from the point of view of software developers in the context of software evolution.
In this work, we present the state-of-the-art about program transformations. This state-of-the-art is collected using a systematic review, organized as following: In Section 2, we present the review methods, including our research strategy, search string, selection methodology, quality assessment, inclusion and exclusion criteria, extracted information, and execution. In Section 3, we present the results of our systematic review. Finally, in Section 4, we discuss the works in our state-of-the-art.

1.1 REVIEW QUESTIONS

We base our review on Kitchenham and Charters guidelines to do systematic reviews in software engineering. Our research question is the following:

**RQ 01:** what are the techniques to do transformations in software engineering?

We collect studies to directly answer our RQ (i.e., primary studies). In addition, we collect studies that review primary studies (i.e., secondary studies) such as systematic literature review, mapping study, and surveys.

2 REVIEW METHODS

We first conduct an exploratory study. This study aims to select studies to base our review. It helps to define places to search for studies (i.e., databases) and terms to build our search string (e.g., major terms and synonyms). Table 1 shows works on this exploratory study.

| Reference       | Name                                                                 |
|-----------------|-----------------------------------------------------------------------|
| Nguyen et al.   | A Study of Repetitiveness of Code Changes in Software Evolution       |
| Ray et al.      | The Uniqueness of Changes: Characteristics and Applications           |
| Meng et al.     | LASE: Locating and Applying Systematic Edits by Learning from Examples|
| Robbes and Lanza| Example-based Program Transformation                                  |
| Raychev et al.  | Refactoring with Synthesis                                            |
| Kim and Meng    | Recommending Program Transformations to Automate Repetitive Software Changes |
| Nguyen et al.   | A Graph-Based Approach to API Usage Adaptation                        |
| Bravenboer et al.| Stratego/XT 0.17: A language and toolset for program transformation |
| Cordy           | Source Transformation, Analysis and Generation in TXL                |
| Boshernitsan et al.| Aligning Development Tools with the Way Programmers Think about Code Changes |
| Toomim et al.   | Managing Duplicated Code with Linked Editing                         |

2.1 SEARCH STRATEGY

To select studies for our SLR, we apply the following methodology:
a) We derive from population, intervention, and outcome major terms for our search string.
b) We identify alternative spelling and synonym for major terms.
c) We identify databases to apply the search string.
d) We build a search string for each database.
e) We do the search.
f) We do backward snowballing (i.e., analyze works in reference section) and forward snowballing (i.e., analyze who cited this work). For backward snowballing, we analyze related works. Based on Kitchenham Brereton, for forward snowballing, we select critical works (10% of total) based on number of citations by year and collect works citing it.

Regard the item a), search terms derive from population, intervention, and outcome. Population is who/what intervention affects. Intervention is what is being studied, and outcome denotes observed results. Our population is software engineers (target of transformation techniques). Intervention is techniques to do transformation, and outcome is technique transformations.

To answer the item b), we use our exploratory study to identify synonyms and alternative spelling, which are shown in Table 2.

| Value               | Alternative Spelling and Synonyms                                      |
|---------------------|-------------------------------------------------------------------------|
| Software engineer   | Developer, programmer                                                   |
| Systematic change   | Repetitive edit, repetitive change, non-unique change, duplicate change, systematic edit, linked edit, refactoring |
| Technique           | Tool, recommender, algorithm, synthesizer                                |

For item c), we identify databases. Based on Kitchenham and Brereton 2013, we choose ACM and IEEE along with ScienceDirect, Engineering Village, and Scopus. Table 3 shows these databases.

| Reference        | Name                                      |
|------------------|-------------------------------------------|
| Language         | English                                   |
| Search method    | The work must be available online.        |
|                  | IEEE Explore                              |
|                  | ACM Digital Library                       |
| Databases        | ScienceDirect                             |
|                  | Engineering Village                       |
|                  | Scopus                                    |
| SR evaluation    | Author                                    |
| Publication year | After 2005 until 2016                     |
2.2 SEARCH STRING

Search strings may vary among databases. Default search string for our SLR is the following:

(“software engineering” OR developer OR programmer) AND (“systematic change” OR “program repair” OR “API usage adaptation” OR “Code change” OR “program change” OR “program transformation” OR “script transformation” OR “source transformation” OR “code transformation” OR “repetitive change” OR “non-unique change” OR “duplicated change” OR “duplicated code” OR “systematic edit” OR “linked edit” OR “refactoring” OR “bug-fix”) AND (technique OR tool OR recommender OR algorithm OR synthesizer)

For each database, we search based on title, abstract, and meta-data. If databases lack these characteristics, we search based on the full text (if allowed) or default (otherwise).

- **IEEE Explorer**: we use advanced search and do three searches based on title, abstract, and meta-data. This database restricts search string to 15 terms. Since we have more than 15 terms, we do different search and concatenate results.
- **ACM Digital Library**: we use advanced search and do three searches based on title, abstract, and meta-data.
- **ScienceDirect**: we use advanced search. We select “Computer Science” for area and do the search.
- **Engineering Village**: we use advanced search. We select “Computer Science” for area and do the search.
- **Scopus**: we access search area and do the search.

