Faculty of Automation and Computers

Bachelor Program: Computers and Information Technology

DETECTING ANTI-PATTERNS BY ANALYZING THE EVOLUTION OF SOFTWARE SYSTEMS

Bachelor Thesis

Andrei HEIDELBACHER

Supervisor:
Prof. dr. eng. Radu MARINESCU

Timișoara,
2018

Abstract

The quality of software design has a strong impact on the flexibility and maintainability of a system, which enable it to evolve and adapt to the requirements of clients. Therefore, assessing the design quality is essential to guide the maintenance efforts. Towards this goal, we developed ChronoLens, a tool that extracts the history stored in version control systems and enables the analysis of software evolution. On top of the tool, we built three specialized analyzers, each detecting a specific anti-pattern: *Encapsulation Breakers*, *Open-Closed Breakers*, and *Single Responsibility Breakers*. We performed a study on several large open-source systems and noted some patterns: modules tend to decapsulate many fields at once, sometimes to allow inheritance; a few methods account for most of the extension and modification effort, indicating a lack of abstraction; overloaded methods and related test methods usually have a tight temporal coupling, but large sets of methods with a weak temporal coupling with the rest of the module to which they belong are also frequent. Thus, assessing software evolution can offer important insights on the quality of a design. Moreover, our tool can be used to develop new software evolution analyses.

Contents

List of Figures

List of Tables

Listings

1 Introduction. Problem Statement

1.1 Motivation

Software that does not change and adapt to the requirements of the client becomes obsolete. Flexibility and maintainability are two essential characteristics that allow software systems to evolve at a fast pace, to have defects fixed and new features added. The design quality of a system has a strong impact on these characteristics. Thus, software quality assurance aims to assess and improve the quality of software in general, and that of design in particular.

Manually assessing the quality of a design is infeasible, considering the complex systems of today. Consequently, many automated static analyzers have been developed in an attempt to measure and quantify software quality. Although the syntactic structure of software can be used to compute metrics that estimate quality, the evolution itself could provide deeper insights on the quality of a design.

Version control systems are tools devised to aid in the process of software development. They enable multiple programmers to work on the same piece of code concurrently, allow tracking the performed changes, managing release versions, etc. Although they were not designed with this goal in mind, the tracked history provides essential information that allows reconstituting the evolution of software systems. As version control systems became ubiquitous in the industry, analyzing and understanding software evolution with the help of automated tools became feasible.

Our goal is to analyze the information stored in version control systems and to detect design flaws specific to the evolution of software systems.

1.2 Contribution

In order to achieve our goal, we develop an application consisting of a tool named ChronoLens that extracts the history of a system and specialized analyzers that inspect the extracted history to detect design flaws.

The tool uses version control systems and parsers to extract the history into a language-independent model that integrates changes applied to software entities, allowing the analyzers to focus on what changed from one version to another and on the evolution itself.

On top of the tool, we devise three analyses:

- *Encapsulation Breakers* — detects software modules that break encapsulation, attempting to overcome the limitations of simple static analyzers

- *Open-Closed Breakers* — detects software modules that are extended through modification, violating the Open-Closed Principle, and indicates a lack of abstraction in the highlighted modules
- *Single Responsibility Breakers* — detects software modules that fulfill multiple responsibilities, violating the Single Responsibility Principle, and hints at how the highlighted modules could be split up in multiple, more cohesive modules

Our application lays the foundation of a solid tool that enables many software evolution analyses. The proof-of-concept analyses show how the tool can be used in this sense, as well as what type of insights can be inferred about the design of software systems.

1.3 Organization

We continue by presenting the fundamental concepts required to understand software quality and evolution. Therefore, in Chapter 2, we describe software design principles and anti-patterns, programming idioms used in the implementation of our application, as well as an overview of how version control systems work and what information they store.

Afterwards, in Chapter 3, we present the state of the art in the field of software evolution analysis, and compare our work with existing approaches.

Then, we describe the implementation of our history analysis tool, the underlying infrastructure of our anti-pattern detectors. Therefore, in Chapter 4, we present the architectural overview of our application, the significant design decisions we made and their trade-offs, as well as the models and algorithms we used to extract the history of software systems.

In Chapter 5, we arrive at the key parts of our application: the anti-pattern detectors. We present the metrics used to detect design flaws, the reasoning that led us to the chosen metrics, and the detection algorithms used to implement the detectors.

We begin Chapter 6 by illustrating the command-line interface of our tool and the visualization techniques we used to better understand the results of our analyses. Then, we continue with the experimental results obtained by analyzing several software systems, and note the patterns we observed in the results.

We conclude with Chapter 7 by summarizing our contributions and results and proposing a few directions to continue our work.

2 Theoretical Foundation

2.1 Object-Oriented Design

Object-oriented programming is a paradigm that facilitates software development using the fundamental concept of **object**, an aggregation of both *data* and *behavior* that operates on the data.

An object exposes a set of operations that comprise its **interface**. A **type** is a name given to a specific interface that multiple objects can share (those objects belong to that specific type). Clients can send requests to objects to execute certain operations. This is the only mechanism by which code is being run in the object-oriented paradigm.

Many object-oriented languages use **classes** as a construct to model types and to define the data (fields) and the implementation of the operations (methods) provided by categories of objects, whereas objects are **instances** of classes.

Encapsulation is the mechanism by which fields and methods are grouped together into a single entity, whereas **information hiding** is the mechanism by which access to certain components of an object is restricted. Sometimes, the two concepts are used interchangeably, or the meaning of encapsulation includes information hiding. These two concepts enable the compartmentalization of data and behavior into classes and objects, as well as abstraction by hiding implementation details from clients. Thus, objects become self-contained components that expose a public interface, bearing the responsibility of managing their internal state and making such changes invisible to clients.

Subtyping is a form of **polymorphism** in which objects belonging to a subtype can be substituted for objects belonging to a supertype. The interface provided by a subtype includes the interface provided by the supertype. In the context of object-oriented programmign, polymorphism is achieved through *inheritance* and *dynamic dispatch*.

Inheritance is a code reuse mechanism that implements the *is-a* relationship: a subclass that extends a superclass includes the fields and methods defined in the superclass. Inheritance is more than just subtyping: an object belonging to the subclass will inherit not only the interface (type) of the superclass, but the implementation as well. **Composition** is an orthogonal code reuse mechanism that implements the *has-a* relationship: an object can contain other objects as fields and use the provided operations to implement its own methods.

Dynamic dispatch is the process of selecting the implementation of a method at runtime based on the type of the object on which the method is called. This enables subclasses to specialize the implementations of methods in superclasses.

Lehman's *Law of Continuing Change* states that "a program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful" [1]. As a result, many heuristics and principles were developed to guide software design towards accommodating change.

Gamma et al. proposed two heuristics to achieve a better object-oriented design: "program to an interface, not an implementation" and "favor object composition over class inheritance" [2]. This way, clients depend on interfaces and do not know or care about the actual implementations (classes) provided at runtime, and complex objects are implemented using the interfaces provided by other objects instead of depending on the internal state and behavior of a superclass. In his book, *Agile Software Development: Principles, Patterns, and Practices* [3], Robert C. Martin presents several object-oriented design principles, which later became known as the *SOLID* principles:

Single Responsibility Principle
"A class should have only one reason to change."

Open-Closed Principle
"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification" (originally stated by Meyer [4]).

Liskov Substitution Principle
"Subtypes must be substitutable for their base types" (originally stated by Liskov [5]).

Interface Segregation Principle
"Clients should not be forced to depend on methods that they do not use."

Dependency Inversion Principle

a. "High-level modules should not depend on low level modules. Both should depend on abstractions."
b. "Abstractions should not depend on details. Details should depend on abstractions."

Even with all the design guidelines, software deteriorates according to Lehman's *Law of Increasing Complexity*: "as an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it" [1]. But how exactly does software deteriorate, and what can be done to maintain or reduce software complexity?

In *Refactoring: Improving the Design of Existing Code* [6], Fowler et al. describe **refactorings** — methods to improve the quality of existing code — and **code smells** — structures and patterns in code that indicate low quality and that could be improved through refactoring. Among the code smells they describe, three of them are of particular interest to us:

Divergent Change
"One class is commonly changed in different ways for different reasons."

Shotgun Surgery
"Every time you make a kind of change, you have to make a lot of little changes to a lot of different classes."

Feature Envy
"A method that seems more interested in a class other than the one it actually is in."

2.2 Design Patterns

Every software system is unique and solves a different problem, but certain tasks or goals might be common to multiple systems. While these tasks might not be general enough to be solved by a single piece of reusable code, the designs of successful solutions might be reusable.

Thus, we arrive at the concept of **design patterns**, defined by Gamma et al. as follows: "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" [2]. They built a catalog of design patterns they encountered in successful systems and described the applicable context, advantages and disadvantages. From that catalog, we present those patterns that are useful for our application.

2.2.1 Abstract Factory

Abstract Factory is a creational pattern used to "provide an interface for creating families of related or dependent objects without specifying their concrete classes" [2]. Figure 2.1 presents the structure of this pattern. `AbstractFactory` is an interface that provides creation methods for `AbstractProduct` objects, whereas `ConcreteFactory` is an implementation that handles the creation of concrete `Product` objects. Thus, clients interact only with abstractions and do not depend on particular implementations.

