CHAPTER

11 CHANGE COUPLING BETWEEN SOFTWARE ARTIFACTS: LEARNING FROM PAST CHANGES

Gustavo Ansaldi Oliva*, **Marco Aurélio Gerosa***

*Software Engineering & Collaborative Systems Research Group (LAPESSC), University of São Paulo (USP), São Paulo, Brazil**

CHAPTER OUTLINE

11.1 INTRODUCTION

Version control systems store and manage the history and current state of source code and documentation. As early as 1997, Ball and colleagues wrote a paper entitled "*If your version control system could talk...*" [1], in which they observed that these repositories store a great deal of contextual information about software changes. Over the years, researchers have leveraged such information to understand how software systems evolve over time, enabling predictions about their properties.

While mining these repositories, researchers observed an interesting pattern: certain artifacts are frequently committed together. These artifacts are connected to each other from an evolutionary point of view, in the sense that their histories intertwine. We call this connection *change coupling*. We also say that an artifact A is *change coupled* to B if A often co-changes with B. Other names employed in the literature include logical dependencies/coupling, evolutionary dependencies/coupling, and historical dependencies.

Change coupling can be calculated at different abstraction levels. In this chapter, we will focus on file-level change coupling. Analyzing change couplings at this level has two key benefits [2]: first and foremost, it can reveal hidden relationships that are not present in the code itself or in the documentation. For instance, a certain class A might be change coupled to another class B without structurally depending on it. Second, it relies on historical file co-change information only, which can be easily extracted from commit logs. Therefore, it does not require parsing code, making it more lightweight than structural analysis. It is also programming language agnostic, making it flexible and a good candidate to be used in studies that involve many subject systems written in different languages.

Most importantly, recent research has shown that detecting and analyzing change couplings supports a series of software development tasks. Suppose you are a software developer and just made a change to a certain part of a system. What else do you have to change? Based on the idea that artifacts that changed together in the past are bound to change together in the future, researchers leveraged change couplings to help answer this question [3]. Another traditional application regards discovering design flaws [2]. For instance, detecting and visualizing change couplings might reveal artifacts or modules that are being frequently affected by changes made to other parts of the system (encapsulation problem).

This chapter intends to provide researchers and practitioners with an overview of change coupling and its main applications. In Section 11.2, we introduce the concept of change coupling and highlight its key benefits. This will help you familiarize yourself with the topic and understand why it has been adopted so frequently in software engineering empirical studies. In Section 11.3, we present the main change coupling identification approaches, along with their key characteristics. We also link to tools and present code snippets that show how you can extract these couplings from the systems you develop, maintain, or analyze. In Section 11.4, we discuss the current key challenges of accurately identifying change coupling from version control systems. Besides highlighting fertile research areas that call for further exploration, we also provide some practical advice to help you identify change couplings more accurately. We conclude this section by discussing the trade-offs of identifying change coupling from monitored IDEs. In Section 11.5, we present the main applications researchers have discovered by detecting and analyzing change couplings from version control systems. This will provide you with a big picture of how researchers have leveraged these couplings to understand software systems, their evolution, and the developers around it. Finally, in Section 11.6, we draw our conclusions on the topic.

11.2 CHANGE COUPLING

According to Lanza et al. [8], "change coupling is the implicit and evolutionary dependency of two software artifacts that have been observed to frequently change together during the evolution of a software system." This term was introduced in late 2005 by Fluri et al. [4] and Fluri and Gall [5]. It gained more popularity with a book edited by Mens and Demeyer in 2008 called *Software Evolution* [6, 7]. Other studies then started employing it [8, 9]. As mentioned in the introduction, alternative terms found in the literature include logical dependency/coupling [10, 11], evolutionary dependencies/coupling [3, 12], and historical dependencies [13].

In the following, we go deeper into the concept of change coupling. In Section 11.2.1, we discuss the rationale behind change coupling by introducing one of the main forces that makes artifacts co-change. In Section 11.2.2, we highlight some key practical benefits of using change coupling.

11.2.1 WHY DO ARTIFACTS CO-CHANGE?

The ideas underlying the concept of co-changes and change coupling date back to the beginning of the 1990s when Page-Jones introduced the concept of "connascence" [14]. The term connascence is derived from Latin and means "having been born together." The Free Dictionary defines connascence as: (a) the common birth of two or more at the same time, (b) that which is born or produced with another, and (c) the act of growing together. Page-Jones borrowed the term and adapted it to the software engineering context: "connascence exists when two software elements must be changed together in some circumstance in order to preserve software correctness" [15].

Connascence assumes several forms and can be either explicit or implicit. To illustrate this point, consider the following code excerpt[1] written in Java and assume that its first line represents a *software element A* and that its second line represents a *software element B*:

```
String s;               //Element A (single source code line)
s = ‘‘some string’’;    //Element B (single source code line)
```

There are (at least) two examples of connascence involving elements A and B. If A is changed to `int s;` then B will have to be changed too. This is called *type connascence*. Instead, if A is changed to `String str;` then B will need to be changed to `str = ‘‘some string’’`. This is called *name connascence*. These two forms of connascence are called *explicit*. A popular manifestation of explicit connascence comes in the form of structural dependencies (e.g., methods calls). In turn, as we mentioned before, connascence can also be implicit, such as when a certain class needs to provide some functionality described in a design document.

[1]This example is adapted from Page-Jones' book [15].

In general, connascence involving two elements A and B occurs because of two distinct situations:

(a) A depends on B, B depends on A, or both: a classic scenario is when A changes because A structurally depends on B and B is changed, i.e., the change propagates from B to A via a structural dependency (e.g., a method from A calls a method from B). However, this dependency relationship can be less obvious, as in the case where A changes because it structurally depends on B and B structurally depends on C (transitive dependencies). Another less obvious scenario is when A changes because it semantically depends on B.

(b) Both A and B depend on something else: this occurs when A and B have pieces of code with similar functionality (e.g., use the same algorithm) and changing B requires changing A to preserve software correctness. As in the previous case, this can be less obvious. For instance, it can be that A belongs to the presentation layer, B belongs to the infrastructure layer, and both have to change to accommodate a new change (e.g., new requirement) and preserve correctness. In this case, A and B depend on the requirement.

Therefore, artifacts often co-change because of connascence relationships. This is what makes artifacts "logically" connected. Most importantly, the theoretical foundation provided by connascence is a key element that justifies the relevance and usefulness of change couplings.

11.2.2 BENEFITS OF USING CHANGE COUPLING

The use of change coupling in software engineering empirical studies is becoming more frequent every day. There are many reasons that justify this choice. In the following, we highlight some practical key benefits of detecting and analyzing change coupling.

Reveals historical relationships

Change couplings reveal hidden relationships undetectable by structural analysis. This means you may discover a change coupling involving two classes that are not structurally connected. Moreover, you may find change couplings involving artifacts of different kinds. For instance, you may find change couplings from domain classes to configuration files (e.g., XML files or Java .properties files), presentation files (e.g., HTML or JSP files), or build files (e.g., Maven's pom.xml file or Ant's build.xml file). These couplings are just undetectable by static analysis. Just to give a practical example, McIntosh and colleagues used change coupling to evaluate how tight the coupling is between build artifacts and production or test files [16]. Their goal was to study build maintenance effort and answer questions like "are production code changes often accompanied by a build change?"

Lightweight and language-agnostic

Change coupling detection from version control systems (file-level change coupling) is often lightweight, as it relies on co-change information inferred from change-sets. It is also programming-language-agnostic, as it does not involve parsing source code. This makes it a good choice for empirical studies involving several systems written in different languages. Approaches for detecting change couplings are presented in Section 11.3. We note, however, that accurately identifying change couplings is still a challenging task due to noisy data and certain commit practices. These problems are discussed in Section 11.4.

Might be more suitable than structural coupling

Certain empirical studies admit the use of both structural coupling and change coupling. Researchers have shown that using change coupling leads to better results for some specific applications. For instance, Hassan and Holt [17, 18] showed that change couplings were far more effective than structural couplings for predicting change propagation (more details in Section 11.5.1.1). In addition, Cataldo and Herbsleb [19] showed that change couplings were more effective than structural couplings for recovering coordination requirements among developers (more details in Section 11.5.4). An important aspect, however, is that change coupling requires historical data. If you are just starting a project and no historical data is available, then using structural analysis might be the only way out. In fact, a promising research area concerns conceiving hybrid approaches that combine structural analysis with historical data to cope with the dynamics of software development [20].

Relationship with software quality

Researchers have found evidence that change couplings detected from version control systems provide clues about software quality. D'Ambros et al. [8] mined historical data from three open source projects and showed that change couplings correlate with defects extracted from a bug repository. Cataldo et al. [21] reported that the effect of change coupling on fault proneness was complementary and significantly more relevant than the impact of structural coupling in two software projects from different companies. In another study, Cataldo and Nambiar [22] investigated the impact of geographic distribution and technical coupling on the quality of 189 global software development projects. By technical coupling, they mean overall measures of the extent to which artifacts of the system are connected. Their results indicated that the number of change couplings among architectural components were the most significant factor explaining the number of reported defects. Other factors they took into consideration include the number of structural coupling, process maturity, and the number of geographical sites. In Sections 11.5.2 and 11.5.3, we describe visualization techniques and metrics to help manage change couplings and uncover design flaws.

Broad applicability

Besides its relation with software quality, analyzing change couplings is useful for a series of key applications, which include change prediction and change impact analysis (Section 11.5.1), discovery of design flaws and opportunities for refactoring (Section 11.5.2), evaluation of software architecture (Section 11.5.3), and detection of coordination requirements among developers (Section 11.5.4).

11.3 CHANGE COUPLING IDENTIFICATION APPROACHES

Identifying and quantifying change couplings inherently depends on how the artifacts' change history is recovered. Due to practical constraints, change couplings are often detected by parsing and analyzing the logs of version control systems (e.g., CVS, SVN, Git, and Mercurial). In this case, the change history of an artifact (file) is determined based on the commits in which such artifact appears. In most cases, researchers assume that if two artifacts are included in the same commit, then these artifacts co-changed. The more two artifacts are committed together, the more change coupled they become. In older version control systems that do not support atomic commits, such as CVS, some preprocessing is often needed to reconstruct change transactions [23].

In this section, we show how change couplings can be both discovered and quantified from the logs of version control systems. In the following, we present three approaches: raw counting (Section 11.3.1), association rules (Section 11.3.2), and time-series analysis (Section 11.3.3). Given the broad adoption of the first two, we provide code snippets and instructions to help you run them in your projects. All code is available on GitHub at https://github.com/golivax/asd2014.

11.3.1 RAW COUNTING

Several studies identify change couplings using a raw counting approach. In this approach, change-sets are mined from the logs of the version control system and co-change information is stored in a suitable data structure or in a database. A commonly used data structure is an *artifact* $\times$ *artifact* symmetric matrix in which each cell $[i, j]$, with $i \neq j$, stores the number of times that artifact i and artifact j changed together (i.e., appeared in the same commit) over the analyzed period. In turn, cell $[i, i]$ stores the number of times artifact i changed in this same period. We call this a co-change matrix (Figure 11.1).

Once the co-change matrix is built, change couplings can be inferred using the following two strategies:

(a) Non-directed relationship. Every non-zero cell $[i, j]$ out of the main diagonal implies the existence of a change coupling involving artifacts i and j. In this approach, change coupling is regarded as a non-directed relationship, i.e., it is not possible to know if artifact i is coupled to artifact j, if artifact j is coupled to artifact i, or both. In the example from Figure 11.1, the cell $[2, 1] = 3$ implies the existence of a change coupling involving A and B whose strength is 3. That is, coupling strength simply corresponds to the number of times the involved artifacts changed

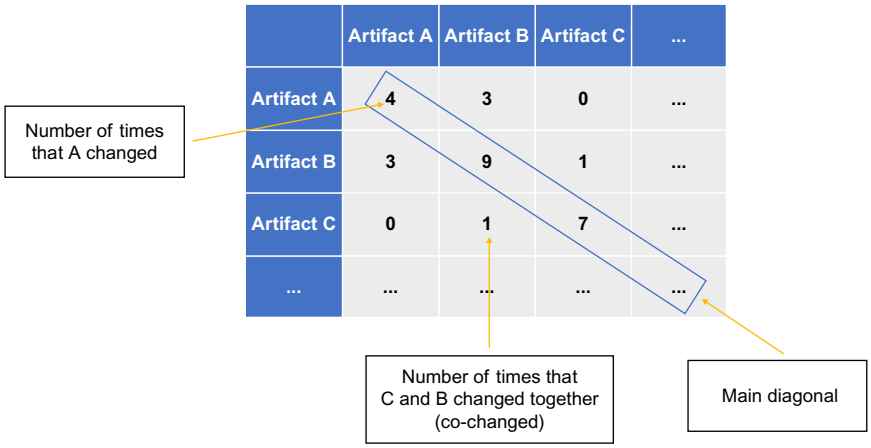

FIGURE 11.1

Hypothetical co-change matrix.