2.3 SELECTION METHODOLOGY

We use the StArt [LaPES 2017], a tool for conducting systematic review. We divided our methodology in selection and extraction stages as follows. For each database, we searched based on database syntax (see Section 2.2). On selection stage, we analyze title and abstract. If the work fits an exclusion criterion, we reject; otherwise, we accept it. On extraction stage, we scan the work. If work fits an exclusion criterion, we reject; otherwise, we analyze full text and classify it. Finally, we write the report based on accepted PSs.
2.4 QUALITY ASSESSMENT

We base on Kitchenham and Charters (2007) quality assessment. We remove works focused on automatic program repair since extensive study focuses on it [Monperrus 2015] that by itself can be a systematic review. We also discard works on clone removal. In addition, we remove works on transformations for a too specific problem (stylized edits)[Meng et al. 2013]. For instance, works in concurrent transformations [Brown et al. 2013, Wloka et al. 2009, Schäfer et al. 2011], removal of security issues [Joiner et al. 2014, Ma et al. 2016], refactoring of web applications [Nguyen et al. 2013c], refactoring to aspects [Vidal and Marcos 2013, Vidal and Marcos 2012, Marin et al. 2009], refactoring C macros [Medeiros et al. 2014, Overbey et al. 2014], refactoring product lines [Schulze et al. 2013], among others. Table 4 shows our quality criteria.

| Id | Criterion | Score response |
|----|-----------|----------------|
| 01 | Clear, unambiguous | Yes = 1, No = 0 |
| 02 | Full work (≥8 pages) | Yes = 1, No = 0 |
| 03 | Besides Fowler catalog | Yes = 1, No = 0 |
| 04 | Automate transformations | Yes = 1, No = 0 |
| 05 | Many transformations | Yes = 1, No = 0 |

2.5 INCLUSION AND EXCLUSION CRITERIA

We are interested in works related to software engineering, answering our RQ. The inclusion and exclusion criteria for our systematic review can be found on Table 5. In particular, we are interested in systematic change studies. Works that apply identical or similar transformation throughout the source code.

| Reference | Name |
|-----------|------|
| Inclusion criteria | a) The work answers our research question |
| | b) The study has a high quality |
| | a) The work does not deal with systematic changes |
| | b) The work is external to software engineering |
| Exclusion criteria | c) The document does not make part of a conference or journal |
| | d) The document is unavailable online |
| Study type | Cases studies, empirical studies, and second studies |
| Duplicate studies | Select the most recent version |
2.6 INFORMATION EXTRACTION

From PSs, besides the standard data (e.g., title, author, and proceeding) shown in Table 6, we collect data to answer our RQ. For secondary studies, we collect PSs that contribute to answer our RQ.

To answer our RQ, we collect:

- **Locations**: we collect location strategy (i.e., manual, semi-automatic, or automatic).
- **Transformation**: we collect edit strategy (i.e., manual, semi-automatic, or automatic).
- **Location template**: we collect template learning for location (i.e., manual, semi-automatic, or automatic).
- **Transformation template**: we collect template for transformation (i.e., manual, semi-automatic, or automatic).
- **Transformation scope**: we collect transformation scope (e.g., method, API, and refactoring).
- **Manipulation**: we collect technique manipulation (i.e., text, syntax, semantic).
- **Language**: we collect target language (e.g., Java, C#, C).
- **Input**: we collect input (e.g., input-output examples, history, before-after API).
- **Output**: we collect output (e.g., modification list, modified source code).
- **Learning**: we collect learning (i.e., on-line, off-line).
- **Reuse**: we collect whether transformation could be reused.
- **Catalog-bounded**: we collect whether transformation catalogs bound techniques.
- **Abstraction**: we collect elements abstracted.
- **Interaction**: we collect developers’ interaction model.
- **Technique scope**: we collect technique scope (e.g., method body, attribute, or class).
- **Context**: we collect whether technique models context.
- **Similarity**: we collect whether technique does similar or identical transformations.
- **Multiple or single**: we collect whether technique works on single or multiple files.

For example-based techniques, we collect:

- **Number of examples**: we collect number of examples.
- **Example selection methodology**: we collect example selection methodology.
If any feature is absent, we set NP value. If feature could not be identified, we set a NA value.

| Item     | Description                      |
|----------|----------------------------------|
| Item type| Study type (e.g., journal, conference) |
| Title    | Study title                      |
| Author(s)| Study author(s)                  |
| Publication | Study series                    |
| Volume   | Study volume                     |
| Issue    | Study issue                      |
| Pages    | Study pages                      |
| Year     | Study year                       |

2.7 EXECUTION

In the selection stage, we selected 4,741 works. From these works, 2,849 were rejected, 1,319 were duplicated, and 573 were accepted. From the 573 works for the extraction stage, 362 were accepted, 173 were rejected, and 40 were duplicated. Due to the large number of accepted works (573), we apply further filtering. First, we reject works based on publication year. We only accepted works published after 2005. Second, we reject works based on quality criteria. We only accept works with maximum quality score. Finally, we end up with 51 PSs.

3 RESULTS

We select 51 works on total. Figure 1 summarizes PSs based on events.

![Number of PS for each event](image)
4 DISCUSSION

In this section, we present the primary studies selected to be included in the list of works for our systematic review. Techniques to automate transformations range from PBE (Section 4.1), linked editing (Section 4.2), API usage (Section 4.3), bug fixing (Section 4.4), complex refactoring (Section 4.5), and complex transformations (Section 4.6).

4.1 PROGRAMMING BY EXAMPLES

PBE techniques use developers' edits or modification from repositories to learn transformations to edit other code locations.

Automatic transformation techniques Lase [Meng et al. 2013] is a technique for doing repetitive edits using examples. Developers give two or more edited methods as examples, and Lase creates a context-aware abstract transformation that could be applied to other locations in codebase. It uses clone detection and dependence analysis to identify methods where transformation should be applied. Andersen and Lawall (2008) present spdiff, a technique that extracts common edits from two versions of source files. Then, it learns a transformation that could be applied to other codebase locations. The key idea of Meng et al. (2015) is to remove code clones from the source code based on examples. Their technique (Rase) combines refactorings (e.g., extract method, add parameter, introduce exit label) to extract and remove similar code. It extracts common code from clones and creates a new method that abstract clones, removing them from code. Kessentini et al. (2011) propose a technique to remove design defect from the source code. Their technique receives defect examples and quality metrics to learn rules to refactor source code.