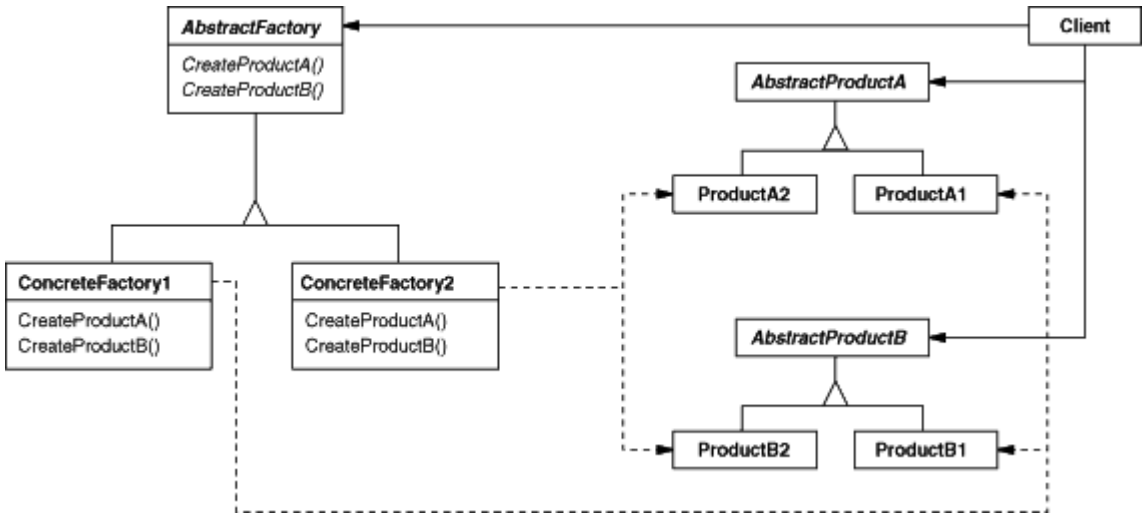


Figure 2.1: UML class diagram of the Abstract Factory pattern structure [2]

2.2.2 Singleton

Singleton is another creational pattern used to "ensure a class only has one instance, and provide a global point of access to it" [2]. Figure 2.2 presents the structure of this pattern. The `Singleton` class is responsible for creating its unique instance and controlling access

to it. It is a simple pattern and it can be combined with other patterns, such as *Abstract Factory* when only one instance for each `ConcreteFactory` should exist.

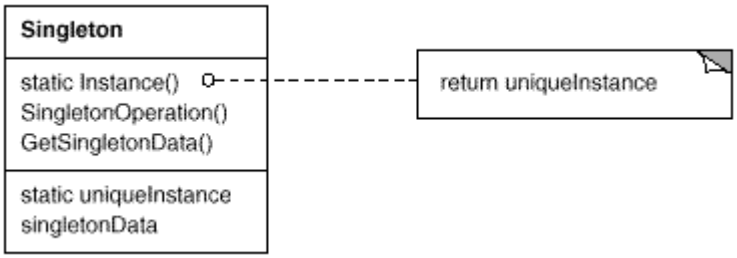


Figure 2.2: UML class diagram of the Singleton pattern structure [2]

2.2.3 Proxy

Proxy is a structural pattern used to "provide a surrogate or placeholder for another object to control access to it" [2]. Figure 2.3 presents the structure of this pattern. `Subject` is the exposed interface. `Proxy` is the placeholder responsible for communicating with the `RealSubject` and delegates its operations to the real implementation. It is especially useful for implementing a local representative that delegates the provided operations to a remote object.

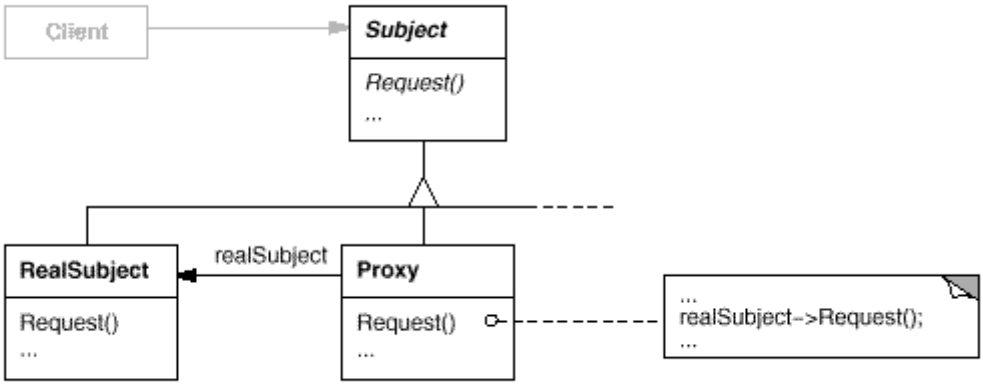


Figure 2.3: UML class diagram of the Proxy pattern structure [2]

2.2.4 Visitor

Visitor is a behavioral pattern used to "represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates" [2]. Figure 2.4 presents the structure of this pattern. `Visitor` is an interface that aggregates all the operations that must be

provided by a `ConcreteVisitor` in order to process an `ObjectStructure`. The object structure consists of an abstract `Element` with multiple `ConcreteElement` types that can be processed by visitors. This pattern makes it easy to add new operations (implement a new `Visitor`), but difficult to add new types to the object structure (every existing `ConcreteVisitor` must handle the newly added `ConcreteElement`). Thus, it is useful when the object structure is unlikely to change and must support many unrelated operations.

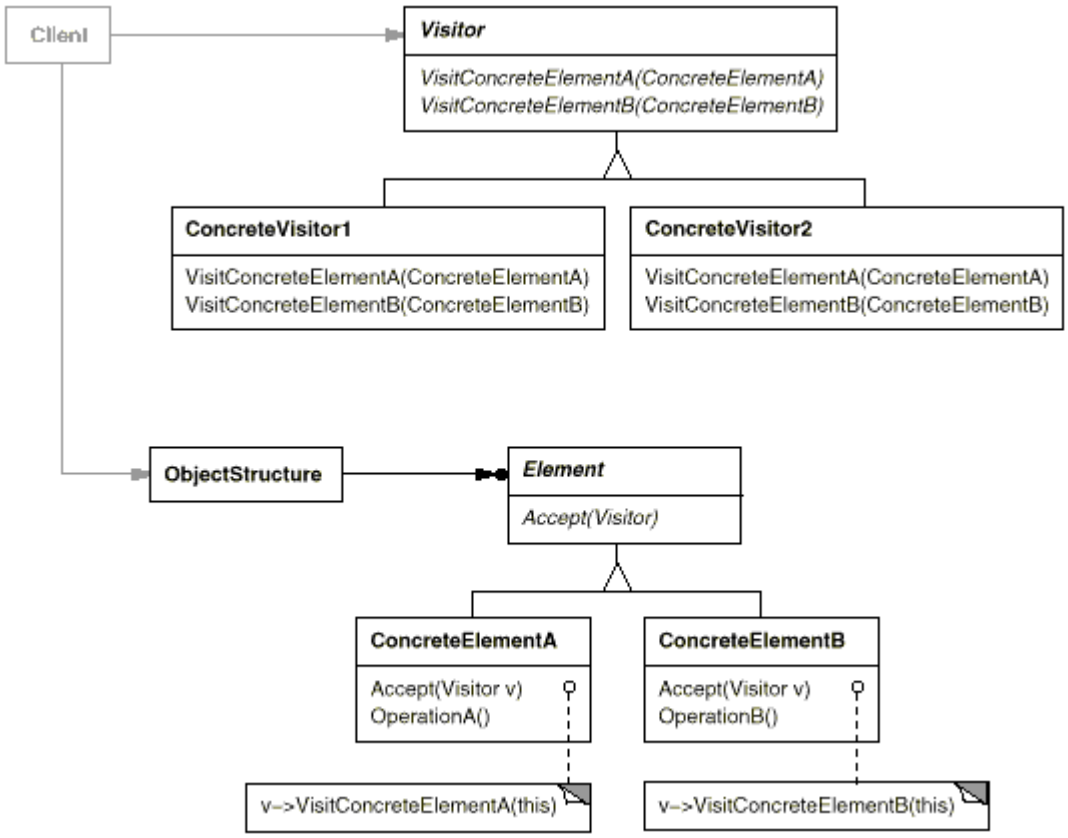


Figure 2.4: UML class diagram of the Visitor pattern structure [2]

2.3 Functional Programming

Functional programming is another paradigm, which originated from *lambda calculus*. We present the main concepts that are useful for our application.

Computations consist of evaluating mathematical **functions**, the fundamental concept of this paradigm. Moreover, functions can receive other functions as arguments or return another function. This special category of functions represents **higher-order functions**.

Programs are built using expressions instead of statements, avoiding mutable state. Formally, this immutability translates to the property of **referential transparency**.

Namely, an expression is referentially transparent if it can be substituted with the result of evaluating it (or vice versa) without changing the behavior of the program.

Typically, data structures in functional languages are represented as **algebraic data types**. Usually, algebraic data types are either *product types* or *sum types*. Product types, also called touples or records, contain multiple values, or fields. Sum types, also called disjoint unions, represent types whose instances are one of several other fixed types.

Oftenly, functional languages provide a mechanism to handle algebraic data types called **pattern matching**. Namely, it checks the value of an expression for a certain pattern indicated in code, typically in the form of sequences or tree structures (e.g., matching a list of at least three elements having the second element equal to a specific value, or a binary tree having a certain key associated to the left child node of the root).

Reynolds [7] proposed the *expression problem* to discuss advantages and disadvantages of various programming paradigms. The problem requires to model an expression that may have operands (types) and operators (behavior). Object-oriented programming makes it easy to add new operands (add a new type to the class hierarchy of operands and implement the existing operators), but difficult to add a new operator (its behavior must be implemented in every existing operand class). Functional programming makes it easy to add new operators (define a new function that handles the existing operands modeled by an algebraic data type), but difficult to add a new operand (the algebraic data type must be updated to include the new operand, and all operator implementations must be updated).

This comparison highlights the differences between functional and object-oriented programming. However, the two paradigms are not mutually exclusive, but orthogonal. Modern object-oriented languages incorporate functional concepts, such as *lambda functions* (e.g., *Java 8*), and other *hybrid* languages have been created (e.g., *Scala*). Thus, both paradigms can benefit software design by leaning towards the functional one when the programmer predicts it is more likely to add new behavior than new types, and leaning towards object-oriented when the programmer predicts it is more likely to add new types than new behavior.