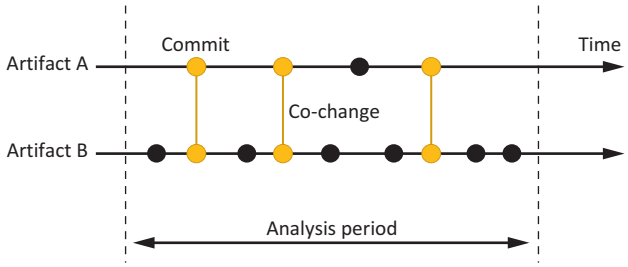

FIGURE 11.2

Scenario for change coupling analysis.

together. The rationale is that pairs of artifacts that co-change more often have a stronger evolutionary connection than those pairs that co-change less often. In the data mining field, this measure is known as *support*.

(b) Directed relationship. This approach is analogous to the previous one with regards to the identification of change couplings. However, this approach assumes that the strength of these couplings can be different. The strength of a change coupling from A to B is determined by the ratio of co-changes (support) and the number of times the artifact B changed. In the example from Figure 11.1, the strength of the change coupling from A to B would be $\text{cell}[1,2]/\text{cell}[2,2] = 3/9 = 0.33$. In turn, the strength of the change coupling from B to A would be $\text{cell}[1,2]/\text{cell}[1,1] = 3/4 = 0.75$. Therefore, this approach assumes that the last coupling is much stronger than the first one, since commits that include A often include B as well (Figure 11.2). In the data mining field, this measure is known as *confidence*.

Some of the studies that have employed this identification method include those of Gall et al. [24], Zimmermann et al. [25], and Oliva and Gerosa [26].

Determining relevant couplings

When identifying change couplings via *raw counting with non-directed relationship*, relevant couplings are often filtered by choosing a *support* threshold that is suitable for the study at hands. One approach is to iteratively test different values and qualitatively analyze the output. An alternative approach is to perform a statistical analysis of the distribution of support, such as a quartile analysis (Figure 11.3).

In the scope of a quartile analysis, one approach is to consider that sufficiently relevant couplings have a support value above the third quartile. The third quartile ($Q3$) is the middle value between the median and the highest value of the data, splitting off the highest 25% of data from the lowest 75%. Another approach consists of taking the upper whisker (a.k.a. upper fence) as the threshold. The upper whisker is determined using the following formula: $Q3 + [1,5^*\,(\text{IQR})] = Q3 + [1,5^*\,(Q3 - Q1)]$. In a quartile analysis, values above the upper whisker are frequently considered as outliers. Therefore, this latter approach is more restrictive than the former, in the sense that only unusually highly evident couplings are selected. An even more restrictive approach would only take extreme outliers, which are those above the threshold given by the following formula: $Q3 + 3^*\,(\text{IQR})$.

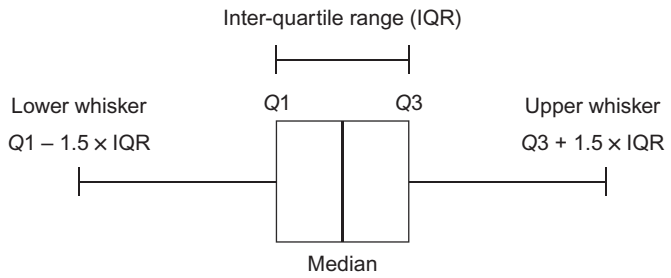

FIGURE 11.3

Schematics of a boxplot.[2]

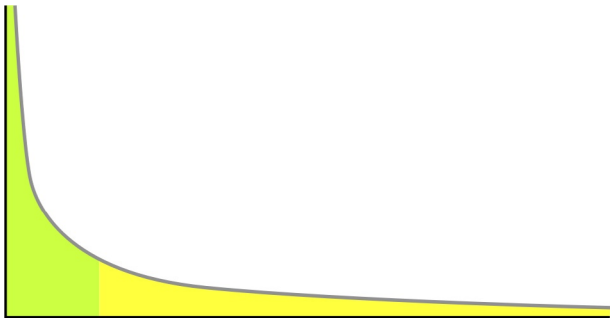

FIGURE 11.4

Power law distribution.[3]

Although the quartile analysis is simple and straightforward, we note that it might not be suitable for some studies, since the support distribution tends to follow a power law (Figure 11.4). This distribution is very right-skewed (long tail to the right) and traditional statistics based on variance and standard deviation often provide biased results. A more cautious approach would be thus to skip the green (left-hand side) part of Figure 11.4, which corresponds to 80% of the data, and stick to the yellow (right-hand side) part, which represents the frequently co-changed artifacts. In any case, researchers should always validate and analyze the output produced by each filtering strategy.

When identifying couplings via *raw counting with non-symmetric strength*, relevant couplings are determined based on paired values of support and confidence. In this mindset, support has been interpreted as a measure of how evident a certain change coupling is, in the sense that couplings

[2]Adapted from http://commons.wikimedia.org/wiki/File:Boxplot_vs_PDF.svg (CC Attribution-Share Alike 2.5 Generic).
[3]http://commons.wikimedia.org/wiki/File:Long_tail.svg Picture by Hay Kranen/PD.

supported by many co-changes are more evident than couplings supported by few co-changes [25]. In turn, confidence has been interpreted as a measure of the strength of the coupling. According to Zimmermann and colleagues, confidence actually answers the following question: of all changes to an artifact, how often (as a percentage) was some other specific artifact affected? Therefore, *relevant couplings* have sufficiently high values of support and confidence, i.e., they are simultaneously evident and strong.

A threshold for support is often the first parameter to be determined. As shown before, this can be calculated using different approaches. After having determined the couplings that are sufficiently evident, a confidence threshold is applied to filter those that are also sufficiently strong. The most conservative approach is to iteratively try different values of confidence, analyze the output, and pick the threshold that provides the best results for the study at hand [25]. To that end Zimmermann and colleagues implemented visualization techniques (*pixelmaps* and *3D bar charts*) to help spot coupling with high values of support and confidence (although high ends up being a little bit subjective in this case).

An alternative approach consists of performing a statistical analysis of the confidence distribution (such as a quartile analysis). Another idea is to establish three adjacent intervals for the confidence values [26] as follows:

- Confidence [0.00, 0.33]: low coupling
- Confidence [0.33, 0.66]: regular coupling
- Confidence [0.66, 1.00]: strong coupling

Tools

In this section, we show how to extract change couplings using the raw counting approach. In fact, we focus on how to parse a list of change-sets from either SVN or Git, since this is the most difficult part of the process. In the sections that follow, we show UML diagrams and Java code snippets. A complete implementation is available in GitHub.[4] We emphasize that such an implementation is just a starting point from you can develop more sophisticated or tailored solutions.

(1) Change-Set class

We use a very simple `ChangeSet` class to store the change-set of each commit (Figure 11.5). This class has an attribute called `commitID`, which stores the id of the commit. The other attribute is a set called `changedArtifacts`, which we will use to store the name of the artifacts changed in the associated commit. The class has two constructors: one to handle numeric commit ids (SVN) and another to handle alphanumeric commit ids (Git).

(2) Extracting change-sets from SVN

In order to extract change-sets from the SVN repository, we rely on the SVNKit[5] framework. SVNKit is a Java implementation of SVN, offering an API to work with both local and remote repositories. We use this framework to parse SVN commit logs and extract the change-sets. The class

[4] https://github.com/golivax/asd2014.
[5] http://svnkit.com/.

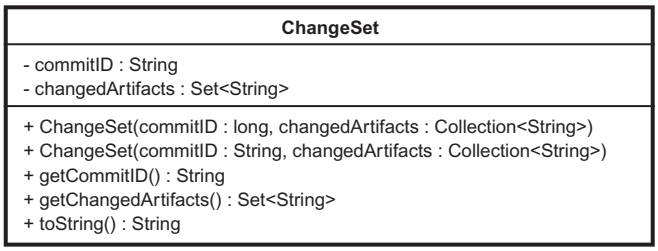

FIGURE 11.5

ChangeSet *class* (/src/main/java/br/usp/ime/lapessc/entity/ChangeSet.java).

/src/main/java/br/usp/ime/lapessc/svnkit/SVNKitExample.java in our GitHub is a complete example that shows how to extract change-sets from SVN.

In the code excerpt that follows, we show how to initialize the repository. It requires the URL of the repository and credentials access (username and password). Most repositories from open source software systems enable read permission with the username "anonymous" with password "anonymous."

```
DAVRepositoryFactory.setup();
try {
    SVNRepository repository = SVNRepositoryFactory.create(
            SVNURL.parseURIEncoded(url));
    ISVNAuthenticationManager authManager =
            SVNWCUtil.createDefaultAuthenticationManager(username, password);
    repository.setAuthenticationManager(authManager);
}catch(SVNException e){
    // Deal with the exception
}
```

After initializing the repository, we are ready to interact with it. The method `log (targetPaths, entries, startRev,endRev,changedPath,strictNode)` from the `SVNRepository` class performs a SVN log operation from `startRev` to `endRev`. The specific meaning of each parameter in this method call is summarized as follows. We refer readers to the SVNKit's documentation[6] for further details about this method.

- `targetPaths`—paths that mean only those revisions at which they were changed

[6] http://svnkit.com/javadoc/index.html.

- `entries`—if not null then this collection will receive log entries
- `startRevision`—a revision to start from
- `endRevision`—a revision to end at
- `changedPath`—if true then revision information will also include all changed paths per revision
- `strictNode`—if true then copy history (if any) is not to be traversed

This log method outputs a collection of `SVNLogEntry`, which are the objects that represent commit logs. In the following, we show an example of how to extract the commit id (revision number) and the change-set from each `SVNLogEntry` instance.

```
Set<ChangeSet> changeSets = new LinkedHashSet<ChangeSet>();
for(SVNLogEntry logEntry : logEntries){
    Map<String,SVNLogEntryPath> changedPathsMap = logEntry.
    getChangedPaths();
    if (!changedPathsMap.isEmpty()) {
        long revision = logEntry.getRevision();
        Set<String> changedPaths = logEntry.getChangedPaths().keySet();
        ChangeSet changeSet = new ChangeSet(revision, changedPaths);
        changeSets.add(changeSet);
    }
}
```

The `main()` method shown below triggers the mining with five important parameters: repository URL (`url`), user name (`name`), password (`password`), start revision number (`startRev`), and end revision number (`endRev`). In this example, we extract the change-sets from an open-source project called Moenia,[7] which is hosted by SourceForge and contains only 123 revisions.

```
public static void main(String[] args) {
        String url = "https://github.com/golivax/JDX.git";
        String cloneDir = "c:/tmp/jdx";
        String startCommit = "ca44b718d43623554e6b890f2895cc80a2a0988f";
        String endCommit = "9379963ac0ded26db6c859f1cc001f4a2f26bed1";
```

[7] http://sourceforge.net/projects/moenia/.

```
        JGitExample jGitExample = new JGitExample();
        Set<ChangeSet> changeSets =
           jGitExample.mineChangeSets(url,cloneDir,startCommit,
                endCommit);
        for(ChangeSet changeSet : changeSets){
            System.out.println(changeSet);
        }
    }
```

(3) Extracting change-sets from Git

In order to extract change-sets from Git repositories, we suggest using the JGit[8] framework. JGit is a Java implementation of Git, offering a fluent API to manipulate Git repositories. We use this framework to parse Git commit logs and extract the change-sets. Given the distributed nature of Git, as well as its particular notion of branching, extracting change-sets is a little bit more complicated. A complete example showing how to mine change-sets from Git repositories is available on GitHub.[9]

The first thing we need to do is to clone the Git repository (or open an already cloned repository). The code excerpt below shows how to accomplish this programmatically using the JGit API. In this example, the repository in the URL is cloned to `cloneDir`.

```
//Cloning the repo
Git git = Git.cloneRepository().setURI(url).
    setDirectory(new File(cloneDir)).call();
//To open an existing repo
//this.git = Git.open(new File(cloneDir));
```

After that, we need to decide from which branch we will extract the commits in the range [`startCommitID`, `endCommitID`]. A popular strategy to extract the "main branch" is follow the first parent of every commit. Hence, what we do is start at the `endCommitID` and keep on following the first parent of every commit until the first parent of the `startCommitID` is reached. This is equivalent to the following git command: git log `startCommitID`∧∧ `..endCommitID` –first-parent. The meaning of the –first-parent parameter is as follows (extracted from the Git manual):

> **–first-parent:** Follow only the first-parent commit upon seeing a merge commit. This option can give a better overview when viewing the evolution of a particular topic branch, because

[8] http://eclipse.org/jgit.
[9] https://github.com/golivax/asd2014/src/main/java/br/usp/ime/lapessc/jgit.

merges into a topic branch tend to be only about adjusting to updated upstream from time to time, and this option allows you to ignore the individual commits brought in to your history by such a merge.