Edit-based techniques Some techniques require that developers indicate locations where transformations could be applied. Sydit [Meng et al. 2011] receives one or more edited methods as examples and learned a transformation to apply to locations indicated by developers. Santos et al. (2015) propose MacroRecorder, a technique that records developers code edit events from IDE to reproduce them in other code locations. MacroRecorder semi-automatically learns a transformation and allows the developers to customize the transformation.

Semi-automatic techniques Some techniques help developers to write their own transformations. Boshernitsan et al. (2007) present iXj, a technique that semi-automatically allows developers to specify transformations. Based on mental models of developer interaction, they developed a visual
tool that allows developers to specify transformations. Rosemari (2008) is a semi-automatic technique that migrates programs for annotated version of a framework based on examples. It infers refactoring rules from two versions of a single class provided by developers. Developers configure transformations inferred by Rosemari, which refactor the source code automatically.

**Refactoring** Other techniques use developers’ edits on IDE to recommend refactorings. BeneFactor [Ge et al. 2012] is a technique that detects whether developers are applying a refactoring and can finish the refactoring to developers. BeneFactor uses developers’ edits as example and automatically detects whether developers are applying a refactoring. If this occurs, BeneFactor identify the refactoring and can finish the refactoring.

4.2 LINKED EDITING

Managing clones is expensive. The main bottleneck relies in update code clones. Linked editing techniques offer strategies to update clones. JSync [Nguyen et al. 2012] and Clever [Nguyen et al. 2009] are techniques for clone management. These techniques detect code clones, detect edits in clones, and notify developers for clone inconsistencies. CloneTrack [Duala-Ekoko and Robillard 2010] is a technique that uses clone region descriptors (CRDs) to management of code clones. CRDs combine syntactic, structural, and lexical information to represent code clones in codebase. CloneTrack uses output of a clone detection technique and produces CRDs to represent clone regions. Developers can specify clones to track, and the technique notifies developers for modification and supports update in these clones. The key idea of Wit et al. (2009) is to keep track copy and paste operations as clone relations. Their technique (CloneBoard) detects when developers edit a clone relation and shows clones in the relation. It semi-automatically supports update of clones in this relation.

4.3 API USAGE

Source code often uses external API providers. These providers should maintain external APIs unchanged, but sometimes it does not happen [Xing and Stroulia 2007]. Thus, clients need to update usage to conform new API. In addition, developers often need code completion systems to guide them to use an API. IDEs provide code completion systems, but IDEs’ recommendations often rank suggestions based on alphabetic order, which does not fit API context.
**DSL-based API adaptation** To help clients to adapt API, providers may give tools to automate it (e.g., Microsoft wizard, which helps in migrating Visual Basic to Visual Basic.Net [Li et al. 2015]). However, providing a transformation tool is difficult, and developers often migrate API manually. To help developers, some studies use transformation DSL. Nita and Notkin (2010) present twinning, a technique to update API by mapping usage between two codebase versions. Developers write rules describing correspondences usage in these versions to update code. Patl [Wang et al. 2016] is a DSL that allows developers to write rules for a transformation where many invocations from old API are replaced by many invocations to new API (many-to-many mapping). The key idea for Li et al. (2015) is a type-safe transformation DSL for API migration. Their technique (Swin) avoids type bugs in transformed program. This language updates code where one method invocation in old API usage is replaced by one or multiple invocations in the new API usage (one-to-many mapping). Wu et al. (2015) present CMIT, a transformation language for API adaptation. CMIT allows transformation where a sequence of method invocation is considered together during transformation (compositional-mapping).

**Automatic and semi-automatic API adaptation** To reduce the burden of writing transformations since developers have to learn how to use a DSL, some techniques analyze code usage from new and old API to update API. Nguyen et al. (2010a) present LibSync, a technique that recommends locations and edits for API adaptation. LibSync uses clients that migrated the API to learn transformations. LibSync compares two library versions to identify edits, extract templates for the usage, and build patterns of API usage. Based on these patterns, it recommends API adaptation. The key idea of Dagenais and Robillard (2008) is to suggest updates based on adaptations of the own framework. Their technique (SemDiff) analyzes framework source code, identifies edits (e.g., method additions and deletions) and, based on these edits and methods removed from framework, recommends replacements. Similarly, Xing and Stroulia (2007) present DiffCatchUp, a technique that recognizes API changes and support client update based on examples from framework source code. The key idea of Cossette et al. (2014) is to help developers to migrate API when framework updates are undocumented or incomplete. In this case, developers must discover how to use new API. They propose Umani, a technique that discovers replacements for breaking APIs based on structural similarity from new API and old API. Other techniques are semi-automatic. Kapur et al. (2010) propose Trident, a technique for API migration. After selection a code fragment, developers configure refactorings and Trident shows all locations following the same pattern. Then, developers select locations that need to be edit and configure transformation to be applied.
**Code completion based on statistical models** Another way to help developers in API usage is using code-completion. Some techniques use high repetitiveness of code edits [Gabel and Su 2010, Hindle et al. 2012] to build statistical models for code completion. Nguyen et al. (2016) present APIRec, a technique that recommends API usage based on repetitiveness of code edits. APIRec is trained in a corpus of repetitive edits, learn a statistical model based on the co-occurrence of fine-grained changes, and recommends API calls based on edit context. GraLan and ASTLan [Nguyen and Nguyen 2015] are graph-based and AST-based techniques that recommend the next element (e.g., API method, field access, and related control units) based on statistical models of code usage. These techniques are based on n-gram, which is a statistical model that assumes a sequence from right to left, and next elements depend only on current context. They consider source code properties that are difficult model in n-gram (i.e., API usage often lacks order, repetitive code often interleaves with project specific code, and elements could be apart from one another).