2.4 Version Control Systems

Modern-day software systems have complex requirements and oftentimes cannot be developed in a timely manner by only a few programmers. Therefore, teams of programmers that work on a single project grew in size. But how can software development itself scale to multiple programmers? How can they simultaneously write or modify source files without conflicting changes? How do they track what changes fix a small bug, and what changes implement a new feature? What if the new feature is technically infeasible and some changes need to be reverted, but not changes that implement bugfixes?

To address these needs, tools that track changes applied to source code have been developed. Such tools are called **version control systems**. The history of changes applied to files in a project are stored in a **repository**. Every version of the project stored in the repository is denoted by a **revision**. Each revision has assigned a unique identifier, a timestamp and an author.

The state of the project at a specific revision can be *checked out*, creating a local copy of all the files of the project as they are found in that specific version. Then, changes can be applied to the local copy. However, these local changes are not yet stored in the repository. The modified local copy of the project is called a *working copy*. In order to create a new revision and store the working copy in the repository, it must be *checked in*, or *committed* to the repository.

How data is stored in a repository varies from one version control system to another, but usually, a graph structure is used. To save space, when the working copy is committed, instead of storing the entire project, only the local changes and a reference to the revision on which the working copy is based (*parent revision*) are stored (this method is known as *delta compression*). Thus, a chain of revisions, which are sorted chronologically, each referencing the previous as its parent, is formed. Such a chain is called a *branch*. The main development branch is usually called *trunk*, or *master branch*.

But what happens when we want to combine the changes from two (or more) branches (e.g., a bugfix branch based on an older version of the trunk with the latest version of the trunk)? How to resolve conflicting changes applied to the same source file? In this case, the branches are *merged* in a special revision called **merge revision**. Conflicting changes must be resolved manually by the programmer, and a main parent revision is selected as the base for the merge revision (the other branches are merged *into* the main parent).

This manner of storing revisions leads to the **revision graph**, a directed acyclic graph in which nodes represent revisions and edges indicate a parent-child relationship between revisions. Typically, a single root revision corresponding to the empty repository exists. The latest revision of the trunk is called `head` revision, or *tip*. Sometimes, the term `head` is used to denote the currently checked out revision.

Version control systems are an invaluable tool in the process of software development and are a key component in our application because they allow (among many other benefits) tracking the history of software systems.

3 State of the Art

3.1 Automatic Detection of Code Smells

Good software design is essential to ensure the maintainability of a system, and can significantly reduce the maintenance costs. Understanding the entire design of a system requires a lot of effort, and given the complex systems of today, can be close to impossible for a single programmer. Thus, assessing the quality of the design of a system is a difficult task.

However, given its importance, software quality assessment was not abandoned. Because manual assessment is infeasible, significant effort has been invested in developing automatic tools that recognize *bad design* in the form of **anti-patterns**, or **code smells**. Usually, these tools detect bad smells by analyzing the syntactic structure of a system. Fontana, Braione and Zanoni [8] published a comparative study on the existing tools that detect bad smells.

Some of these tools, such as CHECKSTYLE [9], detect classes that violate encapsulation by exposing public fields. However, the provided analysis is quite limited: public accessor methods are not reported as breaking encapsulation, and simple data carriers that expose their fields by design are reported as breaking encapsulation.

We believe a module breaks encapsulation only if its initially hidden fields were later exposed publicly. Therefore, we devised the *Encapsulation Breakers* analysis in an attempt to overcome the limitations of the existing tools.

3.2 Assessing Software Evolution

Although the syntactic structure of a software system provides important insights on its design and can be used to detect **structural smells**, there is more to inspect. As Lehman's *Law of Continuing Change* states, software changes and evolves continously. Perhaps, bad design could be recognized by analyzing *how a system evolves*. But how can we extract and understand the evolution of a system?

Lanza devised a visualization technique called **evolution matrix** [10] based on various metrics computed at various points in time for a system. Every class in a system is represented as a rectangle having its width and height determined by the values of two chosen metrics. Then, a matrix is built, where columns represent different versions of the system and lines represent individual classes. Figure 3.1 shows a schematic evolution matrix.

Thus, from the perspective of the two selected metrics, we can visualize the evolution of an individual class, the state of the entire system at a particular version, as well as the

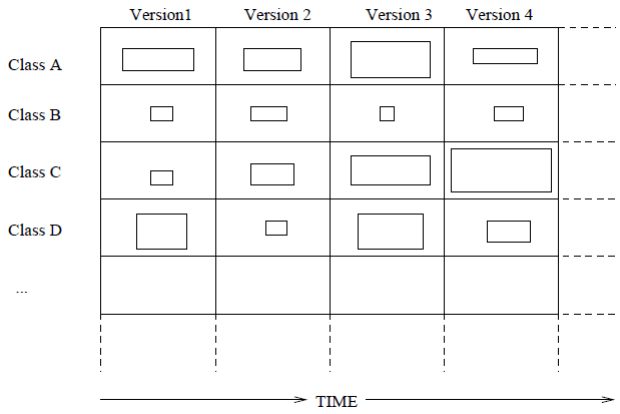


Figure 3.1: Schematic description of the evolution matrix [10]

evolution of the entire system at once. However, this approach has several limitations:

- the system can be visualized only through the perspective of two metrics at once (e.g., *number of methods* for width and *number of instance variables* for height)
- the selected metrics must be computed from a single snapshot of the syntactic structure of the system, missing the opportunity of analyzing the evolution itself (i.e., what pieces of code change from one snapshot to another)
- the visualization is effective only if many versions of the system are available

The last drawback is not specific to the evolution matrix, but applies to any analysis that makes use of the history of a system. Then, how can a large number of versions be retrieved for a typical software system?

The answer to the previous question lies within version control systems. Moreover, with the data stored in version control systems, we can build a better understanding of the evolution of a system than visualizing the evolution of a simple structural metric. Ball et al. [11] explored the potential data that can be extracted from version control systems, as well as what can be inferred from that data.

One of the concepts they introduced is the **co-change graph** of a system, based on the relations between classes. Figure 3.2 shows a sample co-change graph for one of the analyzed systems in their study. Nodes represent classes, edges represent a relation between two classes that change together, and the weight of an edge indicates how often the two classes changed together.

Let C be a class and $R(C)$ be the set of revisions in which C is modified. They compute the weight W of an edge between two classes, C and D, as follows:

$$W = \frac{|R(C) \cap R(D)|}{\sqrt{|R(C)||R(D)|}}$$

Running a clustering algorithm on the resulting weighted graph highlights classes that tend to change together.

This approach gives a better insight on how the *system* evolved, not just on how some *metrics* evolved. However, the visualized information is coarse-grained, as only co-change relations between classes are highlighted, the syntactic structure of the system is ignored.

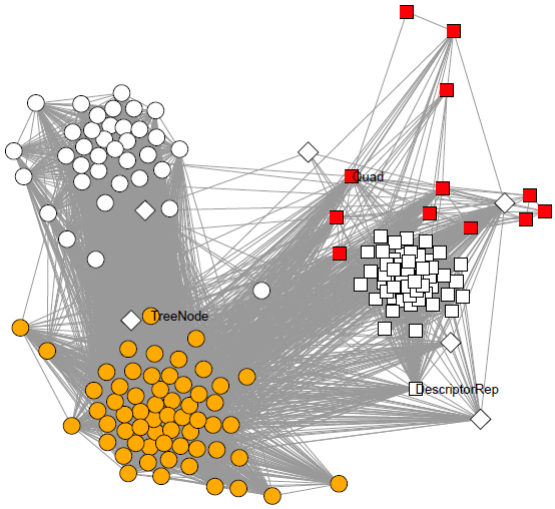


Figure 3.2: Example of a co-change graph [11]

We use a similar co-change graph in our *Single Responsibility Breakers* analysis, although at a more fine-grained level (nodes are methods in a file), with a different edge weight formula and a more suitable clustering algorithm for the purpose of our analysis.

Hismo is a meta-model devised by Gîrba et al. [12] that attempts to unify the history extracted from version control systems and the syntactic structure of a system. Although these approaches aid the understanding of software evolution, they do not detect any *code smells* or *anti-patterns*, giving no information on whether the design of the system is good or bad.

Still, a history model is necessary to enable software evolution analysis. Unlike *Hismo*, which uses the *FAMIX* [13] model to represent syntactical structure and a sequence of versions to represent the history of each entity, we devised a *history model* that makes use of a simpler, language-independent representation of source code, and integrated *changes* into the model. Thus, instead of maintaining multiple snapshots of the same entity, we maintain the initial snapshot of the entity along with what changed in each version. This approach makes analyzing the evolution of a system significantly easier, because we can focus on what changed from one version to another.

3.3 Evolution of Code Smells

Tufano et al. [14] conducted a comprehensive study on the evolution of code smells. Namely, they chose a sample of 200 software systems, extracted the history from version control systems and analyzed various snapshots with conventional tools to detect the following structural smells: *Blob Class*, *Class Data Should be Private*, *Complex Class*, *Functional Decomposition*, and *Spaghetti Code*.

Thus, they were able to observe how various code smells evolved, when they were introduced, and if they were fixed. By manually inspecting revisions in which code smells were introduced, they identified the patterns and reasons that led to bad design.

Rațiu et al. [15] also use history information to refine the detection of *Data Class* and *God Class* design flaws and to give further insights (e.g., whether a detected flaw had persisted for the entire lifetime of a module, or if it was introduced at a later point, or if it didn't cause maintainability problems because the affected module remained stable).

Although the history of a system was used to analyze the evolution of structural code smells or design flaws, the evolution itself was not analyzed.

3.4 Smells of Code Evolution

We believe that just as certain code smells can be detected by analyzing the syntactic structure of a system (*structural smells*), other smells can be detected by analyzing how the software evolves (**evolution smells**).

Towards this direction, D'Ambros, Lanza and Robbes [16] defined the concept of **change coupling** of a class with the entire system, unlike Ball et al. [11], who assigned weights to relations between classes. Furthermore, they mine a bug repository and attempt to predict software defects by correlating the mined bugs with the change coupling of the affected classes.