The method `List<RevCommit> getCommitsInRange(String startCommitID, String endCommitID)` in the `JGitExample` class implements this mining strategy, recovering the following commits:

```
endCommit,firstParent(endCommit),..., startCommit, firstParent(startCommit)
```

Git always determines change-sets by comparing two commits. However, if the chosen start commit happens to be the first commit in the repository, then it has no parent. We treat this special case by determining the artifacts that were *added* in the first commit. The following code excerpt shows how we implemented this algorithm using JGit. Note, however, that we omitted the details of recovering the actual change-sets, as the code is a bit lengthy.

```
private Set<ChangeSet> extractChangeSets(List<RevCommit> commits) throws
        MissingObjectException, IncorrectObjectTypeException,
        CorruptObjectException, IOException {

        Set<ChangeSet> changeSets = new LinkedHashSet<ChangeSet>();

        for(int i = 0; i < commits.size() - 1; i++){
            RevCommit commit = commits.get(i);
            RevCommit parentCommit = commits.get(i+1);
            ChangeSet changeSet = getChangeSet(commit, parentCommit);
            changeSets.add(changeSet);
        }


        //If startCommit is the first commit in repo, then we
        //need to do something different to get the changeset
        RevCommit startCommit = commits.get(commits.size()-1);
        if(startCommit.getParentCount() == 0){
            ChangeSet changeSet = getChangeSetForFirstCommit(startCommit);
            changeSets.add(changeSet);
        }
```

```
        return changeSets;
    }
```

The `main()` method shown below triggers the `mineChangeSets()` method with four important parameters: remote repository URL (`url`), path to local repository (`cloneDir`), start commit id (`startCommit`), and end commit id (`endCommit`). In this example, we extract the change-set from an open-source project called JDX,[10] hosted on GitHub.

```
public static void main(String[] args) {
    String url = "https://github.com/golivax/JDX.git";
    String cloneDir = "c:/tmp/jdx";
    String startCommit = "ca44b718d43623554e6b890f2895cc80a2a0988f";
    String endCommit = "9379963ac0ded26db6c859f1cc001f4a2f26bed1";

    JGitExample jGitExample = new JGitExample();
    Set<ChangeSet> changeSets =
            jGitExample.mineChangeSets(url,cloneDir,startCommit, endCommit);
    for(ChangeSet changeSet : changeSets){
        System.out.println(changeSet);
    }
}
```

11.3.2 ASSOCIATION RULES

In this section, we present the concepts of *frequent itemsets* and *association rules*, which were introduced in the domain of data mining. To this end, we heavily rely on the reference book from Rajaraman, Leskovec, and Ullman entitled *Mining of Massive Datasets* [27] to introduce some fundamental definitions. Interested readers may want to refer to Chapter 4 of that book for more detailed information. As supplementary material, we also recommend the second chapter of the book from Liu entitled *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data* [28].

In the field of data mining, a model often referred to as *market-basket* is used to describe a common form of many-to-many relationship between two kinds of objects: *items* and *baskets* (or transactions). Each basket comprises a set of items, also known as an *itemset*. The number of items in a basket is usually assumed to be small, much smaller than the total number of different items available. In turn, the number of baskets is usually assumed to be very large, meaning that it would be not possible to store them all in main memory. A set of items (itemset) that appears in many baskets is said to be

[10] https://github.com/golivax/JDX/.

frequent. If I is an itemset, then the *support* for I, written as *support(I)*, indicates the number of baskets for which I is a subset.

Itemsets are often used to build useful if-then rules called *association rules*. An association rule is an implication of the form $I \Rightarrow J$, which states that when I occurs, J is likely to occur. In this context, I and J are two disjoint sets of items (itemsets). I is called the antecedent (left-hand-side or LHS) and J is called the consequent (right-hand-side or RHS). For instance, the rule $\{x, y\} \Rightarrow \{z\}$ found in the sales data of a supermarket would indicate that if a customer buys products x and y together, this same customer is also likely to buy z. If one is looking for association rules $I \Rightarrow J$ that apply to a reasonable fraction of the baskets, then the support for $I \cup J$ must be reasonably high. This value is called the *support of the rule* and is written as $support(I \Rightarrow J) = support(I \cup J)$. It simply corresponds to the number of baskets that contain both I and J. The strength of the rule, in turn, is given by a measure called *confidence*. It defines the fraction of baskets containing I where J also appears. Formally:

$$confidence(I \Rightarrow J) = \frac{support(I \Rightarrow J)}{support(I)} = \frac{support(I \cup J)}{support(I)}$$

As we showed in Section 11.3.1, *support* and *confidence* can be calculated for pairs of items using an exhaustive approach, which consists of determining (i) the number of times each item has changed, as well as (ii) the number of times two artifacts have changed together. The main difference here lies in the application of an algorithm to discover frequent itemsets (from which useful rules are then generated). Most frequent itemset mining algorithms, such as the often used *Apriori* [29], require an itemset support threshold as input. Hence, non-frequent (irrelevant) items are preemptively removed. Furthermore, *Apriori* is more flexible, in the sense that it can discover association rules involving an arbitrary number of items in both the LHS and the RHS.

Researchers have often formalized change couplings as association rules: the version control system (database) stores all commit logs, and each commit log (basket) contains a set of modified files (itemset). A change coupling from a versioned artifact x_2 (client) to another versioned artifact x_1 (supplier) can be written as an association rule $X_1 \Rightarrow X_2$, whose antecedent and consequent are both singletons that contain x_1 and x_2, respectively.

Some of the studies that have employed the association rules identification method include those of Bavota et al. [30], Zimmermann et al. [3, 31], Wang et al. [32], and Ying et al. [33]. The first applies the *Apriori* algorithm. The second and third studies perform adaptations to the original *Apriori* algorithm to speed up the calculation of rules, which often involve constraining antecedents and/or consequents based on some study-specific criteria. The last study uses a different (and more efficient) algorithm called FP-Growth [34], which avoids the step of generating candidate itemsets and testing them against the entire database.

Determining relevant couplings

When formalizing change couplings as association rules, researchers often determine their relevance based on thresholds for support and confidence (input to rule mining algorithms). However, as we showed in Section 11.3.1, determining thresholds from which rules become sufficiently relevant is

complicated and depends on the characteristics of the project at hand. For instance, Zimmermann et al. [25] considered relevant those change couplings with support greater than 1 and confidence greater than 0.5. In turn, Bavota et al. [30] considered relevant those couplings that included elements that co-changed in at least 2% of the commits (support/number of commits) and whose confidence scored at least 0.8. According to Zimmermann et al. [3], in practice support against a combination of the average transaction size or the average number of changes per item should be normalized. However, as they pointed out themselves, choosing the right normalization is still research on its own.

Although support and confidence are the most common thresholds to capture relevant change couplings, other approaches do exist. One alternative is to use the *conviction* measure [35]. Given two elements A and B, conviction is $P(A)P(\sim B)/P(A \text{ and } \sim B)$. The implication $A \rightarrow B$ is tautologically equivalent to $(\sim A \text{ or } B)$, which is in turn equivalent to $\sim(A \text{ and } \sim B)$. The idea is then to measure how far $(A \text{ and } \sim B)$ deviates from independence and invert the ratio to take care of the outside negation. According to Brin et al. [35], "conviction is truly a measure of implication, because it is directional, it is maximal for perfect implications, and it properly takes into account both $P(A)$ and $P(B)$" (as opposed to confidence, which ignores $P(B)$).

Tools

Implementations of the *Apriori*, as well as other frequent pattern mining algorithms, are available in Weka,[11] SPMF,[12] and R[13] (*arules* package[14]). These are all open-source projects licensed under GPL (GNU General Public License). In the following, we show how to extract change couplings using R and the arules package. As in Section 11.3.1, the subject system for this example will be Moenia.

(1) Load the library

The first step is to install and load the arules package into your R session. To load the package, run the following command:

```
library(arules)
```

(2) Read transactions from CSV file

The second step is to read change transactions (change-sets) from a CSV file:

```
trans = read.transactions("moenia.csv", format = "basket", sep = ",")
```

The `read.transactions()` function has three basic parameters. The first points to the CSV file, which is read from the working directory by default. You can execute the `getwd()` function in R to discover the current working directory. The second parameter indicates the format of this CSV. In the

[11] http://www.cs.waikato.ac.nz/ml/weka/.
[12] http://www.philippe-fournier-viger.com/spmf/.
[13] http://www.r-project.org/.
[14] http://cran.r-project.org/web/packages/arules/index.html.

basket format, each line in the transaction data file represents a transaction where the items (item labels) are separated by the characters specified bysep (third parameter). More information about this function can be found in the arules package manual.[15]

A CSV in basket format can be built out of a collection of `ChangeSet` instances very easily. In the following code snippet, we show one possible solution that captures files with the .java extension:

```
public String toCSV(Set<ChangeSet> changeSets){
    String csv = new String();
    for(ChangeSet changeSet : changeSets){
        String cs = new String();
        for(String artifact : changeSet.getChangedArtifacts()){
            if(artifact.endsWith(".java")){
                cs+=artifact + ",";
            }
        }
        //StringUtils is a class from the Apache Commons Lang library
        cs = StringUtils.removeEnd(cs, ",");
        if(!cs.isEmpty()){
            csv+=cs + "\n";
        }
    }
    return csv;
}
```

(3) Compute rules using Apriori

In the third step, we run the *Apriori* algorithm to compute the association rules. This is done via the `apriori()` function. The function parameters we will use are as follows:

- transactions: the set of input transactions
- support: a numeric value for the minimal support of an item set (default: 0.1)
- confidence: a numeric value for the minimal confidence of rules (default: 0.8)
- minlen: an integer value for the minimal number of items per item set (default: 1)
- maxlen: an integer value for the maximal number of items per item set (default: 10)

The transactions will be the ones obtained in the following step. The support parameter is defined in relative terms. That is, if we want to find rules whose elements have changed together at least four

[15] http://cran.r-project.org/web/packages/arules/arules.pdf.

times, then the parameter should be 4/number of change-sets (i.e., number of lines in the CSV file). Confidence is defined just like we did in this section. Supposing we want to calculate rules with a single antecedent and a single consequent, then minlen and maxlen should be both 2.