**Mining repository for code completion** Other works mines repositories to recommend code completion. Nguyen et al. (2013b) mined repositories to identify characteristics of repetitive edits, focusing in three dimensions: size, type, and general/fixing edits. They build a recommender system based on these characteristics. Ray et al. (2015) studied edits characteristics of unique/repetitive edit from repository (e.g., extent of unique edits, who introduce unique edits, and where the edits take place). They propose two recommender systems. The first recommends previous edits when developers select a fragment of code to edit. The second recommends edits that co-occurred when developers do a repetitive edit.

**Code completion based on standard machine learning algorithm** Other techniques uses standard machine techniques in code completion. Bruch et al. (2009) present three code completion systems based on repositories using K-nearest neighbor algorithm. These systems search for code fragments that use variable of the same type of variable developers want a recommendation. Each code completion systems recommend based on specific metric: frequency of method calls, rule mining, and the closest code fragment.

**4.4 BUG FIXING**

Bugs are frequent along software development. Although developers try to remove bugs for source code, many bugs persist in systems bug reports. Human resources are often insufficient to
remove all bugs, ever for well know bugs (e.g., Windows 2000 was shipped with thousands of well knows bugs) [Kim et al. 2013]. Bug fixing techniques help developers to remove bugs from code.

**Recurring bug fixing** Some bugs re-occur [Kim et al. 2006]. Thus, developers may fix these bugs similarly. To help developers, some techniques were proposed. Nguyen et al. (2010b) present FixWizard, a semi-automatic technique to fix recurring bugs based on properties of recurring fixes (e.g., why, where, and how fixes occur and how to detect fixes). FixWizard identifies target locations, detects recurring fixes, and recommends fixing edits. BugFix [Jeffrey et al. 2009] is a technique that uses association rules to help developers to fix a bug. This tool suggests fixes by analyzing historic of bug fixing scenarios. Association rules map debugging scenarios with respective fixing scenarios. BugFix uses these rules to suggest a ranked list of fixes. The key idea of Kim et al. (2013) is to use human-written patches to repair programs. They analyze 62,652 human-written and identified ten common fixing templates. Their technique (Par), detects bug locations using a fault location technique [Le Goues et al. 2012] and uses these fixing templates to repair program. Liu et al. (2013) propose R2Fix, a technique for automatic program repair that learns patches from bug reports. R2Fix analysis bug reports, identify common bugs, and recommends program repairs. This technique contains a bug classifier, a parameter extractor that extracts pattern based on bug reports, and a patch generator. Long and Rinard (2016) present Prophet, a technique for automatic program repair that learns a probabilistic application-independent model of correct code from a set of successful human patches.

**Semi-automatic techniques for bug fixing** Some techniques are semi-automatic. Xia and Lo (2016) present SupLocator, a technique that recommends target methods of supplementary bugs fixings. To identify target methods, this technique uses six edit relationships: method invocation, method containment, inheritance, co-change, context similarity, and name similarity. SupLocator builds a graph for each relationship and uses a genetic algorithm to recommend target methods of supplementary fixes. PatternBuild is a semiautomatic technique to help developers to resolve bugs that propagate over codebase. PatternBuild receives a bug version of project and a version with one or more fixed instances. PatternBuild allows developers to specify a bug pattern based on bug version and a fixing pattern based on fixed version. This technique searches for locations target of fixing based on the bug pattern. It shows these locations to developers that can resolve bug using the fixing pattern. Sun et al. (2008) present a semi-automatic technique that helps developers to fix a bug that propagates over codebase. This technique offers basic rules to fix bugs based on three
type of bug fixing usage: precondition, which requires checking input parameters before calling the function; post-condition, which requires checking return after calling the function; and call-pair, which requires that two functions be called together in a particular order. Developers can compose complex rules based on these rules. This technique detects rule violations and suggests fixing.

4.5 COMPLEX REFACTORING

IDEs provide refactorings, but developers often refactor manually. Besides knowing that refactoring automatically is faster than by hand, developers need to know that the refactoring exists, the refactoring name, and how to apply it [Foster et al. 2012]. In addition, IDEs refactoring suites are often limited. For complex refactorings, developers have to refactor manually or implement their own refactoring [Li and Thompson 2012]. Refactoring by hand is error-prone and time-consuming. Specifying a refactoring requires knowing to represent a program and available APIs.

**Search-based techniques** Some works search over a space of complex refactoring, selecting a refactoring that optimizes a function. Recon [Morales et al. 2016] is a technique that formulates refactorings as an optimization problem to find best refactoring sequence for source code. Recon uses developers' context and meta-heuristics to compute a refactoring sequence that affects elements in developer's context. Based on interaction traces, Recon generated an abstract model of source code. This technique detects anti-patterns and searches for refactoring sequences to fix these anti-patterns. O'Keeffe and Cinnéide (2008) present a technique that formulates refactorings as a search problem. Based on design quality function, this technique selects the best refactoring candidate over a space of design refactorings.

**Multi-objective refactoring** Traditional refactorings aim to improve code quality by changing internal structure, but maintaining external behavior or refactored program. Some studies investigate other criteria besides code quality. Ouni et al. (2016) present an approach that recommends a refactoring sequence based on multi-objective criteria. Over a space of refactoring sequences, this approach searches for a refactoring sequence that improves design quality, preserves design coherence and consistency of refactored program, minimizes code edits, and maximizes consistency with developers' history. Ouni et al. (2015) propose a multi-objective approach to recommend a refactoring sequence based on three criteria: minimize code-smells, maximize use of developers' history, and preserve construct semantics. The key idea of Ouni et al. (2013) is to use recorded edits from developers to recommend a refactoring sequence. They propose a multi-objective approach to
find the best refactoring sequence that maximizes use of past refactoring, minimizes semantic errors, and minimizes defects.