They define the concept of **n-coupled classes**, which is a pair of classes that changed together at least n times, and propose four measures of computing the change coupling of a class C with the rest of the system:

- **Number of Coupled Classes** — the number of n-coupled classes with C
- **Sum of Coupling** — the sum of the number of revisions C shares with every other class in the system that is n-coupled with C
- **Exponentially Weighted Sum of Coupling** — a variation of *Sum of Coupling* that assigns exponentially decreasing weights to older revisions
- **Linearly Weighted Sum of Coupling** — a variation of *Sum of Coupling* that assigns linearly decreasing weights to older revisions

These measures are coarse-grained, focusing on the coupling of a class with the entire system, and using absolute values. We use a similar *change coupling* concept in our *Single Responsibility Breakers* analysis, but at a more fine-grained level (between two methods in a class) and with a different goal: finding modules that fulfill multiple responsibilities, not predicting software defects.

In his book, *Your Code as a Crime Scene* [17], Tornhill proposed many analyses that inspect the evolution of a system using version control systems.

He defined a **hotspot** as being a complex module that changes frequently, indicating that a lot of effort is invested in a single, complicated piece of code. To detect hotspots, he combines two metrics: the *lines of code* to estimate complexity, and the *number of revisions* to identify frequently changing files.

We extend his use of hotspots in our *Open-Closed Breakers* analysis, and introduce a new metric to quantify how much a module is extended through modification over time. Tornhill's hotspots highlight files in which a lot of development effort was invested, but does not indicate what the problem is. We believe our *Open-Closed Breakers* analysis indicates a lack of abstraction for the highlighted modules.

Another concept Tornhill uses is that of **temporal coupling** (equivalent to *change coupling*), which aids finding hidden, implicit dependencies between modules. He uses the same *sum of coupling* as D'Ambros, Lanza and Robbes [16] as a preliminary measure to determine files that need a closer inspection. Then, he computes the *degree of coupling* as the percent of shared revisions between two files, as well as the *average number of revisions* of the two files. Thus, pairs of strongly coupled files can be identified.

We compute the temporal coupling between two methods in a similar manner to how he computes the temporal coupling between two files. Once again, the highlighted pairs of coupled files do not indicate the underlying problem. We use the coupling information further and build a co-change graph of the analyzed module in an attempt to identify subsets of methods that fulfill different responsibilities.

Tornhill goes even further and approaches the social aspects of software development: how many programmers modified a file, what percent of a file was written by the same programmer, and other metrics of team dynamics.

Palomba et al. [18] also extract the history of software systems from version control systems and detect the following smells: *Divergent Change*, *Shotgun Surgery*, *Parallel Inheritance*, *Blob*, and *Feature Envy*.

In the case of **Divergent Change**, their approach is based on mining association rules to identify sets of methods in a class that frequently change together. Thus, a class is detected as an instance of Divergent Change if it contains two or more sets of methods, where methods in the same set frequently change together, whereas methods in different sets do not change together oftenly.

The Divergent Change smell implicitly leads to breaking the *Single Responsibility Principle*. Thus, detecting instances of Divergent Change is tightly coupled to detecting instances of *Single Responsibility Breakers*. Whereas Palomba et al. [18] use association rules to identify subsets of methods that change together, we build the co-change graph and use a simple clustering algorithm to find subsets of methods that are tightly coupled to each other.

3.5 Summary

Software evolution can offer deeper insights on the quality of design. We developed a *History Analysis Tool* that extracts the history of a system, storing the changes from one version to another of a software entity. On top of the tool, we built three analyzers targeting specific design flaws. Our *Encapsulation Breakers* analysis attempts to overcome the limitations of simple static analyzers. We built the *Open-Closed Breakers* analysis on top of Tornhill's hotspots [17], adding a new metric and indicating a concrete design flaw. Our *Single Responsibility Breakers* analysis combines the co-change graph visualization developed by Ball et al. [11] and the Divergent Change detection strategy of Palomba et al. [18]. However, our analysis makes use of a different formula for temporal coupling, and uses a different clustering algorithm. The goal of our analyses is to indicate the causes that lead to a flawed design, not only symptoms of bad design. Moreover, our tool can be used to develop new analyses, enabling further research in the field of software evolution analysis.

4 History Analysis Tool

The main goal of our application is to detect design flaws by inspecting the evolution of software systems. To this end, we developed specialized analyzers that target specific anti-patterns. The analyzers were built upon a shared infrastructure that handles querying and extracting data from the history of a software system, whereas the analyzers process the extracted data. A thin wrapper around this infrastructure exposes a command-line interface, which can be used to manually inspect a software system and comprises a stand-alone tool named ChronoLens. The main goal of ChronoLens is to enable clients (humans or applications) to inspect the history of software systems. The tool and the analyzers were developed in the *Kotlin programming language* [19] and run on the *Java Virtual Machine* [20].

4.1 Architecture Overview

The developed tool, **ChronoLens**, is split into several components, which are presented in Figure 4.1. The infrastructure behind the tool and the specialized analyzers is implemented in the `chronolens-core` component, which defines the history model, provides abstractions for parsing source files and interacting with version control systems and implements algorithms for extracting the history model from version control systems. The `chronolens-java` and `chronolens-git` components provide implementations for the abstract services used by the infrastructure. A simple command-line interface that exposes the main operations is provided by the `chronolens` component, which also assembles the service implementations. Testing utilities such as mock objects, builders and assertions are provided by the `chronolens-test` component.

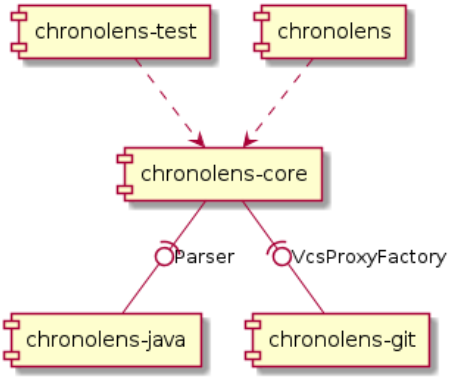


Figure 4.1: UML component diagram of ChronoLens

Figure 4.2 presents the architectural overview. The application infrastructure is split into several packages:

- `model` — contains the definition of the history model used to describe the structure and evolution of a software system
- `serialization` — handles persisting and loading the history model to and from the disk
- `parsing` — contains abstractions for language parsers
- `java` — provides a parser implementation for the *Java* programming language
- `subprocess` — contains utilities for launching and interacting with child processes
- `vcs` — contains abstractions for interacting with version control systems through proxies that make use of subprocesses
- `git` — provides a proxy implementation for the *Git* version control system
- `repository` — extracts the history model from the version control system, allows querying the model and provides persistence for the extracted data; it is the central component that ties everything together and enables the development of specialized analyzers
- `cli` — containts utilities for implementing command-line interfaces

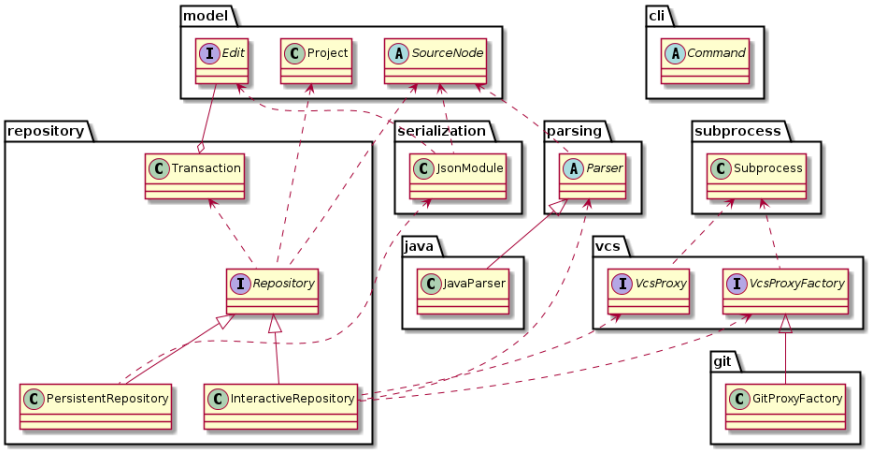


Figure 4.2: UML class diagram of the main ChronoLens modules

4.2 History Model Definition

At the core of ChronoLens is the language-independent **history model** used to represent the structure and evolution of software systems. The history model is implemented mainly through sealed hierarchies of immutable data classes. In this context, a sealed hierarchy emulates the concept of *algebraic data types* from functional programming and prevents the hierarchy from being extended outside of the source file where it is declared. Thus, client code can safely use *pattern matching* or the *Visitor pattern* to process the

model. This approach works well because the model is stable and unlikely to change. The added immutability also prevents the corruption of the model state while analyzing it.

4.2.1 Source Model

The **source model** reflects the structure of a system. We designed it as language-agnostic as possible to allow describing as many programming languages as possible, but retaining the relevant information for software analysis at the same time. To this end, we decided to model source files, types, functions and variables, because these concepts are present in almost all widespread programming languages (dynamic or static). However, variable initializers and method bodies are restricted to simple text, without other syntactic information, because the information would be too fine-grained and language-specific to be described by a general data structure. This leads to a significant limitation regarding the potential for static code analysis, but it is an acceptable trade-off for three reasons:

- dynamic languages become increasingly popular (e.g., Python, Javascript, PHP — see Figure 4.11)
- it is difficult to perform static analysis on dynamic languages
- we aim to perform analyses that inspect the structure and evolution of a software system, and less its exact syntactic content

Source code is modeled by a sealed hierarchy of immutable data classes, which is presented in Figure 4.3:

- `SourceNode` — represents a source node within a project (can be a file, type, variable or function); it is uniquely identified by a fully qualified *id*
- `SourceFile` — represents the aggregated set of top-level *entities* defined in a source file; its *id* consists of the `/`-separated *path* of the file, relative to the project root directory
- `SourceEntity` — represents an entity defined in a source file (can be a type, variable or function); its *id* consists of the concatenated id of the parent source node and its simple id, separated by `:` for types and `#` for functions and variables
- `Type` — represents a type (class, interface, enumeration, etc.) defined in a source file or inside another type (e.g., nested classes); its simple id is its *name*; it may have a set of *supertypes* (superclass, implemented interfaces, etc.) retained as raw strings, a set of *modifiers* (access modifiers such as `public` or `private`, other semantic modifiers such as `static` or `volatile`, annotations like `@Override`, etc.) and a set of *members* (fields, methods or nested types)
- `Function` — represents a method belonging to a type or a top-level function defined in a source file; its simple id is its *signature* and it contains a list of *parameters* describing the parameter names; it may have a set of *modifiers* and a *body* consisting of the raw lines of code that comprise the method implementation (the body is empty for abstract methods)
- `Variable` — represents a field belonging to a type or a top-level variable defined in a source file; its simple id is its *name*; it may have a set of *modifiers* and an *initializer* (the initializer is empty if the field is not initialized, or if it is initialized in a constructor, a static initializer, etc.)