If the data set is not too large, we can first obtain all possible rules and then start tweaking the parameters to obtain better rules. To obtain all rules, we use the following trick: set support equal to `.Machine$double.eps`. This reserved keyword outputs the smallest positive floating-point number the machine can produce. If we had set support equal to zero, then the `apriori()` function would generate artificial rules (rules with support equal to zero). The complete command is shown below.

```
rules <- apriori(trans, parameter = list(
    support = .Machine$double.eps,
    confidence = 0, minlen = 2, maxlen = 2))
```

(4) Inspect rules, analyze output, and fine-tune parameters

In this last step, we inspect rules, analyze the output, and fine-tune the parameters to obtain both evident and strong change couplings. The following command helps to investigate the rules produced in the prior step:

```
> summary(rules)
set of 10874 rules


rule length distribution (lhs + rhs):sizes
    2
10874

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      2       2       2       2       2       2

summary of quality measures:
    support          confidence          lift
 Min.   :0.01111   Min.   :0.02941   Min.   : 0.8823
 1st Qu.:0.01111   1st Qu.:0.25000   1st Qu.: 5.0000
 Median :0.01111   Median :0.33333   Median : 7.5000
 Mean   :0.01834   Mean   :0.45962   Mean   :12.0304
 3rd Qu.:0.02222   3rd Qu.:0.61538   3rd Qu.:15.0000
 Max.   :0.14444   Max.   :1.00000   Max.   :90.0000
```

Given the distribution of support, we use the extreme outliers approach shown in Section 11.3.1 to restrict the set of produced rules: $Q3 + 3^*$ IQR $= 0.02222 + 1.5^*(0.02222 - 0.01111) = 0.038885$. We also take a confidence value of 0.66. Now, we recalculate the rules as follows:

```
rules <- apriori(trans, parameter = list(
    support = 0.038885, confidence = 0.66, minlen = 2, maxlen = 2))
```

Analyzing the produced rules via the `summary()` command shows that we now have 304 rules. Of course, we could tweak the parameters again to obtain a smaller set of rules. To inspect all rules, we just type the following command:

```
inspect(rules)
```

Here, we should not forget that a certain rule $A_1 \Rightarrow A_2$ indicates that A_2 is impacted by changes to A_1, i.e., A_2 is the client and A_1 is the supplier. Finally, to export those rules to a CSV file, we can use the following command:

```
write(rules, file = "data.csv", sep = ",", col.names = NA)
```

11.3.3 TIME-SERIES ANALYSIS

Time-series representation has been successfully employed in different domains (e.g., image/speech processing and stock-market forecasting) to detect commonly occurring similar phenomena that evolve over time [36]. In fact, since the early days of mining software repositories, time-series analysis and associated metrics have been identified as a key research area in the understanding of how software structures change over time [1]. Time series analysis has emerged as a promising approach to cope with some of the problems found in earlier detection algorithms. According to Canfora et al. [37], "although association rules worked well in many cases, they fail to capture logical coupling relations between artifacts modified in subsequent change-sets."

11.3.3.1 Dynamic time warping

Dynamic time warping (DTW) is a technique introduced by Kruskal and Liberman [38] to find an optimal alignment between two given (time-dependent) sequences under certain restrictions (Figure 11.6). The algorithm warps sequences in a non-linear fashion so that they meet each other. In other words, DTW can distort (warp) the time axis, compressing it in some intervals and expanding it in others [39]. Originally, DTW was used to compare different speech patterns in automatic speech recognition systems [40, 41]. A traditional application consists of determining whether two wave forms represent the same spoken phrase under different pronunciation speed, accent, and pitch [39]. DTW was

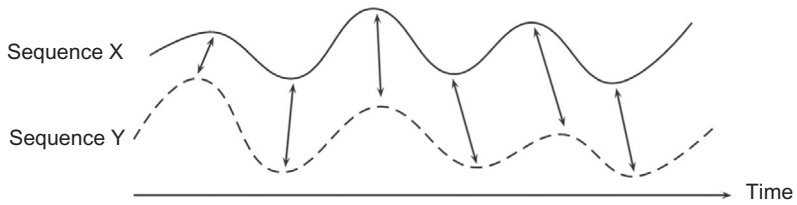

FIGURE 11.6

Time alignment of two time-dependent sequences. Arrows indicate aligned points.[16]

then successfully applied to a number of different domains, including medicine [42], robotics [43], and handwriting recognition [44]. More information about the DTW algorithm can be found in Chapter 4 of the book by Müller [45] entitled "Information Retrieval for Music and Motion."

In the case of change couplings, the set of time instants in which a versioned artifact changes is modeled as a time-series sequence. These sequences are then compared in a pair-wise fashion to determine how well they align. If such sequences align sufficiently well, then it is possible to state that there is a change coupling between the associated artifacts. Some of the studies that employed this identification method include those of Antoniol et al. [39] and Bouktif et al. [46].

Determining relevant couplings

Antoniol et al. [39] compute the DTW distance for every pair of time series in order to detect co-changing files in CSV. Instead of comparing the whole time series, they do it incrementally using windows that include a certain amount of data points (changes). They also do it backwards, i.e., time series starting from the most recent change. File histories with a distance below a certain threshold are considered indistinguishable and belonging to the same history group. In their paper, they experimented from 60 s up to 1200 s in the Mozilla project and showed that threshold values indeed change the number and size of groups (of co-changed files). This threshold can be used to fine-tune sensitivity, since its optimal value is project-dependent. In their paper, they report the results they obtained with threshold values of 270, 600, and 1200 s (4.5, 10, and 20 min, respectively). In a follow-up paper, Bouktif et al. [46] analyzed the change history of a small project called PADL stored in CVS. The authors showed that threshold values between 43,200 s (12 h) and 86,400 s (24 h) were a good compromise between precision and recall. They ended up using the threshold of 86,400 s in their case study, resulting in a minimum average (weighted) precision of 84.8% and recall of 71.8%. To calculate precision and recall, they performed a k-fold cross-validation, by dividing change histories in training and test sets. More details can be obtained in their papers.

Tools

The R statistical tool features a package called dtw[17] that implements the DTW algorithm. Using this package and running the algorithm with the default settings is straightforward, as shown in

[16]Extracted as is from Müller [45]—p. 70. Content licensed by Springer. Copyright cleared.
[17]http://dtw.r-forge.r-project.org/.

the package's guide.[18] The FastDTW[19] is a library written in Java that implements a variation of the original DTW algorithm called FastDTW [47]. This variation provides optimal or near-optimal alignments with an O(N) time and memory complexity, in contrast to the $O(N^2)$ requirement of the standard DTW algorithm.

11.3.3.2 Granger causality test

Granger causality is a statistical hypothesis test for determining whether a time-series sequence is useful in forecasting another [48]. In other words, the algorithm tests for predictive causality. Formally, a time series X is said to Granger-cause Y if it can be shown, usually through a series of t-tests and F-tests on lagged values of X (and with lagged values of Y also included), that those X values provide statistically significant information about future values of Y. The basic idea is that the cause cannot come after the effect. Hence, if a variable x affects a variable y, then the former should help improving the predictions of the latter [37].

Ceccarelli et al. [49] used the bivariate Granger causality test to address the issue of detecting change couplings between artifacts that are modified in subsequent change-sets. They model the time series of a versioned artifact f_k as follows. Let $f_k(t),\ t = 1, \ldots, t$ be the change time series of the versioned artifact f_k defined as:

$$f_k(t) = \begin{cases} 1, & f_k \in \Delta_t, \\ 0, & f_k \notin \Delta_t, \end{cases}$$

i.e., $f_k(t)$ is one if the file f_k was changed in snapshot Δ_t, zero otherwise. The authors found that the number of relevant recommendations provided by the Granger causality test is complementary to those inferred by association rules. In a follow-up study by the same authors [37], they used a slightly more sophisticated model, in which they replaced the binary variable $f_k(t)$ with a continuous variable that accounted for the number of changes that each file underwent during the test period. Their evaluation of four open source systems showed that while association rules provided more precise results, the Granger causality test achieved better results for recall and F-measure. Again, they highlighted that the set of true couplings provided by the two techniques is mostly disjoint.

Determining relevant couplings

The null hypothesis (H_0: file f_1 does not Granger-cause f_2) is rejected based on the calculation of a score S that takes into account the sum of squared residuals. If such score is higher than the 5% critical value for an F(p,T-2p-1) distribution, then the hypothesis is rejected [37]. The set of relevant couplings comprise the top N in the list of versioned artifacts pairs ranked by S in decreasing order. The optimal value for N seems to be project-dependent.

[18] http://cran.r-project.org/web/packages/dtw/vignettes/dtw.pdf.
[19] http://code.google.com/p/fastdtw/.

Tools

The Granger causality test is available in R via the package MSBVAR.[20] The granger.test section of the reference manual[21] shows how to execute the statistical test. It also describes the input parameters, as well as the kind of output given by the function.

11.4 CHALLENGES IN CHANGE COUPLING IDENTIFICATION

Despite the importance of change couplings, accurately detecting them is far from trivial. In Section 11.4.1, we discuss how certain commit practices impair change coupling identification. In Section 11.4.2, we give some guidelines to help avoid noise and improve the accuracy of change coupling detection. Finally, in Section 11.4.3, we discuss the trade-offs of an alternative approach that involves detecting change couplings from monitored IDEs.

11.4.1 IMPACT OF COMMIT PRACTICES

Despite the flexibility and relevance of change couplings, accurately identifying them from the logs of version control system is far from trivial. When operationalizing change couplings this way, their detection become subject to different developers' commit practices. For instance, while a certain developer might commit very frequently, another developer might work for a long period in the code and commit all changes at once. This latter scenario favors the appearance of *overloaded commits*, i.e., commits with tangled changes [25, 50]. In turn, overloaded commits generate artificial change couplings, which link artifacts that belong to different changes. An example would be a commit in which a certain developer implements a new feature in a set of files, as well as fixes an unrelated bug in other files. In such case, change couplings will link artifacts related to the new feature with artifacts related to the bug fix. Another problematic situation refers to developers who produce *incomplete commit*s. By incomplete commits, we mean those in which a developer forgets to perform a certain action or deliberately splits a single well-defined change into several consecutive commits. For instance, it might be that a developer changes certain domain classes but forgets to update an associated XML configuration file. This scenario leads to missing change couplings, because the developer will perform the forgotten actions in a separate commit. Finally, some commits might be the result of merging two branches from the repository. Such commits often involve a large number of artifacts and therefore originate several artificial change couplings. Detecting and coping with these problems is research on its own.

Researchers have recently tried to characterize developers' commit behavior. Ma et al. [51] conducted an empirical investigation on commit intervals in four open source projects from the Apache Software Foundation. In particular, they tried to fit lifecycle- and release-level commit intervals into well-known statistical distributions. They found that their datasets often presented a power-law distribution, i.e., most of the intervals between two consecutive commits in the repository were very short and only a few were distinctively high. Lin et al. [52] discovered that the number of commits per

[20] http://cran.r-project.org/web/packages/MSBVAR.
[21] http://cran.r-project.org/web/packages/MSBVAR/MSBVAR.pdf.

class (from creation of the class until its deletion) and the number of commits per time unit (e.g., one day, one week, one month) roughly follow a power-law distribution as well. These two studies provide empirical evidence that commits vary in size and frequency.

To make things even more complicated, commit practices are in turn influenced by a series of factors, including the project's development process, the way tasks are defined in Issue Tracking Systems or backlogs, and the specific version control system being used. For instance, recent studies have shown that the interaction protocol of version control systems influences commit frequency and the number of files in the change-sets. Brindescu et al. [53] conducted a large-scale empirical study (358k commits, 132 repositories, 5890 developers) and showed that commits made in distributed repositories were 32% smaller (fewer files) than in centralized repositories, and that developers split commits more often in distributed repositories.

All these studies leave us with several challenges. Should all commits be treated the same? What if the software development process does not enforce commit policies and developers end up having very different commit behaviors? How do we detect periods where commits have similar properties (size and frequency), so that the choice of thresholds for spotting relevant change coupling are meaningful and appropriate? What do we do when a project moves from a centralized version control system (e.g., SVN) to a distributed one (e.g., Git)? These are all challenges that call for further investigation. In the next section, we offer some hints to help detect change couplings in practice.

11.4.2 PRACTICAL ADVICE FOR CHANGE COUPLING DETECTION

In the following, we offer practical advice on how to collect and preprocess the input (commits) in order to extract file-based change couplings more accurately. We highlight that such recommendations are independent of the specific identification method chosen (Section 11.3), as they deal with the input only. These recommendations derive both from the aforementioned studies and from our own experience in the topic (lessons learned).

(1) Selecting subject projects

(a) Choose projects that use the same version control system. As we saw at the end of Section 11.4.1, the repository technology (e.g., centralized vs. distributed) influences commit frequency, as well as the average number of artifacts per change set. In order to reduce bias in empirical studies, we recommend selecting subject projects that use the same version control system product (e.g., SVN or Git). As a desirable consequence, this will also make the study's technological infrastructure lighter.

(b) Choose projects that link commits to tasks. Several open source projects, like Apache Lucene and Apache Hadoop, link commits to the tasks in the issue tracker (e.g., by explicitly mentioning the task id in the commit's comments). This adds contextual information to the commit and is beneficial for a series of purposes. For instance, having the task link enables you to calculate change couplings for specific types of software changes, such as bug fixes or new features. Most importantly, knowing the type of task also helps to conceive preprocessing mechanisms that improve the accuracy of change couplings (see advice item 4).