**Program language for complex refactoring** Some researchers allow developers to write their own refactoring using a DSL. Li and Thompson (2012) present a framework that allows developers to write their own complex refactorings. The framework is built on Wrangler refactoring tool. Developers write refactorings using Erlang syntax, making it suitable for developers familiar with Erlang. RefaCola [Steimann et al. 2011] is a refactoring DSL that allows writing refactoring with constraint. The key idea of Ruhroth et al. (2011) is to allow developers to write refactoring for any language that follows Backus-Naur-Form. They present ReL, a domain specific language for refactoring that allows developers to specify refactoring and execute refactorings in source code.

**Learning refactoring from developer edits** Some works use developers' edits to learn complex refactorings. Raychev et al. (2013) propose ReSynth, a technique that synthesizes a refactoring sequence. Developers active ReSynth, edit code base, and call ReSynth to complete refactoring. Given before/after version, ReSynth computes edits to synthesize a refactoring sequence. ReSynth requires developers to provide a successors function describing how to refactor a program. WitchDoctor (2012) is a technique that detects a refactoring while developers edit code and recommends a refactoring before developers complete refactoring. They key idea of Hayashi et al. (2006) is to learn a refactoring sequence based on developers' edits on repositories. This technique identifies locations and applies refactoring automatically.

**Complex refactoring** Some techniques analyze source code to identify complex refactoring. Alkhalid et al. (2010) propose a technique that clusters code at function/method level to suggest refactoring. This technique reprocesses code using an Adaptive K-nearest Neighbor to cluster source code. This technique shows clusters for developers that could apply extract method to refactoring code. R3 [Kim et al. 2016] is a technique that encodes Java entities in a database and maps refactorings in database to the source code. R3 compose complex refactoring as a sequence of primitive refactorings. Refactorings are applied to database. Source code is edited when developers modify pretty-printing ASTs mapped on database.
4.6 COMPLEX TRANSFORMATIONS

Besides refactoring, developers may apply other transformations (e.g., add a feature, or fix a bug). Some techniques allow developers to apply complex transformations. Cordy (2006) present TXL, a DSL for source transformations. TXL requires specification of DSL structure and transformation rules. Based on these rules, TXL applies transformations and outputs transformed versions. Stratego/XT [Bravenboer et al. 2008] is an infrastructure for program transformations. Stratego/XT uses Stratego, a DSL for program transformations and XT, an environment for development of transformations. Erwig and Ren [Erwig and Ren 2007] present a DSL for program transformation. They show language elements and an application of this DSL to update lambda calculus programs.

4.7 VALIDATION

In this section, we present the validation process of our systematic review. To validate our protocol, we conduct a pilot study following our protocol. We used this pilot study to review of the search string and databases. We have a set of works relevant to our study in a folder named Relevant Papers. Our study must select this set of works. If work in this folder is not returned, we added this work to our exploratory study, refine our search string and do a new search. We do this process until we select all works in relevant works folder.

5 CONCLUSION

Along with software evolution, the software needs to receive edits in order to fix bugs, add new features, or even change the internal structure of the software but maintaining its behavior. Some of these changes are performed frequently are currently integrated into integrated development environments. However, there are other changes that are not added into these environments, due to aspects such as the change being specific of a particular project. Therefore, many repetitive changes need to be performed manually, which can be tedious and error-prone. Due to this fact, many techniques are proposed to help developers in performing repetitive changes. In other to have a general view of these techniques, this work performs a systematic review of the literature of the techniques to perform transformations in software engineering. As a result, we identified 51 works ranging from the domains programming-by-examples, linked editing, API usage, bug fixing, complex refactoring, and complex transformations, which can designers to proposed new tools.
AGRADECIMENTOS

This research was supported by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) and by the Universidade Federal Rural do Semi-Árido - UFERSA and the Pró-Reitoria de Pesquisa e Pós-Graduação (PROPPG) through the PROPPG Edital N° 39/2019 to Support to Research Groups, granted to the project PIH00022-2019 of the Laboratório de Inovações em Software of the leader Reudismam Rolim de Sousa.

REFERÊNCIAS

[Alkhalid et al. 2010] Alkhalid, A., Alshayeb, M., and Mahmoud, S. (2010). Software refactoring at the function level using new adaptive k-nearest neighbor algorithm. *International Journal of Advances in Software Engineering*, 41(10-11):1160–1178.

[Andersen and Lawall 2008] Andersen, J. and Lawall, J. L. (2008). Generic patch inference. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE ’08, pages 337–346, Washington, DC, USA. IEEE Computer Society.

[Boshernitsan et al. 2007] Boshernitsan, M., Graham, S. L., and Hearst, M. A. (2007). Aligning development tools with the way programmers think about code changes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI ’07, pages 567–576, New York, NY, USA. ACM.

[Bravenboer et al. 2008] Bravenboer, M., Kalleberg, K. T., Vermaas, R., and Visser, E. (2008). Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70.

[Brown et al. 2013] Brown, C., Hammond, K., Danelutto, M., Kilpatrick, P., Schöner, H., and Breddin, T. (2013). Paraphrasing: Generating parallel programs using refactoring. In *Proceedings of the 10th International Symposium on Formal Methods for Components and Objects*, FMCO ’11, pages 237–256. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Bruch et al. 2009] Bruch, M., Monperrus, M., and Mezini, M. (2009). Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE ’09, pages 213–222, New York, NY, USA. ACM.