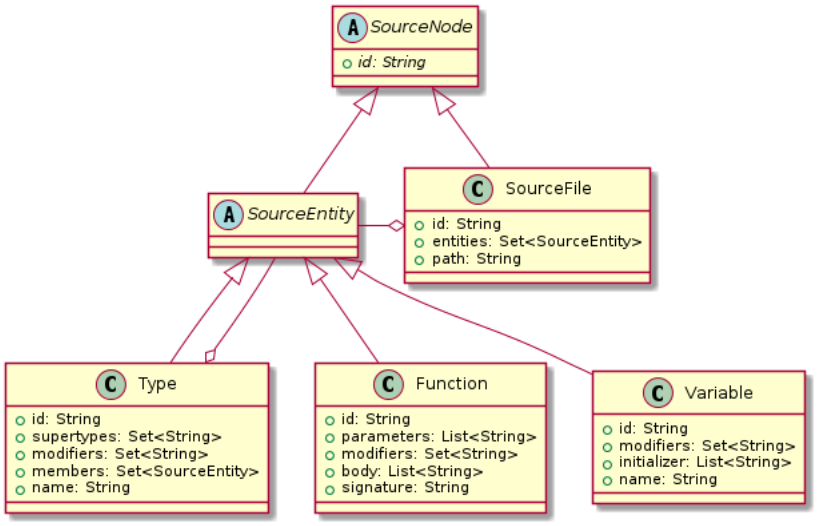


Figure 4.3: UML class diagram of the source model

Note that source entities have no knowledge of the parent source node. This prevents circular references in the model and makes serialization and immutability simpler to implement. However, the parent source node of an entity can still be uniquely identified just by looking at its fully qualified *id*: the parent *id* is the prefix of the entity's *id*, up to the last occurrence of the separator characters `:` or `#`.

The source model defined so far allows us to extract the code structure across all versions of each source file, opening up the door to many introspection ideas: how often does a type, function or variable change, how much does it change, how do the methods of a class change relative to each other, etc. An advantage of this approach is parallelism: the history of each source file can be inspected independently on a different thread, allowing to scale such analyses to arbitrarily large software systems.

However, there is a significant disadvantage: we cannot track the evolution of the system as a whole, but only the evolution of individual source files. It would be more difficult to track correlations across different source files (e.g., a method from a class that changes frequently together with many methods from a different class might indicate an instance of *Feature Envy*) and limits the spectrum of analyses that can be performed. Moreover, detecting certain refactorings (rename a file, move a method to another class, etc.) can increase the quality of the performed analyses, but detecting such refactorings would be difficult if the history of each file is tracked independently.

In order to allow tracking the history of a software system as a whole, we added a new data structure to the source model, described in Figure 4.4:

- `Project` — represents the entire system; it contains the set of *sources* within the system, as well as the entire *source tree* (the set of all source nodes, from source files and top-level classes to fields of inner classes), and provides fast access to source nodes by *id* without having to walk the source tree

Unlike the rest of the history model, this data structure is mutable and allows a restricted set of changes to be applied to its state. Although persistent data structures such as those described by Driscoll et al. in "Making Data Structures Persistent" [21]

could maintain the immutability and add only a logarithmic factor to the running time of the tool, in this case we preferred the mutable approach for the increased performance.

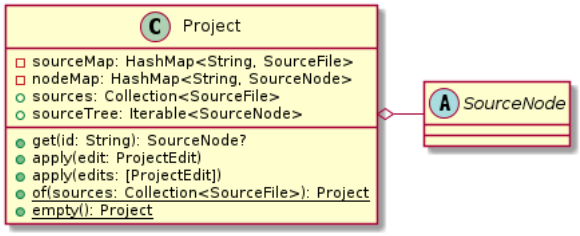


Figure 4.4: UML class diagram of Project

4.2.2 Delta Model

Having access to the structure of a software system at every snapshot in its history is useful, but poses several problems:

- huge amounts of data are produced due to the quadratic growth (the amount of data is proportional to the product between the number of source files and the number of revisions in the project repository)
- computing this data might require a long time
- it is difficult to handle such a large amount of data, both in-memory and on-disk
- while it allows inspecting the structure of the system at every snapshot in its history, it doesn't facilitate inspecting its evolution: there is no easy way to tell how much the system changed from one snapshot to another, nor what exactly changed

Thus, we decided to include in the history model the actual changes applied to the system from one snapshot to another. These changes comprise the **delta model**[1] and reflect the evolution of a system. In order to accurately describe the evolution, we must model changes for every data structure used in the source model: *lists*, *sets*, *source nodes* and *projects*. The changes are modeled by a few sealed hierarchies of immutable data classes, which are presented in Figures 4.5, 4.6 and 4.7:

- `Edit` — represents a simple, atomic change that cannot be split into smaller sets of changes, and should be applied on a generic entity of type `T` (adding a line to a method body, removing a modifier from a field, etc.).
- `SetEdit` — represents an edit that should be applied on a *set* of objects
  - `Add` — signals that a certain *value* should be inserted in the modified set
  - `Remove` — signals that a certain *value* should be deleted from the modified set
- `ListEdit` — represents an edit that should be applied on a *list* of objects

[1] In mathematics, the greek letter *delta* (uppercase Δ, lowercase δ) is oftenly used to describe a change or difference, hence why we chose the name *delta model*.

  - `Add` — signals that a certain *value* should be inserted at the specified *index* in the modified list
  - `Remove` — signals that the value at the specified *index* should be deleted from the modified list
- `ProjectEdit` — represents an edit that should be applied on the node with the specified *id* from the modified *project*
  - `AddNode` — signals that a certain *node* (and all of its descendants) must be inserted in the modified project
  - `RemoveNode` — signals that the node with the given *id* (and all of its descendants) must be deleted from the modified project
  - `EditType` — signals that the specified *modifier edits* and *supertype edits* must be applied on the *type* with the given *id*
  - `EditFunction` — signals that the specified *modifier edits*, *parameter edits* and *body edits* must be applied on the *function* with the given *id*
  - `EditVariable` — signals that the specified *modifier edits* and *initializer edits* must be applied on the *variable* with the given *id*

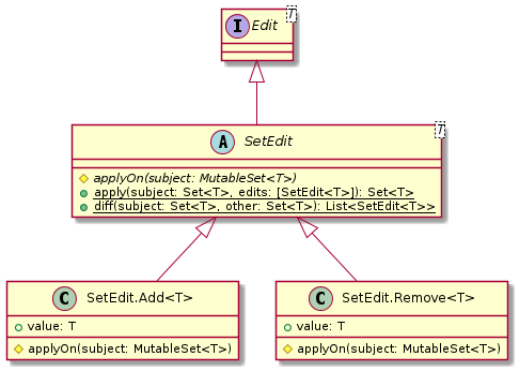


Figure 4.5: UML class diagram of SetEdit

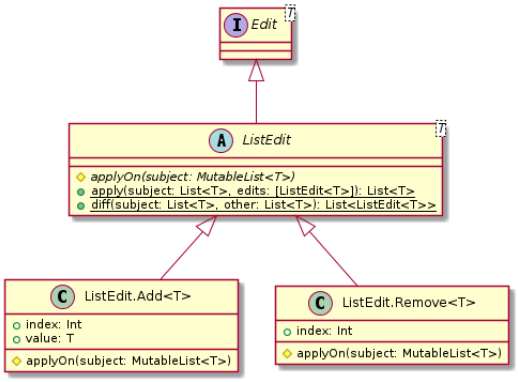


Figure 4.6: UML class diagram of ListEdit

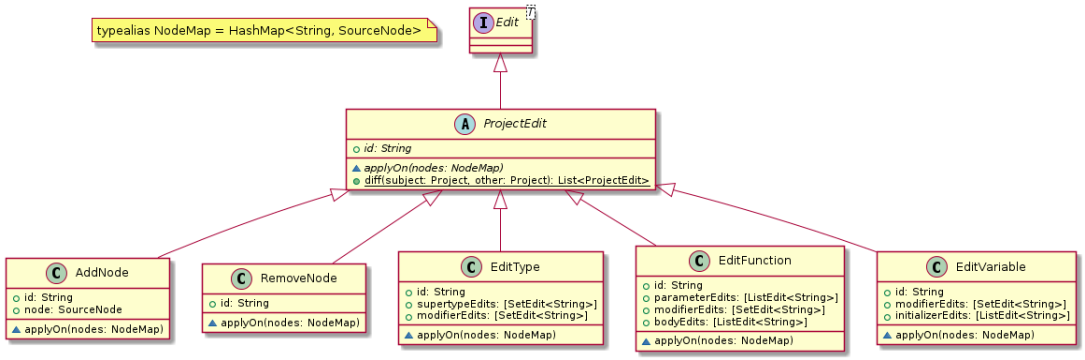


Figure 4.7: UML class diagram of ProjectEdit

The history of a system consists of multiple revisions, each having a date, an author and a set of changes. The edits from the delta model encapsulate a single complex change

applied to the system and are insufficient to capture all the details of a revision. Thus, we added a new data class to model revisions, which is presented in Figure 4.8:

- `Transaction` — represents complex changes comprising multiple *edits* applied to a *project* (e.g., adding a method to a type and removing a nested class) and encapsulates the modifications performed in a specific revision of the project repository, including the *date*, *author* and *change set* (modified file paths) of the revision

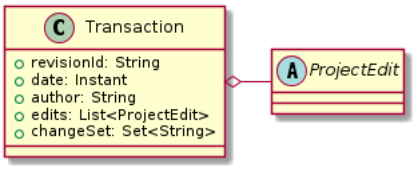


Figure 4.8: UML class diagram of Transaction

Finally, the **history model** consists of the list of *transactions* applied to a software system, which correspond to the revisions from the history of the system.