(c) Avoid projects that often move and migrate data. Most change coupling mining tools track files based on their paths. Although the tools are often able to track files that get renamed over time, they might face difficulties when tracking files that are moved to different paths. Cases in

which files are deleted and then readded to a different path are especially complicated to deal with. Therefore, prefer projects where the "trunk" folder (or the folder you are mining) does not move over time. In addition, prefer projects that do not migrate data from other repositories. For example, an SVN repository that synchronizes with a CVS repository. The reason is that these migrations might alter the way developers originally made the commits, possibly leading to new commits that either group unrelated tasks or split cohesive changes.

(d) Open source projects are at your disposal. Thanks to the open source movement, several version control systems are freely available (in the sense that anyone can "read" them). Several research studies include projects from the Apache Software Foundation, a non-profit organization that has developed nearly a hundred distinguishing software projects that cover a wide range of technologies and domains. Examples of Apache projects include Apache HTTP Server, Apache Geronimo, Cassandra, Lucene, Maven, Ant, Struts, and JMeter. All Apache projects are hosted under a single SVN repository at https://svn.apache.org, which currently stores more than 1.6 million commits. A Git mirror with all projects is also available at http://git.apache.org/. Other hubs such as SourceForge and GitHub also contain a plethora of open source projects.

(2) Manipulating the repositories

(a) Work locally. To avoid dealing with network instability and other problems, we recommend mirroring (replicating) the repository locally when possible. The following sample script shows how to mirror the SVN repository of the Apache JMeter project to a local path in the filesystem:

```
1) Use the svnadmin utility to create a new (empty) repository
in the local file system.
   $ svnadmin create /mirrors/jmeter
After running the command, a non-empty directory called ‘‘jmeter’’
is created. The results should look similar to the following:
total 54K
drwxrwxr-x+ 1 user None 0   Dec 02 17:01 .
drwxrwxrwt+ 1 user None 0   Dec 02 17:01 ..
drwxrwxr-x+ 1 user None 0   Dec 02 17:01 conf
drwxrwxr-x+ 1 user None 0   Dec 02 17:01 db
-r--r--r--  1 user None 2   Dec 02 17:01 format
drwxrwxr-x+ 1 user None 0   Dec 02 17:01 hooks
drwxrwxr-x+ 1 user None 0   Dec 02 17:01 locks
-rw-rw-r--  1 user None 246 Dec 02 17:01 README.txt


2) Create an empty file "jmeter/hooks/pre-revprop-change" with the
‘‘execute permission’’ set. If in Windows, add a .bat extension to the
file. For more details, please read the template file
"/jmeter/hooks/pre-revprop-change.tmpl"
   $ touch /mirrors/jmeter/hooks/pre-revprop-change
```

```
   $ chmod 777 /mirrors/jmeter/hooks/pre-revprop-change


3) Initialize the mirror repository using the "svnsync init" command.
This ties your local repository to the remote one
   $ svnsync init --username anonymous
   file:///mirrors/jmeter https://svn.apache.org/jmeter


4) Use the "svnsync sync" command to populate the mirror repository:
   $ svnsync sync file:///mirrors/jmeter
```

The sample script above should work for most scenarios, but you can tailor it for your own requirements. Detailed instructions for mirroring a SVN repository are included in the free book "Version Control with Subversion."[22] Since mirroring and interacting with the repository might induce a lot of I/O, we also suggest using a solid-state drive (SSD) when possible.

(3) Determining the analysis scope
If the project has a stable release cycle, then identifying change couplings from each release can be a good approach. If not, then it is often better to analyze commit chunks (sequence of contiguous commits). The number of commits per chunk depends on the particular study and on the total number of commits the project has.

(4) Preprocessing commits to avoid noise and improve accuracy
Zimmermann and Weißgerber [23] emphasized that cleaning data is an important part of improving change coupling detection. In their work, they tackle two issues: large commits (i.e., commits with many files) and merge commits. They consider large commits as noise because the files they comprise often refer to infrastructure changes. In other words, these commits are not the consequence of relevant connascence relations (check Section 11.2.1). The authors advise filtering out commits of a size greater than a certain N, where N is a threshold defined on a per project basis.

The authors also regard *merge commits* (Figure 11.7) as noise. There are two main reasons. First, merge commits often contain unrelated changes. For instance, let us assume that commit 20 addresses a certain issue and commit 22 addresses a different one. In this case, commit 24 would contain unrelated changes, since A and C originally changed for different reasons. Second, merge commits rank changes on branches higher. For instance, A and B appear in both commit 20 and commit 24 (same thing happens to C).

Zimmermann and Weißgerber [23] note that depending on the purpose of the analysis, these merge commits should be ignored or at least receive some special treatment. Fluri and Gall [5] argue that commits including code styling and minor adjustments are also not significantly relevant in the context of change coupling identification.

[22] http://svnbook.red-bean.com/en/1.7/svn.reposadmin.maint.html#svn.reposadmin.maint.replication.

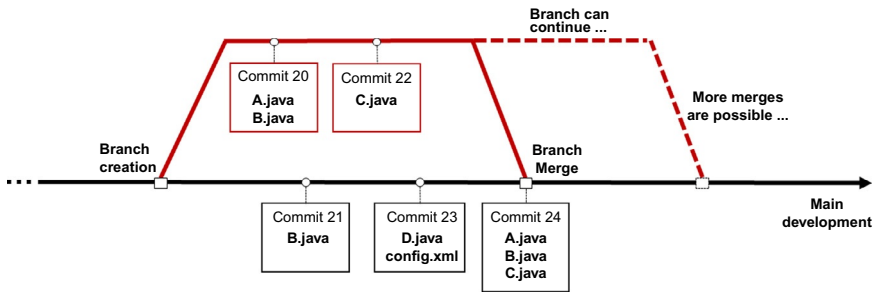

FIGURE 11.7

Influence of merges in change coupling identification.

In fact, the solution to these problems boils down to being able to classify or understand the kinds of changes implemented in each commit. Simply ignoring large commits might make you miss large refactorings or relevant changes. Our experience has shown that choosing projects that link commits to tasks in the issue track is often a better solution, since it helps to discover the purpose of each commit a lit bit better (see advice item 1b). Other strategies based on keyword matching against the commits' comments might also work reasonably well for certain applications (e.g., searching for "bug fix" or "refactoring"). This information can also be used to detect and bypass commits that link to two different tasks, thus mitigating the problem of overloaded commits. You may also leverage it to group commits that tackle the same tasks, thus mitigating the problem of incomplete commits. We note that Herzig and Zeller [50] performed a detailed manual classification of tasks found in the issue tracker of five open source systems: HTTPClient, Jackrabbit, Lucene, Rhino, and Tomcat 5. These data can be used to conceive preprocessing mechanisms that filter out unwanted commits, such as the ones that involve infrastructure changes, branch merging, and code styling.

11.4.3 ALTERNATIVE APPROACHES

The identification of change couplings from version control systems has one intrinsic shortcoming-*development information loss*. This problem was put in the spotlight by Robbes and colleagues [54, 55]. According to the author, versioned artifacts might undergo several changes in the period delimited by code checkout and commit. By analyzing such development session, one might conclude that although artifacts A, B, C, and D were modified, change couplings exist only between A and B, as well as between C and D. Moreover, it might be the case that A and B have a stronger change coupling when compared to the coupling level of C and D. However, the logs of version control systems will only store the set of modified files, the associated change operation (add, remove, replace, etc.), and the lines that changed. According to Robbes and colleagues [54, 55], this implies that "a large amount of data is needed before the measure can be accurate." They conclude that change coupling identification from periods of very active development (as opposed to projects in *maintenance mode*) may suffer even more from this issue.

Other researchers have an even stronger position. Negara et al. [56] consider that, although convenient, research based on version control systems is often incomplete and imprecise. They also note that many interesting research questions that involve code changes and other development activities (e.g., automated refactorings) require evolution data that is not captured by version control systems at all.

All these problems can be reduced to the fact that version control systems only store coarse-grained information. The alternative solution proposed by both Negara et al. and Robbes et al. relies on instrumenting developers' IDE. Quoting [56]:

> Code evolution research studies how the code is changed. So, it is natural to make changes be first-class citizens and leverage the capabilities of an Integrated Development Environment (IDE) to capture code changes online rather than trying to infer them post-mortem from the snapshots stored in VCSs (version control systems).

Negara et al. [56] developed an Eclipse plug-in called CodingTracker that unobtrusively collects fine-grained data about the code evolution of Java programs. This tool records every code edit performed by a developer, as well as other development actions, such as invocations of automated refactorings, tests and application runs, interaction with the version control system, etc. According to Negara and colleagues, the collected data is so precise that it enables them to reproduce the state of the underlying code at any point in time. To represent the raw code edits collected by CodingTracker uniformly and consistently, they implemented an algorithm that infers changes as Abstract Syntax Tree (AST) node operations. The solution from Robbes et al. [54, 55] is similar. They developed a tool called SpyWare that is notified by the Smalltalk compiler in the Squeak IDE whenever the AST of the underlying program changes. The solution from Negara and colleagues seems to be more complete and flexible, in the sense that their tool captures additional information (i.e., evolution data that does represent changes to code) and does not expect the underlying code to be compilable or even fully parsable.

What we see here is actually a trade-off. These studies show strong evidence that fine-grained information obtained from IDE monitoring is more accurate. However, they also have disadvantages. First, both approaches are targeted to specific IDEs: SpyWare interacts with Squeak IDE and CodingTracker is an Eclipse plug-in. Consequently, it can be that their software needs to be adjusted when new IDE versions are released (as often occurs with Eclipse plug-ins, for instance). Second, both approaches are targeted to specific programming languages: SpyWare records only Smalltalk code changes and CodingTracker records only Java code changes. What if the subject project is written in C#? What if the subject project is written mainly in Java, but also has a lot of XML files and other resources (which is common)? Third, although their tools seem unintrusive, they can only capture information from instrumented IDEs. In other words, all software development that occurred before the release of their tool is inevitably left behind. In particular, such period encompasses the core development of a huge amount of free/libre open source software (FLOSS) projects. In addition, given the highly collaborative and distributed nature of FLOSS development, instrumenting the IDEs of all developers becomes much more complicated (maybe even unfeasible in most cases). Therefore, while monitoring the IDE provides much more accurate data, it is also much more restrictive. In turn, research that mines version control systems only requires the existence of the log files (change-sets). Depending on the objective of the study, relying solely on these

logs is perfectly adequate, as acknowledged by both Negara et al. [56] and Robbes et al. [54, 55] themselves.

11.5 CHANGE COUPLING APPLICATIONS

Now that we have presented different change coupling identification approaches and some pieces of practical advice, we will switch the focus to what we can do with the mined couplings. In the following sections, we present some key applications for change coupling, which include change prediction and change impact analysis (Section 11.5.1), discovery of design flaws and opportunities for refactoring (Section 11.5.2), architecture evaluation (Section 11.5.3), and identification of coordination requirements (Section 11.5.4).

11.5.1 CHANGE PREDICTION AND CHANGE IMPACT ANALYSIS

Change impact analysis (or simply *impact analysis*) concerns "identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change" [57]. Preventing side effects and estimating ripple effects have been two commons uses of impact analysis [58]. In a more general sense, developers use change impact analysis information for planning changes, deciding changes, accommodating certain types of changes, and tracing the effects of changes [57]. For more information, the interested reader might refer to the seminal book *Software Change Impact Analysis* by Arnold [57].

Change impact analysis probably constitutes the main application of change couplings. The rationale behind it is that *entities that have changed together in the past are likely to change together in the future* [10]. Zimmermann et al. [3, 31] developed an Eclipse plug-in that captures change couplings from the CVS version control system. Inspired by the way large e-commerce websites (e.g., Amazon[23] and eBay[24]) suggest related products to their visitors, their tool informs about related software changes: "programmers that changed these functions also changed. . ." More specifically, right after a certain developer changes a piece of code, the tool suggests locations where, in similar transactions in the past, other changes were made. These recommended locations can be very specific, like a class attribute or a class method. The tool captures change couplings using association rules, which are mined "on-demand" using a modified version of the *Apriori algorithm* [29]. After obtaining the rules and calculating their respective values of *support* and *confidence*, the tool displays them to the end-user (sorted by confidence).