[Cordy 2006] Cordy, J. R. (2006). Source transformation, analysis and generation in TXL. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM ’06, pages 1–11, New York, NY, USA. ACM.

[Cossette et al. 2014] Cossette, B., Walker, R., and Cottrell, R. (2014). Using structural generalization to discover replacement functionality for API evolution.

[Dagenais and Robillard 2008] Dagenais, B. and Robillard, M. P. (2008). 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. ACM.

[Wit et al. 2009] de Wit, M., Zaidman, A., and van Deursen, A. (2009). Managing code clones using dynamic change tracking and resolution. In Proceedings of the 2009 IEEE International Conference on Software Maintenance, ICSM ’09, pages 169–178, Piscataway, NJ, USA. IEEE Press.

[Duala-Ekoko and Robillard 2010] Duala-Ekoko, E. and Robillard, M. P. (2010). Clone region descriptors: Representing and tracking duplication in source code. ACM Transactions on Software Engineering and Methodology, 20(1):3:1–3:31.

[Erwig and Ren 2007] Erwig, M. and Ren, D. (2007). An update calculus for expressing type-safe program updates. Science of Computer Programming, 67(2-3):199–222.

[Foster et al. 2012] Foster, S. R., Griswold, W. G., and Lerner, S. (2012). WitchDoctor: IDE support for real-time auto-completion of refactorings. In Proceedings of the 34th International Conference on Software Engineering, ICSE ’12, pages 222–232, Piscataway, NJ, USA. IEEE Press.

[Gabel and Su 2010] Gabel, M. and Su, Z. (2010). A study of the uniqueness of source code. In Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE ’10, pages 147–156, New York, NY, USA. ACM.

[Ge et al. 2012] Ge, X., DuBose, Q. L., and Murphy-Hill, E. (2012). Reconciling manual and automatic refactoring. In Proceedings of the 34th International Conference on Software Engineering, ICSE ’12, pages 211–221, Piscataway, NJ, USA. IEEE Press.

[Hayashi et al. 2006] Hayashi, S., Saeki, M., and Kurihara, M. (2006). Supporting refactoring activities using histories of program modification. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, 89-D(4):1403–1412.

[Hindle et al. 2012] Hindle, A., Barr, E. T., Su, Z., Gabel, M., and Devanbu, P. (2012). On the naturalness of software. In Proceedings of the 34th International Conference on Software Engineering, ICSE ’12, pages 837–847, Piscataway, NJ, USA. IEEE Press.

[Jeffrey et al. 2009] Jeffrey, D., Feng, M., Gupta, N., and Gupta, R. (2009). BugFix: A learning-based tool to assist developers in fixing bugs. In Proceedings of the 2009 IEEE 17th International Conference on Program Comprehension, ICPC ’09, pages 70–79, Piscataway, NJ, USA. IEEE Press.

[Joiner et al. 2014] Joiner, R., Reps, T., Jha, S., Dhawan, M., and Ganapathy, V. (2014). Efficient runtime-enforcement techniques for policy weaving. In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE ’14, pages 224–234, New York, NY, USA. ACM.

[Kapur et al. 2010] Kapur, P., Cossette, B., and Walker, R. J. (2010). Refactoring references for library migration. In Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA ’10, pages 726–738, New York, NY, USA. ACM.

[Kessentini et al. 2011] Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., and Ouni, A. (2011). Design defects detection and correction by example. In Proceedings of the 19th IEEE International Conference on Program Comprehension, ICPC ’11, pages 81–90, Piscataway, NJ,
USA. IEEE Press.

[Kim et al. 2013] Kim, D., Nam, J., Song, J., and Kim, S. (2013). Automatic patch generation learned from human-written patches. In Proceedings of the 35th International Conference on Software Engineering, ICSE ’13, pages 802–811, Piscataway, NJ, USA. IEEE Press.

[Kim et al. 2016] Kim, J., Batory, D., Dig, D., and Azanza, M. (2016). Improving refactoring speed by 10x. In Proceedings of the 38th International Conference on Software Engineering, ICSE ’16, pages 1145–1156, New York, NY, USA. ACM.

[Kim and Meng 2014] Kim, M. and Meng, N. (2014). Recommending Program Transformations to Automate Repetitive Software Changes, pages 1–31. Springer Berlin Heidelberg.

[Kim et al. 2006] Kim, S., Pan, K., and Whitehead, Jr., E. E. J. (2006). 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. ACM.

[Kitchenham and Brereton 2013] Kitchenham, B. and Brereton, P. (2013). A systematic review of systematic review processes research in software engineering. Information and Software Technology, 55(12):2049–2075.

[Kitchenham and Charters 2007] Kitchenham, B. and Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering. In Evidence-Based Software Engineering. Technical Report, EBSE, pages 1–65.

[LaPES 2017] LaPES (2017). Start: State of the art through systematic review. A http://lapes.dc.ufscar.br/tools/start_tool.

[Le Goues et al. 2012] Le Goues, C., Dewey-Vogt, M., Forrest, S., and Weimer, W. (2012). A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In Proceedings of the 34th International Conference on Software Engineering, ICSE ’12, pages 3–13, Piscataway, NJ, USA. IEEE Press.

[Li and Thompson 2012] Li, H. and Thompson, S. (2012). Let’s make refactoring tools user-extensible! In Proceedings of the Fifth Workshop on Refactoring Tools, WRT ’12, pages 32–39, New York, NY, USA. ACM.