This approach solves the problems posed by processing the entire system at every snapshot because the amount of produced data is linear in the number of changes applied to the system.

The presented approaches are two extremes: one tracks changes of files individually and allows large-scale parallelization, whereas the other tracks changes of the system as a whole and is not scalable to multiple threads. We propose a middle-ground solution: process the history of a specified directory from the project repository as if it were the only source root. Thus, a system can be analyzed in two ways:

- if the system is of average size and can be fully analyzed using a reasonable amount of time and memory, then the root directory of the repository will be processed and all correlations from the system will be captured (even across different components, possibly written in different programming languages)
- if the system is too large to be analyzed as a whole, a list of directories (corresponding to smaller, logical components) can be processed in parallel, capturing correlations within each analyzed component, but not between components

We believe this middle-ground approach is a suitable trade-off between the two extremes, because it allows choosing between scalability and introspection granularity for the analyzed system.

4.3 Model Data Format

Computing the history model every time we wish to analyze a software system is timewise expensive and redundant. A solution to this problem is to compute the model only once and persist it on the disk, allowing various analyses to be performed without having to extract the history model again, thus saving a significant amount of computation time.

We chose the *JSON* data format, a compact textual representation of data, because it is human-readable, widespread and highly interoperable with front-end technologies [22]. The data format follows closely the history model and can be understood with a comprehensive example.

Consider the following fictional scenario: Alice and Bob just started learning programming together and wish to write a Java program that prints `"Hello, world!"` to the screen, while tracking their work with a version control system that assigns increasing integers starting with 1 to the revision ids. Bob sets up the project repository, writes the first version of the program `Main.java` and saves his work in the first revision.

Listing 4.1: The first version of the Main.java file

```java
public class Main extends Object {
    private static final String name = "world";
    static void main(String[] args) {
        System.out.println("Hello, "+ name + "!");
    }
}
```

Extracting the model from the first revision will produce the following transaction:

Listing 4.2: The transaction corresponding to the first revision

```
{
  "revisionId": "1",
  "date": 1517844341968,
  "author": "Bob",
  "edits": [{
    "@class": "AddNode",
    "id": "src/java/Main.java",
    "node": {
      "@class": "SourceFile",
      "id": "src/java/Main.java",
      "entities": [{
        "@class": "Type",
        "id": "src/java/Main.java:Main",
        "supertypes": ["Object"],
        "modifiers": ["public", "class"],
        "members": [{
          "@class": "Variable",
          "id": "src/java/Main.java:Main#name",
          "modifiers": ["private", "static", "final"],
          "initializer": ["\"world;\""]
        }, {
          "@class": "Function",
          "id": "src/java/Main.java:Main#main(String[])",
          "parameters": ["args"],
          "modifiers": ["static"],
          "body": ["{", "System.out.println(\"Hello, \" + name + \"!\");", "}"]
        }]
      }]
    }
  }]
}
```

Looking at Bob's code, Alice notices a mistake: the `main` method must be public to allow the program to run, so she adds the `public` modifier. She also removes the `Object` superclass and the `name` field from the `Main` class, as they are not necessary for the purpose of their program. Then, Alice saves the second version of the `Main.java` file in the second revision.

Listing 4.3: The second version of the Main.java file

```
1 public class Main {
2     public static void main(String[] args) {
3         System.out.println("Hello, world!");
4     }
5 }
```

Extracting the model from the second revision will produce the following transaction:

Listing 4.4: The transaction corresponding to the second revision

```
1  {
2    "revisionId": "2",
3    "date": "1517844361571",
4    "author": "Alice",
5    "edits": [{
6      "@class": "RemoveNode",
7      "id": "src/java/Main.java:Main#name",
8    }, {
9      "@class": "EditType",
10     "id": "src/java/Main.java:Main",
11     "supertypeEdits": [
12       { "@class": "Set.Remove", "value": "Object" }
13     ]
14   }, {
15     "@class": "EditFunction",
16     "id": "src/java/Main.java:Main#main(String[])",
17     "modifierEdits": [
18       { "@class": "Set.Add", "value": "public" }
19     ],
20     "bodyEdits": [{
21       "@class": "List.Remove",
22       "index": 1
23     }, {
24       "@class": "List.Add",
25       "index": 1,
26       "value": "System.out.println(\"Hello, world!\");"
27     }]
28   }]
29 }
```

With the help of Alice and Bob's learning experience, we were able to describe the data format used to store the history model in detail.

Serializing and deserializing the model to and from the presented JSON format is achieved with the help of the *Jackson* third-party library [23]. Reading and writing arbitrary objects is handled by the `JsonModule` *singleton*, which serializes all non-transient fields (including the private ones) via reflection. The `@class` special field added to the classes that are part of the history model is used to facilitate polymorphic deserialization

of collections that contain abstract types. The mappings from logical names to fully qualified classes achieved through the `@class` field are given in Table 4.1.

Table 4.1: Mappings from logical names to fully qualified history model classes

Logical name	Fully qualified class
SourceFile	org.chronolens.core.model.SourceFile
Type	org.chronolens.core.model.Type
Function	org.chronolens.core.model.Function
Variable	org.chronolens.core.model.Variable
List.Add	org.chronolens.core.model.ListEdit.Add
List.Remove	org.chronolens.core.model.ListEdit.Remove
Set.Add	org.chronolens.core.model.SetEdit.Add
Set.Remove	org.chronolens.core.model.SetEdit.Remove
AddNode	org.chronolens.core.model.AddNode
RemoveNode	org.chronolens.core.model.RemoveNode
EditType	org.chronolens.core.model.EditType
EditFunction	org.chronolens.core.model.EditFunction
EditVariable	org.chronolens.core.model.EditVariable

4.4 Delta Model Computation

Although the *source model* is directly reflected in the structure of a software system, there is no obvious way to figure out what the *delta model* is just by looking at two versions of the system. It is important to note that there are multiple valid delta models for two snapshots of a system. The most straightforward one is to delete all sources from the first snapshot and add all sources from the second snapshot. However, this model has an unnaceptable quality, because it doesn't capture the evolution of the system. It is desirable for the delta model to minimize the number of deletions and insertions of source nodes, as well as the number of edits that must be applied to attributes of source entities. Computing a delta model that reflects the evolution of the system according to the aforementioned properties is achieved through a family of algorithms that work at different levels of granularity:

- computing differences between sets and lists
- computing differences between the attributes of source entities
- computing differences between projects

In our application, we implemented these algorithms as a series of overloaded *extension* methods named `diff`. An extension method has an implicit receiver accessible through the `this` keyword and can be invoked on receiver instances as a member method. Under the hood, extension methods are compiled as static methods with an additional parameter corresponding to the receiver instance.

Computing the differences between two sets of objects is straightforward: remove the objects from the first set that do not exist in the second, and add the objects from the second set that do not exist in the first.

The minimum number of edits, as well as the actual edits that must be applied on a list of objects to obtain another list can be computed with the *Wagner-Fischer algorithm* [24]. This step could be optimized to use a faster algorithm, such as the one devised by Berghel and Roach [25], but it takes only a small amount of the execution time and the implementation effort was not justified.

When comparing two source entities, they must have matching ids and types and only their attributes can be edited (added or removed child source nodes are handled separately). Note that no edits can be applied to a source file, because source files have no attributes besides their child source entities. The `diff` method implementation for functions (the implementations for types and variables follow the same pattern) is given in the following listing:

Listing 4.5: Function.diff

```
1  private fun Function.diff(other: Function): EditFunction? {
2      val parameterEdits = parameters.diff(other.parameters)
3      val modifierEdits = modifiers.diff(other.modifiers)
4      val bodyEdits = body.diff(other.body)
5      val edit = EditFunction(id, parameterEdits, modifierEdits, bodyEdits)
6      val changed = parameterEdits.isNotEmpty()
7          || modifierEdits.isNotEmpty()
8          || bodyEdits.isNotEmpty()
9      return if (changed) edit else null
10 }
```

We also implemented a utility method that computes the differences between two arbitrary source nodes, however, as we mentioned, they must have the same id and type:

Listing 4.6: SourceNode.diff

```
1  internal fun SourceNode.diff(other: SourceNode): ProjectEdit? {
2      require(id == other.id && this::class == other::class) {
3          "Can't compute diff between '$id' and '${other.id}'!"
4      }
5      return when (this) {
6          is SourceFile -> null
7          is Type -> diff(other as Type)
8          is Function -> diff(other as Function)
9          is Variable -> diff(other as Variable)
10     }
11 }
```

Computing the differences between two snapshots of a system must take into account the strong impact insertions and deletions of source nodes have on the quality of the resulting delta model. We devised the following strategy that minimizes the number of insertions and deletions, similarly to the approach for computing the differences between two sets:

- if a source node exists in the first snapshot, does not exist in the second snapshot, and its parent node exists in the second snapshot, then it was removed, along with all of its descendants (direct or indirect child nodes)

- if a source node exists in the second snapshot, does not exist in the first snapshot, and its parent node exists in the first snapshot, then it was added, along with all of its descendants
- if a source node exists in both snapshots, we assume it was not added or removed, but may have some changed attributes
- a source node whose parent does not exist in one of the two snapshots is ignored (it is covered by adding or removing one of its ascendants)

Consider the source tree which contains the nodes that exist in at least one of the two snapshots and retains only the ids, ignoring the attributes. The delta model can be computed according to the devised strategy by iterating over the node ids of this tree in any order and processing the pairs of source nodes with the specified id from the two snapshots. The implementation of this algorithm is given in the following listing:

Listing 4.7: Project.diff

```
fun Project.diff(other: Project): List<ProjectEdit> {
    val thisSources = this.sources.map(SourceFile::id)
    val otherSources = other.sources.map(SourceFile::id)
    val allSources = thisSources.union(otherSources)

    val nodesBefore = NodeHashMap()
    val nodesAfter = NodeHashMap()
    for (path in allSources) {
        this[path]?.let(nodesBefore::putSourceTree)
        other[path]?.let(nodesAfter::putSourceTree)
    }

    fun parentExists(id: String): Boolean {
        val parentId = id.parentId ?: return true
        return parentId in nodesBefore && parentId in nodesAfter
    }

    val nodeIds = nodesBefore.keys + nodesAfter.keys
    return nodeIds.filter(::parentExists).mapNotNull { id ->
        val before = nodesBefore[id]
        val after = nodesAfter[id]
        val edit = when {
            before == null && after != null -> AddNode(after)
            before != null && after == null -> RemoveNode(id)
            before != null && after != null -> before.diff(after)
            else -> throw AssertionError("Node '$id' doesn't exist!")
        }
        edit
    }
}
```

With the help of the algorithms described in this section, we are able to accurately compute the list of changes that occur in a revision of a software system.