The main benefits of the tool are: (a) suggesting and predicting likely changes, (b) showing coupling between items that would not be detectable via static analysis, and (c) preventing errors resulting from incomplete changes [3, 31]. These benefits are especially helpful for newcomers joining a software project, since they are less acquainted with the software architecture and with the semantics of certain classes. The results from the evaluation conducted by Zimmermann and colleagues showed that their tool was helpful in suggesting further changes and in warning about missing changes. However, they

[23] http://www.amazon.com.
[24] http://www.ebay.com.

highlight that the more there is to learn from history, the more and better suggestions that can be made. More details about this work can be found in their papers [3, 31] and at: http://thomas-zimmermann.com/publications/details/zimmermann-tse-2005/.

11.5.1.1 Other research results

Almost in parallel with Zimmermann and colleagues, a different group of researchers investigated the very same problem [33]. However, instead of generating association rules with the *Apriori* algorithm, they employed the more efficient FP-Growth algorithm [34] to find frequent itemsets, which avoids the step of generating candidate itemsets and testing them against the entire database. Ying and colleagues recommended the set of files to be changed by taking the union of frequent itemsets (change patterns) that includes the file being currently modified by the developer.

Hassan and Holt [17] conceived four different change propagation heuristics. The first heuristic (DEV) returns all program-level entities previously changed by the same developer who is performing the current change. The second one (HIS) returns all entities previously changed together with the entity being modified. The third heuristic (CUD) returns all entities structurally related to the entity being modified. The last heuristic (FIL) returns all entities defined in the same file as the entity being modified. The authors evaluated the performance of these heuristics in five open source systems: NetBSD, FreeBSD, OpenBSD, Postgres, and GCC. The heuristic based on change couplings (HIS) had the best recall (0.87) and the second best precision (0.06). Their results cast doubts on the effectiveness of using structural dependencies alone for predicting change propagation. Four years later, Malik and Hassan [20] worked on an adaptive change propagation recommender that relies on both structural and historical information to provide better suggestions.

Kagdi et al. [59] presented an approach for change impact analysis based on the combination of conceptual coupling analysis and change couplings analysis. Information retrieval techniques are used to derive conceptual couplings from the source code of a specific version of the subject system (e.g., a release). As usual, the authors identify change couplings by mining association rules from the logs of version control systems. The authors conducted an empirical study with historical data from four open source projects, Apache httpd, ArgoUML, iBatis, and KOffice. The results showed that the combination of the two techniques provide statistically significant improvement in accuracy when compared to the use of either technique individually. More specifically, the authors obtained improvements of up to 20% over the use of conceptual coupling technique alone in KOffice and up to 45% over the technique of change couplings in iBatis.

11.5.2 DISCOVERY OF DESIGN FLAWS AND OPPORTUNITIES FOR REFACTORING

Change couplings reveal how the evolution of versioned artifacts intertwine. In particular, artifacts that are highly change coupled to many other artifacts are intrinsically problematic, since this implies that these artifacts are frequently affected by changes made to other parts of the system. High change coupling among modules generally points to design flaws or even to architectural decay.

In order to help developers understand how change coupled artifacts are, D'Ambros and colleagues introduced a change coupling visualization tool called Evolution Radar [2, 10, 60, 61, 77]. The Evolution Radar is interactive and integrates information about change coupling at the file level and at the module level (group of files) in a scalable way. Furthermore, it enables developers to study and inspect change couplings in an interactive way by guiding them to the files responsible for strong change

couplings (outliers). More specifically, the Evolution Radar helps to answer the following questions: (a) What are the components (e.g., modules) with the strongest (change) coupling? (b) Which low level entities (e.g., files) are responsible for these couplings?

Figure 11.8 shows the schematics of the Evolution Radar. The module chosen by the developer is visualized as a highlighted circle placed in the center of the radar. All other modules of the system are represented as sectors. The sector's size is proportional to the number of files it contains. Sectors are ordered according to their sizes, with the smallest one at 0 radian and the remaining ones arranged clockwise. Within each sector, files are represented as colored circles. Arbitrary metrics can be mapped to the color and size of file circles. Each circle is positioned according to polar coordinates, where the radius d and the angle θ are computed according to the following rules:

- Radius d (distance to the center): it is inversely proportional to the level of change coupling between the file (f) and the module (M). The more coupled they are, the closer they are to each other. In their study, Lanza and colleagues measured change coupling according to the following formula:

$$\mathrm{LC}(M,f) = \max_{f_i \in M} \mathrm{LC}(f_i,f), \text{ where}$$

$$\mathrm{LC}(f_i,f_j) = \text{number of that } f_i \text{ and } f_j \text{ changed together}$$

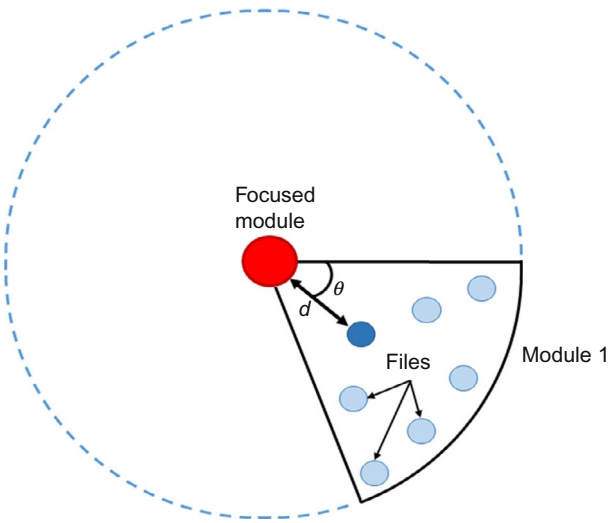

FIGURE 11.8

Schematics of Evolution Radar.

- Angle θ: the files in each module are ordered alphabetically considering their paths and uniformly distributed along the sector.

The main features of Evolution Radar are as follows [2]:

(a) Moving through time: When creating the radar, the end-user can divide the lifetime of the system into time intervals. For each of them a different radar is created (and change coupling is computed with respect to the given time interval). The radius coordinate has the same scale in all the radars, so that end-users can compare radars and analyze the evolution of coupling over time.

(b) Tracking: When a file is selected for tracking in a visualization related to a particular time interval, it is highlighted in all the radars (with respect to all the other time intervals) in which the file exists. This feature allows the end-user to keep track of files over time.

(c) Spawning: This feature enables end-users to discover how intensively files inside the module in focus are change coupled to other files of the system, thus providing a more detailed view of coupling.

In the following, we present an evaluation of the ArgoUML project done by D'Ambros and colleagues using the Evolution Radar [10]. Figure 11.9 depicts some of the radars they built for this evaluation. These radars focus on showing the change couplings between the Explorer module (the focused module in the center) and all other artifacts (all other circles) of the system for three consecutive analysis periods. A color temperature mapping is used: plain blue represents the lowest coupling and plain red represents the highest coupling.[25] The size of file circles is proportional to the total number of lines modified in all commits during the considered time interval.

The first radar (a) highlights a class named ModelFacade that underwent several modifications during the analysis period and that was highly coupled to the Explorer module. The authors further investigated the ModelFacade and discovered that it was a God class [62] with thousands of lines of code and around 450 methods (all static). The second radar (b) does not include the ModelFacade, i.e., a certain developer deleted it in the associated analysis period.

Using the tracking feature, the authors discovered that the NSUMLModelFacade class was the most coupled class in the second and third radars. In fact, its coupling with the Explorer module increased over time (its circle was getting closer to the central circle). A closer look revealed that the NSUMLModelFacade was also a God class with 317 public methods. The authors also discovered that more than 75% of its code was duplicated from the deleted ModelFacade class. Therefore, it appears the developers just relocated the problem instead of doing a proper refactoring. This example highlights how change couplings help detect design flaws and shows how artifacts (co)evolve over time. A more detailed evaluation of the ArgoUML project can be found in their journal article [10].

11.5.2.1 Other research results

Vanya et al. [63, 64] investigated whether interactive visualizations of co-changed software artifacts could be used beyond the mere identification of unwanted change couplings. More specifically, they investigated whether these techniques could help architects reason about and resolve these couplings. To evaluate their proposal, the authors conducted a case study in which they invited the architect and developers of a large medical system at Philips Healthcare to use iVIS to investigate unwanted change couplings in the system. The authors (i) selected the unwanted couplings the architect and developers

[25] Check the original paper for colored pictures [10].

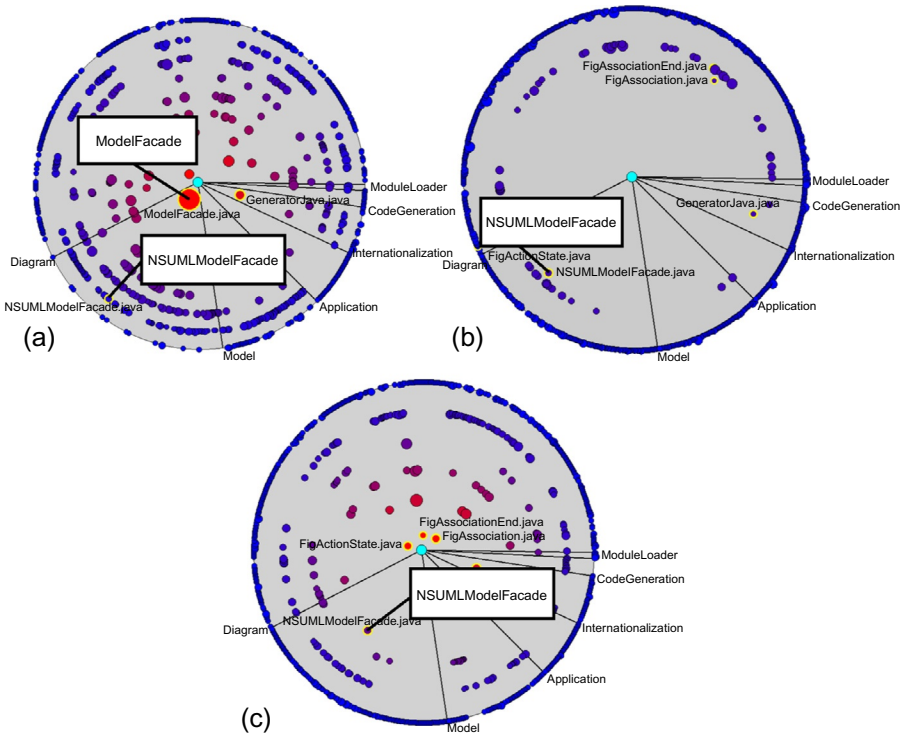

FIGURE 11.9

Evolution Radars for the ArgoUML.[26] (a) From June to December 2004; (b) From January to June 2005; (c) From June to December 2005.

decided to analyze, (ii) defined and implemented the interactions to be tested, and (iii) organized working sessions with the architect and developers to analyze the unwanted couplings. Solutions to unwanted couplings could be found in 7 out of the 10 working sessions conducted.

Beyer and Hassan [65, 66] introduced a visualization technique called *evolution storyboards*, which builds on a previous study by Beyer et al. [67] that clusters artifacts based on their change couplings. Just as directors and cinematographers use storyboards to study movie scenes and uncover potential problems before they occur, evolution storyboards were conceived to replay and study the history of software systems based on change coupling graphs. This is essentially an alternative to the Evolution Radar.

Ratzinger et al. [68] used change couplings to detect *bad smells*, which are somewhat subjective perceptions of design shortcomings. They developed a visualization tool called Evolens [68] that produces change coupling graphs, where large ellipses denote packages, smaller ellipses denote classes,

[26] Adapted from D'Ambros and Lanza [10, p. 6-7]. Content licensed by IEEE. Copyright cleared.

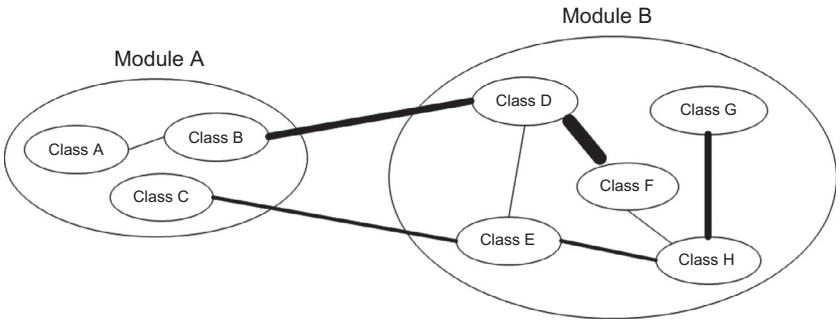

FIGURE 11.10

Schematic of the Evolens visualization.

and edges denote change couplings (Figure 11.10). The thickness of the edges is directly proportional to the number of co-changes involving the associated classes. The authors hypothesize that their tool assists developers in finding and fixing design flaws via refactoring. The authors introduced two *change smells*, i.e., *man-in-the-middle* and *data container*. Man-in-the-middle refers to a central class that is change coupled to many others scattered over several modules of the system. In turn, the data container smell involves two classes: one that holds the data and another that interacts with other classes of the system that require the data from the first class. The authors analyzed the history of a large industrial system for 15 months and found occurrences on both smells.