[Li et al. 2015] Li, J., Wang, C., Xiong, Y., and Hu, Z. (2015). Swin: Towards type-safe Java program adaptation between apis. In Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM ’15, pages 91–102, New York, NY, USA. ACM.

[Liu et al. 2013] Liu, C., Yang, J., Tan, L., and Hafiz, M. (2013). R2fix: Automatically generating bug fixes from bug reports. In 6th IEEE International Conference on Software Testing, Verification and Validation, ICST ’13, pages 282–291, Piscataway, NJ, USA. IEEE Press.

[Long and Rinard 2016] Long, F. and Rinard, M. (2016). Automatic patch generation by learning correct code. Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 51(1):298–312.
[Ma et al. 2016] Ma, S., Lo, D., Li, T., and Deng, R. H. (2016). CDRep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xian, China, May 30 - June 3, 2016*, pages 711–722.

[Marin et al. 2009] Marin, M., Deursen, A., Moonen, L., and Rijst, R. (2009). An integrated crosscutting concern migration strategy and its semi-automated application to JHotDraw. *Automated Software Engineering*, 16(2):323–356.

[Medeiros et al. 2014] Medeiros, F., Ribeiro, M., Gheyi, R., and dos Santos Neto, B. F. (2014). A catalogue of refactorings to remove incomplete annotations. *The Journal of Universal Computer Science*, 20(5):746–771.

[Meng et al. 2015] Meng, N., Hua, L., Kim, M., and McKinley, K. S. (2015). 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. IEEE Press.

[Meng et al. 2011] Meng, N., Kim, M., and McKinley, K. S. (2011). 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. ACM.

[Meng et al. 2013] Meng, N., Kim, M., and McKinley, K. S. (2013). LASE: Locating and applying systematic edits by learning from examples. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE ’13, pages 502–511, Piscataway, NJ, USA. IEEE Press.

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

[Monperrus 2015] Monperrus, M. (2015). Automatic Software Repair: a Bibliography. Technical Report hal-01206501, University of Lille.

[Morales et al. 2016] Morales, R., Soh, Z., Khomh, F., Antoniol, G., and Chicano, F. (2016). On the use of developers’ context for automatic refactoring of software anti-patterns. *Journal of Systems and Software*, pages.

[Negara et al. 2014] Negara, S., Codoban, M., Dig, D., and Johnson, R. E. (2014). Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE’14, pages 803–813, New York, NY, USA. ACM.

[Nguyen et al. 2016] Nguyen, A. T., Hilton, M., Codoban, M., Nguyen, H. A., Mast, L., Rademacher, E., Nguyen, T. N., and Dig, D. (2016). API code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 24th ACM SIG- SOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 511–522, New York, NY, USA. ACM.

[Nguyen and Nguyen 2015] Nguyen, A. T. and Nguyen, T. N. (2015). Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE ’15, pages 858–868, Piscataway, NJ, USA. IEEE Press.
[Nguyen et al. 2013a] Nguyen, H. A., Nguyen, A. T., Nguyen, T. T., Nguyen, T. N., and Rajan, H. (2013a). A study of repetitiveness of code changes in software evolution. In *Proceedings of 28th the IEEE/ACM International Conference on Automated Software Engineering*, ASE ’13, pages 180–190, New York, NY, USA.

[Nguyen et al. 2013b] Nguyen, H. A., Nguyen, A. T., Nguyen, T. T., and Nguyen, T. N. (2013b). A study of repetitiveness of code changes in software evolution. In *Proceedings of the IEEE/ACM 28th International Conference on Automated Software Engineering*, ASE ’17, pages 180–190.

[Nguyen et al. 2013c] Nguyen, H. A., Nguyen, H. V., Nguyen, T. T., and Nguyen, T. N. (2013c). Output-oriented refactoring in php-based dynamic web applications. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, pages 150–159.

[Nguyen et al. 2012] Nguyen, H. A., Nguyen, T. T., Pham, N. H., Al-Kofahi, J., and Nguyen, T. N. (2012). Clone management for evolving software. *IEEE Transactions on Software Engineering*, 38(5):1008–1026.

[Nguyen et al. 2010a] Nguyen, H. A., Nguyen, T. T., Wilson, Jr., G., Nguyen, A. T., Kim, M., and Nguyen, T. N. (2010a). 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. ACM.

[Nguyen et al. 2010b] Nguyen, T. T., Nguyen, H. A., Pham, N. H., Al-Kofahi, J., and Nguyen, T. N. (2010b). Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd IEEE/ACM International Conference on Software Engineering - Volume 1*, ICSE ’10, pages 315–324, New York, NY, USA. ACM.

[Nguyen et al. 2009] Nguyen, T. T., Nguyen, H. A., Pham, N. H., Al-Kofahi, J. M., and Nguyen, T. N. (2009). Clone-aware configuration management. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE ’09, pages 123–134, Washington, DC, USA. IEEE Computer Society.

[Nita and Notkin 2010] Nita, M. and Notkin, D. (2010). Using twinning to adapt programs to alternative APIs. In *Proceedings of the 32Nd IEEE/ACM International Conference on Software Engineering - Volume 1*, ICSE ’10, pages 205–214, New York, NY, USA. ACM.

[O'Keeffe and í Cinnéide 2008] O’Keeffe, M. and í Cinnéide, M. (2008). Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502–516.

[Ouni et al. 2013] Ouni, A., Kessentini, M., and Sahraoui, H. (2013). Search-based refactoring using recorded code changes. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, CSMR ’13, pages 221–230, Washington, DC, USA. IEEE Computer Society.

[Ouni et al. 16] Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., and Deb, K. (16). Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology*, 25(3):23:1–23:53.
[Ouni et al. 2015] Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., and Hamdi, M. S. (2015). Improving multi-objective code-smells correction using development history. *J. Syst. Softw.*, 105(C):18–39.