4.5 Model Extraction

Extracting the *history model* from a software system requires the raw history stored in version control systems, parsers that interpret source code, the delta model computation algorithms presented in Section 4.4, and a central component to tie everything together.

The dataflow during the history extraction and inspection is described in Figure 4.9. The project repository plays the role of a database that can be queried. Specialized parsers interpret the raw source code retrieved from the version control system through the proxy into the source model. From there, the interactive repository performs the delta model computation. Having access to the full history model, the persistent repository writes it to the disk, making it available to specialized analyzers, while the command-line interface makes it available to human users.

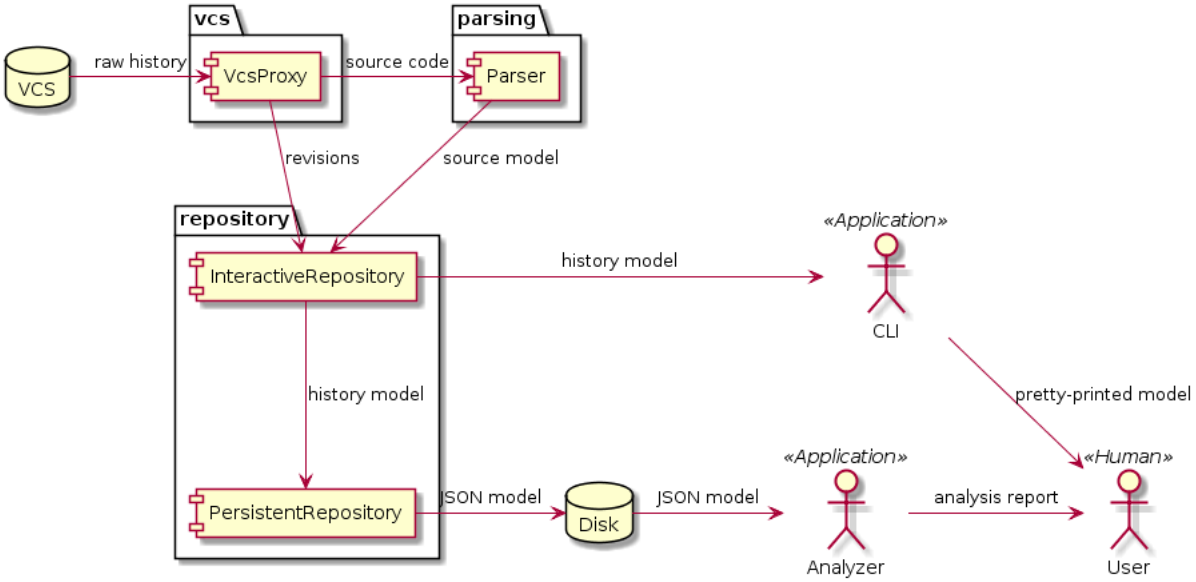


Figure 4.9: Dataflow during history extraction and inspection

4.5.1 Version Control System Proxies

Analyzing the evolution of a software system requires snapshots at different points in time over the lifetime of the system. This data is retrieved using version control systems.

Version control systems are an essential tool in the software development process and, unsurprisingly, are used in the development of most software systems. Although the features differ from vendor to vendor, they all provide several common operations that are useful to our application:

- keep different versions of files in the same location (repository)
- track the changes applied to the files
- query the history of the files

Extracting the history from a specific version control system is achieved through the `VcsProxy`, which delegates the queries to subprocesses that interact with the actual implementation provided by the vendor. The `VcsProxyFactory` *abstract factory* allows adding support for any specific version control system without changing the code that extracts the history model.

Data retrieved from Google Trends [26] shows in Figure 4.10 the increase in popularity of *Git* compared to other version control systems, which motivated us to provide a reference *proxy* implementation for it.

Figure 4.10: Google Trends on popularity of version control systems [26]

4.5.2 Git Proxy

Git is a widely-used distributed version control system, originally created by Linus Torvalds to aid the development of the Linux kernel. Some of its key characteristics are its branching capabilities, the distributed development which also enables working offline, and its design towards performance and scalability.

The proxy implementation for Git makes use of several commands, which are described in detail in the reference manual [27]:

Listing 4.8: Git commands

```
1 # Ensures that the specified revision exists (for validation purposes).
2 git cat-file -e "$revision^{commit}"
3
4 # Retrieves the specified revision by printing "commit $id" on the first line
5 # and "$date:$author" on the second line of output.
6 git rev-list -1 --format=%ct:%an "$revision^{commit}"
```

```
7
8  # Retrieves the list of modified files in the specified revision.
9  git diff-tree -m -r --root --name-only --relative --no-commit-id "$revision"
10
11 # Retrieves the list of files that exist in the repository in the specified
12 # revision.
13 git ls-tree -r --name-only "$revision"
14
15 # Retrieves the contents of the file located at the specified path from the
16 # specified revision.
17 git cat-file blob "$revision:$path"
18
19 # Retrieves the revisions of the specified path that are located on the path
20 # from the HEAD revision to the root of the revision graph in chronological
21 # order, following the first parent in the case of merge revisions. The printing
22 # format is the same as the one for retrieving a single revision.
23 git rev-list --first-parent --reverse --format=%ct:%an HEAD -- "$path"
```

To better understand how these commands are used in the tool, let us recall Alice and Bob's learning experience presented in Section 4.3. Assume they used Git as a version control system and, for the sake of simplicity, ignore the fact that 1 and 2 are not valid commit ids. Now, consider the following scenario: Carol, a friend of Alice and Bob, has access to the project repository and wants to understand how their program evolved. The first step would be to identify the `head` commit and the existing source files:

Listing 4.9: Printing the HEAD commit

```
1 > git rev-list -1 --format=%ct:%an "HEAD^{commit}"
2 commit 2
3 1517844361571:Alice
4
5 > git ls-tree -r --name-only "2"
6 src/java/Main.java
```

Carol now knows the current state of the project. However, previous version might have had other source files as well. Thus, the next step Carol takes is to discover all the commits of the repository and the files modified in each commit:

Listing 4.10: Printing the history and change sets of the repository

```
1  > git --first-parent --reverse --format=%ct:%an HEAD -- "."
2  commit 1
3  1517844341968:Bob
4  commit 2
5  1517844361571:Alice
6
7  > git diff-tree -m -r --root --name-only --relative --no-commit-id "1"
8  src/java/Main.java
9
10 > git diff-tree -m -r --root --name-only --relative --no-commit-id "2"
11 src/java/Main.java
```

What Carol learned from the output of the previous commands is that the project consisted of a single source file at all times, Bob created that source file, and Alice modified it afterwards. The final step is to compare the two versions of the `Main.java` source file:

Listing 4.11: Comparing the two versions of the Main.java source file

```
1 > git cat-file blob "1:src/java/Main.java"
2 public class Main extends Object {
3 ...
4
5 > git cat-file blob "2:src/java/Main.java"
6 public class Main {
7 ...
```

With the help of Carol's exploration of the project repository, we gained a better understanding on how the various Git commands can be used to extract the history of a system.

Still, there is one more important aspect regarding the history extraction process: the `--first-parent` option passed to the `rev-list` command will retrieve only the first (main) parent of *merge commits*. Thus, the history is reconstituted as a linear development branch starting from the inception of the project and ending with the `head` revision. Replicating the entire reivison graph when computing the *delta model* is infeasible, because we cannot logically model merge commits. Therefore, we decided to use this approach, which follows only the main development branch.

4.5.3 Source File Parsers

Source code interpretation is achieved using the `Parser` for a specific programming language.

The parser implementation reads the source code from a source file, builds an abstract syntax tree and converts it into the source model described in Section 4.2.1. The abstract parser allows adding support for any programming language that uses concepts that can be mapped to types, variables and functions without changing the code that extracts the source model.

The TIOBE index [28] is a measure of popularity for programming languages. The rankings until May, 2017 show in Figure 4.11 that *Java* still is the most popular programming language, which motivated us to provide a reference parser implementation for Java.

4.5.4 Java Parser

Parsing Java code into the source model is achieved through the *Eclipse JDT Core* [30] third-party library.