11.5.3 ARCHITECTURE EVALUATION

Zimmermann et al. [25] investigated the extent to which change history can be used to improve the assessment of software architectures. To this end, they detected change couplings between program-level entities (i.e., attributes and methods) and assessed the modularity of several open source projects, including GCC, DDD, Python, Apache, and OpenSSL. Such an assessment was driven by an analysis of the values produced by two metrics. One metric was the Evolutionary Density Index (EDI), which relates the number of actual change couplings to that of possible change couplings. The lower the EDI, the better the modularity. The other metric was the Evolutionary Coupling Index (ECI), which relates the actual number of external change couplings (i.e., couplings between entities defined in different files) to the actual number of internal change couplings. As in the previous case, the lower the ECI, the better the modularity. From their results, they concluded that a change history can either justify the organization and principles of the system architecture or show where reality diverges from policy (e.g., architectural rules).

Recently, Silva et al. [69] used change couplings to assess the modularity of software systems. The rationale comes from the principle that modules should confine implementation decisions that are likely to change together [70]. This is also known as the Common Closure Principle [71]. Silva and colleagues created co-change graphs, with edges representing the number of common changes between artifacts. They applied a clustering algorithm to extract co-change clusters from the graph and then compared

such clusters against the hierarchical (package) structure of the system. Their evaluation included three open source systems: Geronimo, Lucene, and JDT Core. The authors performed this comparison using distribution maps [72], which is a visualization technique they leveraged to depict how clustered classes are distributed over the system packages. During the evaluation of the systems, they were able to see several of the distribution patterns introduced by Ducasse et al. [72]. For instance, some co-change clusters fit the *Octopus pattern*, since they were well encapsulated in one package (the "body") but also spread across others (the "tentacles").

11.5.4 COORDINATION REQUIREMENTS AND SOCIO-TECHNICAL CONGRUENCE

Organizations often cope with complex tasks by first dividing them into smaller interdependent work units and then assigning these units to teams. In this context, coordination among teams arises as a response to such interdependent work units [73]. Cataldo and colleagues conceived an approach to elicit coordination requirements [19, 74, 75]. More specifically, their approach tackles the following problem: *given a particular set of dependencies among tasks, identify which set of individuals should coordinate their activities*.

The approach from Cataldo et al. relies on two sets of relationships (Figure 11.11). The first set is called *Task Assignments* (T_A) and defines which individuals are working on which tasks. This set is represented by a matrix where each cell $[i, j]$ indicates that the developer i was assigned to the task j. In the context of software development, this set might be built upon the set of files modified by each developer on a modification request or throughout the development of a software release. The second set of relationships is called *Task Dependencies* (T_D) and defines the interdependencies between tasks. This set is also represented by a matrix where each cell $[i, j]$ (or $[j, i]$) indicates whether tasks i and j are interdependent. In the context of software development, this set might be built according to either structural coupling or change couplings. Cataldo and colleagues tested the two alternatives and concluded that change couplings provided better results [74]. In the particular case of change coupling, off-diagonal cells of T_D indicate the number of times the two files were changed together. In turn, the main diagonal indicates the total number of times the source code files were changed.

Once T_A and T_D matrices are built, coordination requirements are ready to be determined. Multiplying T_A by T_D results in a "people by task" matrix that represents the extent to which a particular

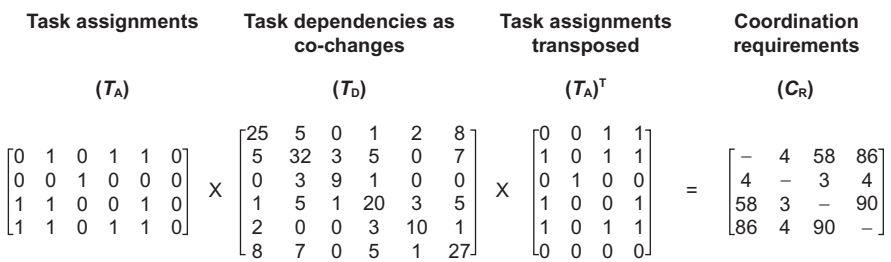

FIGURE 11.11

Illustrative example of coordination requirements calculation.

worker should be aware of tasks that are interdependent to those that he or she is responsible for [74]. Multiplying the $T_A \times T_D$ product by the transpose of T_A results in a people by people matrix where a cell $[i, j]$ represents the extent to which person i works on tasks that share dependencies with the tasks worked on by person j [74]. In other words, this last matrix represents the Coordination Requirements (C_R), or the extent to which each pair of people needs to coordinate their work (values in the main diagonal should be ignored). When calculating T_D using co-changes, the resulting C_R matrix is symmetric (Figure 11.11).

11.6 CONCLUSION

In this chapter, we provided an overview of change coupling to researchers and practitioners. Our goals were to explain the concept (Section 11.2), present the main identification approaches (Section 11.3), discuss current challenges in coupling identification and offer some practical advice (Section 11.4), and present the main application areas (Section 11.5). To us, there is no doubt that detecting and analyzing change couplings is becoming an increasingly useful tool in software engineering empirical studies. Several studies have been published at top conferences, such as those on mining software repositories [12, 23], software engineering [2, 76], software evolution [25], and reverse engineering [10]. The work of Zimmermann et al. [31] on change prediction (Section 11.5.1) won the most influential paper award at the 26th International Conference on Software Engineering (ICSE 2014), the world's most important conference in software engineering. Having said that, we sincerely hope this chapter has given you the fundamentals to detect and analyze change couplings in practice. The authors of this chapter will be glad to answer questions and discuss researches revolving around the topic.

REFERENCES

[1] Ball T, Adam JMK, Harvey AP, Siy P. If your version control system could talk... In: ICSE workshop on process modeling and empirical studies of software engineering; 1997.

[2] D'Ambros M, Lanza M, Lungu M. Visualizing co-change information with the evolution radar. IEEE Trans Softw Eng 2009;35(5):720–35. doi:10.1109/TSE.2009.17.

[3] Zimmermann T, Weissgerber P, Diehl S, Zeller A. Mining version histories to guide software changes. IEEE Trans Softw Eng 2005;31(6):429–45. doi:10.1109/TSE.2005.72.

[4] Fluri B, Gall HC, Pinzger M. Fine-grained analysis of change couplings. In: Proceedings of the fifth IEEE international workshop on source code analysis and manipulation; 2005. p. 66–74. doi:10.1109/SCAM.2005.14.

[5] Fluri B, Gall HC. Classifying change types for qualifying change couplings. In: Proceedings of the 14th IEEE international conference on program comprehension, ICPC 2006; 2006. p. 35–45. doi:10.1109/ICPC.2006.16.

[6] D'Ambros M, Gall H, Lanza M, Pinzger M. Analysing software repositories to understand software evolution. In: Mens T, Demeyer S, editors. Software evolution. Berlin: Springer; 2008. p. 37–67. Retrieved from doi:10.1007/978-3-540-76440-3.

[7] Mens T, Demeyer S. Software evolution. 1st ed. Berlin: Springer Publishing Company, Inc.; 2008.

[8] D'Ambros M, Lanza M, Robbes R. On the relationship between change coupling and software defects. Los Alamitos, CA, USA: IEEE Computer Society; 2009. p. 135–44. doi:10.1109/WCRE.2009.19.

[9] Zhou Y, Wursch M, Giger E, Gall H, Lu J. A Bayesian network based approach for change coupling prediction. In: Proceedings of the 15th working conference on reverse engineering, WCRE'08; 2008. p. 27–36. doi:10.1109/WCRE.2008.39.
[10] D'Ambros M, Lanza M. Reverse engineering with logical coupling. In: 13th working conference on reverse engineering, WCRE '06; 2006. p. 189–198. doi:10.1109/WCRE.2006.51.
[11] Gall H, Hajek K, Jazayeri M. Detection of logical coupling based on product release history. In: Proceedings of the international conference on software maintenance, ICSM '98. Washington, DC, USA: IEEE Computer Society; 1998. p. 190. Retrieved from http://dl.acm.org/citation.cfm?id=850947.853338.
[12] Alali A, Bartman B, Newman CD, Maletic JI. A preliminary investigation of using age and distance measures in the detection of evolutionary couplings. In: Proceedings of the 10th working conference on mining software repositories, MSR '13. San Francisco, CA, USA: IEEE Press; 2013. p. 169–72. Retrieved from http://dl.acm.org/citation.cfm?id=2487085.2487120.
[13] Hassan AE. The road ahead for mining software repositories. Front Softw Maint 2008; 2008:48–57. doi:10.1109/FOSM.2008.4659248.
[14] Page-Jones M. Comparing techniques by means of encapsulation and connascence. Commun ACM 1992;35(9):147–51. doi:10.1145/130994.131004.
[15] Page-Jones M. Fundamentals of object-oriented design in UML. 1st ed. Reading, MA: Addison-Wesley; 1999.
[16] McIntosh S, Adams B, Nguyen THD, Kamei Y, Hassan AE. An empirical study of build maintenance effort. In: Proceedings of the 33rd international conference on software engineering, ICSE '11. Waikiki, Honolulu, HI, USA: ACM; 2011. p. 141–50. doi:10.1145/1985793.1985813.
[17] Hassan AE, Holt RC. Predicting change propagation in software systems. In: Proceedings of the 20th IEEE international conference on software maintenance, ICSM '04. Washington, DC, USA: IEEE Computer Society; 2004. p. 284–93. Retrieved from http://dl.acm.org/citation.cfm?id=1018431.1021436.
[18] Hassan AE, Holt RC. Replaying development history to assess the effectiveness of change propagation tools. Empir Softw Eng 2006;11(3):335–67. doi:10.1007/s10664-006-9006-4.
[19] Cataldo M, Herbsleb JD. Coordination breakdowns and their impact on development productivity and software failures. IEEE Trans Softw Eng 2013;39(3):343–60. doi:10.1109/TSE.2012.32.
[20] Malik H, Hassan AE. Supporting software evolution using adaptive change propagation heuristics. In: Proceedings of the IEEE international conference on software maintenance, ICSM, 2008; 2008. p. 177–86. doi:10.1109/ICSM.2008.4658066.
[21] Cataldo M, Mockus A, Roberts JA, Herbsleb JD. Software dependencies, work dependencies, and their impact on failures. IEEE Trans Softw Eng 2009;35(6):864–78. doi:10.1109/TSE.2009.42.
[22] Cataldo M, Nambiar S. The impact of geographic distribution and the nature of technical coupling on the quality of global software development projects. J Softw Maint Evol Res Pract 2010. doi:10.1002/smr.477.
[23] Zimmermann T, Weißgerber P. Preprocessing CVS data for fine-grained analysis. In: Proceedings 1st international workshop on mining software repositories (MSR 2004). Los Alamitos, CA: IEEE Computer Society Press; 2004. p. 2–6.
[24] Gall H, Jazayeri M, Krajewski J. CVS release history data for detecting logical couplings. In: Proceedings of the 6th international workshop on principles of software evolution. Washington, DC, USA: IEEE Computer Society; 2003. p. 13. Retrieved from http://dl.acm.org/citation.cfm?id=942803.943741.
[25] Zimmermann T, Diehl S, Zeller A. How history justifies system architecture (or not). In: Proceedings of the sixth international workshop on principles of software evolution; 2003. p. 73–83. doi:10.1109/IWPSE.2003.1231213.
[26] Oliva GA, Gerosa MA. On the interplay between structural and logical dependencies in open-source software. In: Proceedings of the 25th Brazilian symposium on software engineering, SBES'11. Washington, DC, USA: IEEE Computer Society; 2011. p. 144–53. doi:10.1109/SBES.2011.39.