[Overbey et al. 2014] Overbey, J. L., Behrang, F., and Hafiz, M. (2014). A foundation for refactoring C with macros. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE ’14, pages 75–85, New York, NY, USA. ACM.

[Pakdeetrakulwong et al. 2014] Pakdeetrakulwong, U., Wongthongtham, P., and Siricharoen, W. V. (2014). Recommendation systems for software engineering: A survey from software development life cycle phase perspective. In *The 9th International Conference for Internet Technology and Secured Transactions*, (ICITST ’14), pages 137–142.

[Park et al. 2012] Park, J., Kim, M., Ray, B., and Bae, D.-H. (2012). An empirical study of supplementary bug fixes. In Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, MSR ’12, pages 40–49, Piscataway, NJ, USA. IEEE Press.

[Ray et al. 2015] Ray, B., Nagappan, M., Bird, C., Nagappan, N., and Zimmermann, T. (2015). The uniqueness of changes: Characteristics and applications. In Proceedings of the 12th Working Conference on Mining Software Repositories, MSR ’15, pages 34–44, Piscataway, NJ, USA. IEEE Press.

[Raychev et al. 2013] Raychev, V., Schäfer, M., Sridharan, M., and Vechev, M. (2013). Refactoring with synthesis. In Proceedings of the 28th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA ’13, pages 339–354, New York, NY, USA. ACM.

[Robbes and Lanza 2008] Robbes, R. and Lanza, M. (2008). Example-based program transformation. In Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems, volume 5301 of MoDELS ’08, pages 174–188. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Ruhroth et al. 2011] Ruhroth, T., Wehrheim, H., and Ziegert, S. (2011). Rel: A generic refactoring language for specification and execution. In 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications, pages 83–90.

[Santos et al. 2015] Santos, G., Etien, A., Anquetil, N., Ducasse, S., and Valente, M. T. (2015). Recording and replaying system specific, source code transformations. In 2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation, SCAM ’15, pages 221–230, Piscataway, NJ, USA. IEEE Press.

[Schäfer et al. 2011] Schäfer, M., Sridharan, M., Dolby, J., and Tip, F. (2011). Refactoring Java programs for flexible locking. In Proceedings of the 33rd International Conference on Software Engineering, ICSE ’11, pages 71–80, New York, NY, USA. ACM.

[Schulze et al. 2013] Schulze, S., Lochau, M., and Brunswig, S. (2013). Implementing refactorings for FOP: Lessons learned and challenges ahead. In Proceedings of the 5th International Workshop on Feature-Oriented Software Development, FOSD ’13, pages 33–40, New York, NY, USA. ACM.
[Steimann et al. 2011] Steimann, F., Kollee, C., and von Pilgrim, J. (2011). A refactoring constraint language and its application to Eiffel. In Proceedings of the 25th European Conference on Object-oriented Programming, ECOOP’11, pages 255–280, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Sun et al. 2008] Sun, B., Chang, R.-Y., Chen, X., and Podgurski, A. (2008). Automated support for propagating bug fixes. In Proceedings of the 2008 19th International Symposium on Software Reliability Engineering, ISSRE ’08, pages 187–196, Washington, DC, USA. IEEE Computer Society.

[Tansey and Tilevich 2008] Tansey, W., and Tilevich, E. (2008). Annotation refactoring: In-ferring upgrade transformations for legacy applications. In Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications, OOPSLA ’08, pages 295–312, New York, NY, USA. ACM.

[Toomim et al. 2004] Toomim, M., Begel, A., and Graham, S. (2004). Managing duplicated code with linked editing. In Proceedings of the 2004 IEEE Symposium on Visual Languages and Human Centric Computing, VL-HCC ’14, pages 173–180, Piscataway, NJ, USA. IEEE Press.

[Vidal and Marcos 2012] Vidal, S. A. and Marcos, C. A. (2012). Building an expert system to assist system refactorization. Expert Systems with Applications, 39(3):3810–3816.

[Vidal and Marcos 2013] Vidal, S. A. and Marcos, C. A. (2013). Toward automated refactoring of crosscutting concerns into aspects. Journal of Systems and Software, 86(6):1482–1497.

[Wang et al. 2016] Wang, C., Jiang, J., Li, J., Xiong, Y., Luo, X., Zhang, L., and Hu, Z. (2016). Transforming Programs between APIs with Many-to-Many Mappings. In Krishnamurthi, S. and Lerner, B. S., editors, 30th European Conference on Object-Oriented Programming (ECOOP 2016), volume 56 of Leibniz International Proceedings in Informatics (LIPIcs), pages 25:1–25:26, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Wloka et al. 2009] Wloka, J., Sridharan, M., and Tip, F. (2009). Refactoring for reentrancy. In Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE ’09, pages 173–182, New York, NY, USA. ACM.

[Wu et al. 2015] Wu, L., Wu, Q., Liang, G., Wang, Q., and Jin, Z. (2015). Transforming code with compositional mappings for API-library switching. In 39th IEEE Annual Computer Software and Applications Conference, COMPSAC 2015, Taichung, Taiwan, July 1-5, 2015. Volume 2, pages 316–325.

[Xia and Lo 2016] Xia, X. and Lo, D. (2016). An effective change recommendation approach for supplementary bug fixes. Automated Software Engineering, pages 1–44.

[Xing and Stroulia 2007] Xing, Z. and Stroulia, E. (2007). API-evolution support with diff-catchup. IEEE Transactions on Software Engineering, 33(12):818–836.

[Zibran and Roy 2012] Zibran, M. F. and Roy, C. K. (2012). The road to software clone management: A survey. Technical report, pages 1–68.