[27] Rajaraman A, Ullman JD, Leskovec J. Mining of massive datasets. 2nd ed. 2013.
[28] Liu B. Web data mining: exploring hyperlinks, contents and usage data. 2nd ed. Berlin: Springer Publishing Company, Inc.; 2011.
[29] Agrawal R, Srikant R. Fast algorithms for mining association rules in large databases. In: Proceedings of the 20th international conference on very large data bases, VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 1994. p. 487–99. Retrieved from http://dl.acm.org/citation.cfm?id=645920.672836.
[30] Bavota G, Dit B, Oliveto R, Di Penta M, Poshyvanyk D, De Lucia A. An empirical study on the developers' perception of software coupling. In: Proceedings of the 2013 international conference on software engineering, ICSE '13. San Francisco, CA, USA: IEEE Press; 2013. p. 692–701. Retrieved from http://dl.acm.org/citation.cfm?id=2486788.2486879.
[31] Zimmermann T, Weissgerber P, Diehl S, Zeller A. Mining version histories to guide software changes. In: Proceedings of the 26th international conference on software engineering, ICSE'04. Washington, DC, USA: IEEE Computer Society; 2004. p. 563–72. Retrieved from http://dl.acm.org/citation.cfm?id=998675.999460.
[32] Wang X, Wang H, Liu C. Predicting co-changed software entities in the context of software evolution. In: Proceedings of the international conference on information engineering and computer science, ICIECS; 2009. p. 1–5. doi:10.1109/ICIECS.2009.5364521.
[33] Ying ATT, Murphy GC, Ng R, Chu-Carroll MC. Predicting source code changes by mining change history. IEEE Trans Softw Eng 2004;30(9):574–86. doi:10.1109/TSE.2004.52.
[34] Han J, Pei J, Yin Y. Mining frequent patterns without candidate generation. In: Proceedings of the 2000 ACM SIGMOD international conference on management of data, SIGMOD'00. Dallas, TX, USA: ACM; 2000. p. 1–12. doi:10.1145/342009.335372.
[35] Brin S, Motwani R, Ullman JD, Tsur S. Dynamic itemset counting and implication rules for market basket data. In: Proceedings of the 1997 ACM SIGMOD international conference on management of data, SIGMOD '97. Tucson, AZ, USA: ACM; 1997. p. 255–64. doi:10.1145/253260.253325.
[36] Kagdi H, Collard ML, Maletic JI. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. J Softw Maint Evol 2007;19(2):77–131. doi:10.1002/smr.344.
[37] Canfora G, Ceccarelli M, Cerulo L, Di Penta M. Using multivariate time series and association rules to detect logical change coupling: an empirical study. In: Proceedings of the IEEE international conference on software maintenance (ICSM); 2010. p. 1–10. doi:10.1109/ICSM.2010.5609732.
[38] Kruskal JB, Liberman M. The symmetric time-warping problem: from continuous to discrete. In: Sankoff D, Kruskal JB, editors. Time warps, string edits, and macromolecules—the theory and practice of sequence comparison. Palo Alto, CA: CSLI Publications; 1999.
[39] Antoniol G, Rollo VF, Venturi G. Detecting groups of co-changing files in CVS repositories. In: Proceedings of the eighth international workshop on principles of software evolution; 2005. p. 23–32. doi:10.1109/IWPSE.2005.11.
[40] Rabiner L, Rosenberg AE, Levinson SE. Considerations in dynamic time warping algorithms for discrete word recognition. IEEE Trans Acoust Speech Signal Process 1978;26(6):575–82. doi:10.1109/TASSP.1978.1163164.
[41] Rabiner L, Juang BH. Fundamentals of speech recognition. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.; 1993.
[42] Caiani EG, Porta A, Baselli G, Turiel M, Muzzupappa S, Pieruzzi F, et al. Warped-average template technique to track on a cycle-by-cycle basis the cardiac filling phases on left ventricular volume. Comput Cardiol 1998;1998:73–6. doi:10.1109/CIC.1998.731723.
[43] Oates T, Schmill MD, Cohen PR. A method for clustering the experiences of a mobile robot that accords with human judgments. In: Proceedings of the seventeenth national conference on artificial intelligence

and twelfth conference on innovative applications of artificial intelligence. Austin, TX: AAAI Press; 2000. p. 846–51. URL: http://dl.acm.org/citation.cfm?id=647288.721117.

[44] Rath TM, Manmatha R. Word image matching using dynamic time warping. In: Proceedings of the IEEE computer society conference on computer vision and pattern recognition, vol. 2; 2003. p. II-521–II-527. doi:10.1109/CVPR.2003.1211511.

[45] Müller M. Dynamic time warping. In: Information retrieval for music and motion. Berlin/Heidelberg: Springer; 2007. p. 69–84. doi:10.1007/978-3-540-74048-3{_}4.

[46] Bouktif S, Gueheneuc YG, Antoniol G. Extracting change-patterns from CVS repositories. In: Proceedings of the 13th working conference on reverse engineering, WCRE '06. Washington, DC, USA: IEEE Computer Society; 2006. p. 221–30. doi:10.1109/WCRE.2006.27.

[47] Salvador S, Chan P. Toward accurate dynamic time warping in linear time and space. Intell Data Anal 2007;11(5):561–80. URL: http://dl.acm.org/citation.cfm?id=1367985.1367993.

[48] Granger CWJ. Investigating causal relations by econometric models and cross-spectral methods. Econometrica 1969;37(3):424–38. doi:10.2307/1912791.

[49] Ceccarelli M, Cerulo L, Canfora G, Di Penta M. An eclectic approach for change impact analysis. In: Proceedings of the 32Nd ACM/IEEE international conference on software engineering, ICSE '10, vol. 2. Cape Town, South Africa: ACM; 2010. p. 163–6. doi:10.1145/1810295.1810320.

[50] Herzig K, Zeller A. The impact of tangled code changes. In: Proceedings of the 10th working conference on mining software repositories, MSR '13. San Francisco, CA, USA: IEEE Press; 2013. p. 121–30. Retrieved from http://dl.acm.org/citation.cfm?id=2487085.2487113.

[51] Ma Y, Wu Y, Xu Y. Dynamics of open-source software developer's commit behavior: an empirical investigation of subversion; 2013. CoRR, abs/1309.0897.

[52] Lin S, Ma Y, Chen J. Empirical evidence on developer's commit activity for open-source software projects. In: Proceedings of the 25th international conference on software engineering and knowledge engineering, SEKE'13, Boston, USA; 2013. p. 455–60. Retrieved from http://dl.acm.org/citation.cfm?id=257734.257788.

[53] Brindescu C, Codoban M, Shmarkatiuk S, Dig D. How do centralized and distributed version control systems impact software changes? (No. 1957/44927). EECS School at Oregon State University; 2014.

[54] Robbes R, Pollet D, Lanza M. Logical coupling based on fine-grained change information. In: Proceedings of the 15th working conference on reverse engineering, WCRE'08. Washington, DC, USA: IEEE Computer Society; 2008. p. 42–6. doi:10.1109/WCRE.2008.47.

[55] Robbes R. Of change and software. University of Lugano; 2008.

[56] Negara S, Vakilian M, Chen N, Johnson RE, Dig D. Is it dangerous to use version control histories to study source code evolution? In: Proceedings of the 26th European conference on object-oriented programming, ECOOP'12. Beijing, China: Springer-Verlag; 2012. p. 79–103. doi:10.1007/978-3-642-31057-7{_}5.

[57] Arnold RS. Software change impact analysis. Los Alamitos, CA, USA: IEEE Computer Society Press; 1996.

[58] Kagdi H, Maletic JI. Software-change prediction: estimated+actual. In: Proceedings of the second international IEEE workshop on software evolvability, SE'06; 2006. p. 38–43. doi:10.1109/SOFTWARE-EVOLVABILITY.2006.14.

[59] Kagdi H, Gethers M, Poshyvanyk D, Collard ML. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In: Proceedings of the 17th working conference on reverse engineering (WCRE); 2010. p. 119–28. doi:10.1109/WCRE.2010.21.

[60] D'Ambros M, Lanza M, Lungu M. The evolution radar: visualizing integrated logical coupling information. In: Proceedings of the 2006 international workshop on mining software repositories, MSR '06. Shanghai, China: ACM; 2006. p. 26–32. doi:10.1145/1137983.1137992.

[61] D'Ambros M, Lanza M. Distributed and collaborative software evolution analysis with churrasco. Sci Comput Program 2010;75(4):276–87. doi:10.1016/j.scico.2009.07.005.

[62] Fowler M. Refactoring: improving the design of existing code. Boston, MA: Addison-Wesley; 1999. Object Technology Series.
[63] Vanya A, Premraj R, Vliet H. Interactive exploration of co-evolving software entities. In: Proceedings of the 14th European conference on software maintenance and reengineering, CSMR'10. Washington, DC, USA: IEEE Computer Society; 2010. p. 260–3. doi:10.1109/CSMR.2010.50.
[64] Vanya A, Premraj R, Vliet H. Resolving unwanted couplings through interactive exploration of co-evolving software entities—an experience report. Inf Softw Technol 2012; 54(4):347–59. doi: 10.1016/j.infsof.2011.11.003.
[65] Beyer D, Hassan AE. Animated visualization of software history using evolution storyboards. In: Proceedings of the 13th working conference on reverse engineering, WCRE '06. Washington, DC, USA: IEEE Computer Society; 2006; pp. 199–210. doi:10.1109/WCRE.2006.14.
[66] Beyer D, Hassan AE. Evolution storyboards: visualization of software structure dynamics. In: Proceedings of the 14th IEEE international conference on program comprehension, ICPC '06. Washington, DC, USA: IEEE Computer Society; 2006; pp. 248–51. doi:10.1109/ICPC.2006.21.
[67] Beyer D, Noack A. Clustering software artifacts based on frequent common changes. In: Proceedings of the 13th international workshop on program comprehension. Washington, DC, USA: IEEE Computer Society; 2005. p. 259–68. doi:10.1109/WPC.2005.12.
[68] Ratzinger J, Fischer M, Gall H. Improving evolvability through refactoring. In: Proceedings of the 2005 international workshop on mining software repositories, MSR'05. St. Louis, MO: ACM; 2005. p. 1–5. doi: 10.1145/1082983.1083155.
[69] Silva L, Valente MT, Maia M. Assessing modularity using co-change clusters. In: Proceedings of the 13th international conference on modularity; 2014. p. 1–12.
[70] Parnas DL. On the criteria to be used in decomposing systems into modules. Commun ACM 1972; 15(12):1053–8. doi:10.1145/361598.361623.
[71] Martin RC, Martin M. Agile principles, patterns, and practices in C#. 1st ed. Upper Saddle River, NJ: Prentice Hall; 2006.
[72] Ducasse S, Girba T, Kuhn A. Distribution map. In: Proceedings of the 22nd IEEE international conference on software maintenance, ICSM '06. Washington, DC, USA: IEEE Computer Society; 2006. p. 203–12. doi:10.1109/ICSM.2006.22.
[73] March JG, Simon HA. Organizations. 2nd ed. New York: Wiley-Blackwell; 1993.
[74] Cataldo M, Herbsleb JD, Carley KM. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In: Proceedings of the second ACM-IEEE international symposium on empirical software engineering and measurement, ESEM '08. Kaiserslautern, Germany: ACM; 2008. p. 2–11. doi:10.1145/1414004.1414008.
[75] Cataldo M, Wagstrom P, Herbsleb JD, Carley KM. Identification of coordination requirements: implications for the design of collaboration and awareness tools. In: Hinds PJ, Martin D, editors, Proceedings of the 2006 ACM conference on computer supported cooperative work, CSCW 2006, Banff, Alberta, Canada, November 4-8. New York, NY, USA: ACM; 2006. p. 353–62. doi:10.1145/1180875.1180929.
[76] Kouroshfar E. Studying the effect of co-change dispersion on software quality. In: Proceedings of the 2013 international conference on software engineering, ICSE'13. San Francisco, CA, USA: IEEE Press; 2013. p. 1450–2. Retrieved from. http://dl.acm.org/citation.cfm?id=2486788.2487034.
[77] D'Ambros M, Lanza M. A flexible framework to support collaborative software evolution analysis. In: Proceedings of the 12th European conference on software maintenance and reengineering, CSMR '08. Washington, DC, USA: IEEE Computer Society; 2008. p. 3–12. doi:10.1109/CSMR.2008.4493295.