The parser builds the abstract syntax tree for the raw source code it receives. Thus, we have access to all language constructs specific to Java. We perform the following mappings from Java to our source model:

- classes, interfaces, enumerations and annotation classes are converted into *types*
- methods become *functions* (regardless of their modifiers — `static`, `abstract`, `final`, `private`, etc.)
- constructors are converted into *functions* with the name equal to the class name
- fields become *variables* (regardless of their modifiers)

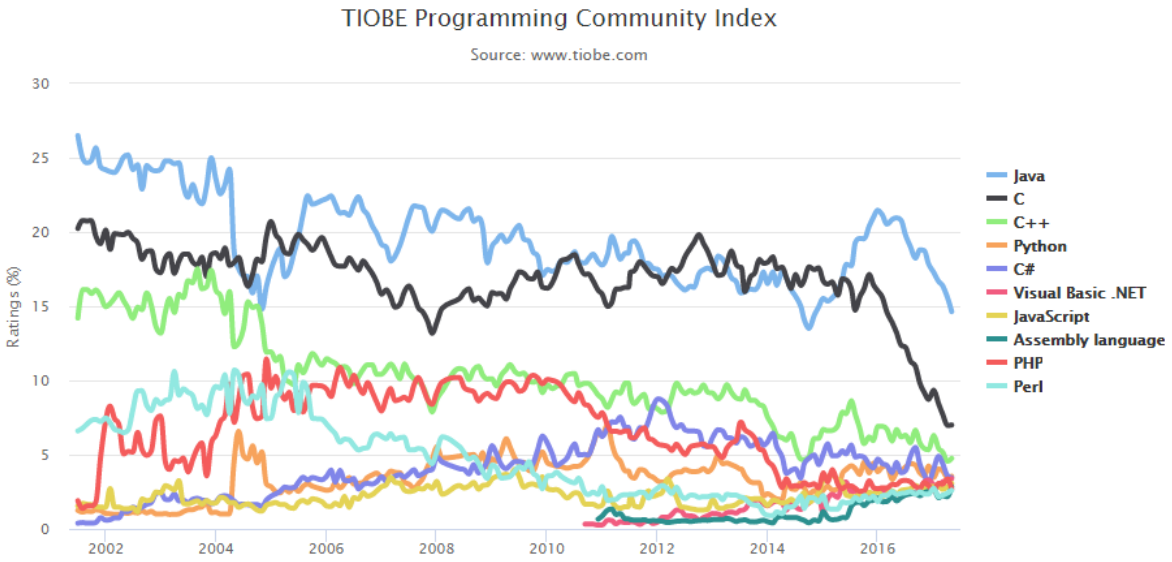


Figure 4.11: TIOBE index on popularity of programming languages [29]

- method bodies and field initializers are parsed as raw lines of code
- enumeration constants become *variables*, and in the case of constants with an anonymous class body, the class body becomes the *initializer*
- annotation members are converted into *variables*, and the default values become the *initializers* (if they exist)

All modifiers, applied annotations and keywords used to define a source entity are retained in the *modifiers* attribute. Note that there are some source elements which are not retained because the source model is too coarse-grained:

- initializers (unnamed blocks of code that are run when an object or class is initialized) are ignored, because they do not have a unique identifier
- only the names of method parameters are retained (applied annotations and other modifiers are ignored)

However, the lost source elements have little importance in the evolution of the system and keeping the model simple and language-independent is worth the loss in granularity.

4.5.5 Repositories

Only one component is missing from the complete design of the tool: a model for the project repository. Thus, we added the `Repository` interface, which has the role of a controller and is presented in Figure 4.12. It is capable of retrieving the `head` revision, the list of interpretable source files and the list of revisions of the system, as well as extracting the full history model and the full source model from the `head` revision. Currently, there are two types of repositories: interactive and persistent. Interactive repositories extract and compute the required data from the version control system. Persistent repositories load the required data from the disk.

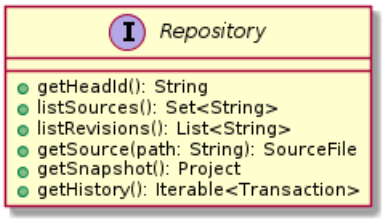


Figure 4.12: UML class diagram of the Repository interface

The interactive repository has the responsibility of extracting and computing the history model. It is primarily used by the command-line interface. It also extends the `Repository` interface with two operations: it can retrieve the list of interpretable source files and the corresponding source model from any revision. The model extraction algorithm works as follows:

- start with an empty `Project`, corresponding to the empty repository
- retrieve the next revision until none are left
- build the subprojects corresponding to the state of the change set before and after the revision
- compute the delta model for the two subprojects
- append the resulting transaction to the history
- update the `Project` with the changes from the current revision

The implementation of this algorithm is given in the following listing:

Listing 4.12: InteractiveRepository.getHistory

```
override fun getHistory(): Iterable<Transaction> {
    val project = Project.empty()
    return history.mapLazy { (revisionId, date, author) ->
        val changeSet = vcs.getChangeSet(revisionId)
        val before = HashSet<SourceFile>(changeSet.size)
        val after = HashSet<SourceFile>(changeSet.size)
        for (path in changeSet) {
            val oldSource = project.get<SourceFile?>(path)
            before += listOfNotNull(oldSource)
            val result = parseSource(revisionId, path)
            val newSource = when (result) {
                is Result.Success -> result.source
                Result.SyntaxError -> oldSource ?: SourceFile(path)
                null -> null
            }
            after += listOfNotNull(newSource)
        }
        val edits = Project.of(before).diff(Project.of(after))
        project.apply(edits)
        Transaction(revisionId, date, author, edits)
    }
}
```

There are three important aspects to mention, which impact performance, memory consumption and the quality of the model:

- when walking the source tree described in Section 4.4, it is sufficient to consider only the subtrees included in the change set of the revision (other source nodes did not change and do not influence the resulting delta model), hence why we compute the delta model only for the two subprojects; this optimization step is essential to ensure the linear time complexity in the number of changes of the algorithm
- the history is returned as a lazy iterable collection to avoid keeping the entire history model in memory; this way, only one transaction is retained in memory at any time, greatly reducing the memory consumption
- source files that contain syntax errors in a certain revision have that revision ignored

Persistent repositories are a matter of serializing and deserializing the data required to satisfy the `Repository` interface. The `head` revision, the list of interpretable sources and the list of revisions are stored each in their own file. Each source file from the `head` revision is stored within a directory structure that mimics the project repository. This way, the stored source model for the file can be located just by looking at the path of the source file. Every transaction is stored in a separate file named after the id of the revision it models. This avoids having to parse a large file to deserialize the history model, which may pose problems to certain JSON parsers, and allows loading and returning the history model as a lazy iterable collection. The time required to deserialize the individual transactions when reconstructing the history model is negligible compared to the extraction and computation of the model. Thus, we preferred this approach instead of serializing the entire history in a single file.

The tool is now complete. With the help of the extracted history model and the query operations offered by the interactive repository implementation, we are capable of inspecting (both manually and programatically) the fine-grained evolution of software systems, as if looking at their history through a magnifying glass, while the persistent repository implementation enables multiple specialized analyzers to efficiently inspect the history model.

5 Detecting Software Anti-Patterns

With the help of ChronoLens, we are able to inspect the evolution of software systems. In order to detect design flaws of systems, we developed several stand-alone **analyzers** that inspect the persisted *history model*, each targeting a specific **anti-pattern**. We believe these analyzers give meaningful insights about the evolution of systems and which modules require attention. The resulting analysis **reports** can aid in improving the overall quality and maintainability of software. For each analyzer, we look into the design principle it targets, what it means to break the principle and how to automatically detect violations of the principle. Thus, we provide automatic mechanisms to identify modules that should be inspected by the programmer, and possibly refactored.

5.1 Encapsulation Breakers

We will use the definition of **encapsulation** provided by Booch et al.: "the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation" [31]. Encapsulation is an essential mechanism to ensure clients of a module do not depend on implementation details.

Some static analyzers that inspect the syntactic structure of a system, such as Checkstyle [9], are able to detect violations of encapsulation. Typically, they report fields or accessor methods with permissive visibility modifiers. However, this does not always break encapsulation. Consider the pattern of *Data Transfer Objects* described by Fowler: "an object that carries data between processes in order to reduce the number of method calls" [32]. Such an object is a simple data carrier that exposes its fields, and carrying data is its sole purpose. Thus, the typical static analyzer will flag a common and useful pattern as a violation of encapsulation. How, then, can we find a more suitable meaning for "breaking encapsulation"?

5.1.1 Detection Strategy

In order to overcome the limitations of simple static analyzers, we introduce the concept of **decapsulations**, a metric that indicates how many times a module exposed an initially hidden field. Empirically, we set the lower threshold for the number of *decapsulations* to 5 in order to detect an instance of *Encapsulation Breakers*. This means that fields or accessors had their visibility relaxed at least 5 times. In the following sections, we formally define the concept of *decapsulation*, strategy refinements, and an algorithm that extracts the *decapsulations* for each field.

5.1.2 Decapsulations

We believe the key to a better definition of "breaking encapsulation" lies in the history of a module. As in the case of Data Transfer Objects, data is exposed by design. However, if data is initially hidden, and at some point in time, it becomes exposed, we believe that is a case of breaking encapsulation. Thus, we define a **decapsulation** as the process of making a hidden piece of data visible to clients.

For most programming languages, data is stored in fields and is either accessed directly or through accessors (how accessors are implemented differs from language to language — getters and setters in Java, properties in C#, etc.). Field visibility is usually implemented either through language constructs (e.g., visibility modifiers in Java and C#) or coding conventions (e.g., prefixing the field or method name with one or two underscores in Python). Having this in mind, we can detect decapsulations by analyzing whether a field or accessor had its visibility relaxed in a revision (i.e., changed from a more restrictive visibility to a less restrictive one).

5.1.3 Constants

For the purpose of this analysis, **constants** are of little importance. Although the location of constants is important to provide a suitable semantic context and to aid understandability, visibility is less important. Constants are not part of the internal state or interface of modules, and exposing them to clients cannot alter the behavior of any module.

Thus, when analyzing decapsulations, we decided to ignore fields that are explicitly marked as constants, both through language constructs and conventions (e.g., `final` fields in Java that have their names all uppercase with words separated by underscores). Still, the analyzer can inspect constants as well if the special option `--keep-constants` is specified.

5.1.4 Detection Algorithm

Detecting decapsulations is the main goal of the analyzer. Because every programming language has its own specific encapsulation mechanisms, a plugin for each supported language is needed to retrieve the visibility of fields and accessors. Figure 5.1 presents the operations provided by such a plugin.

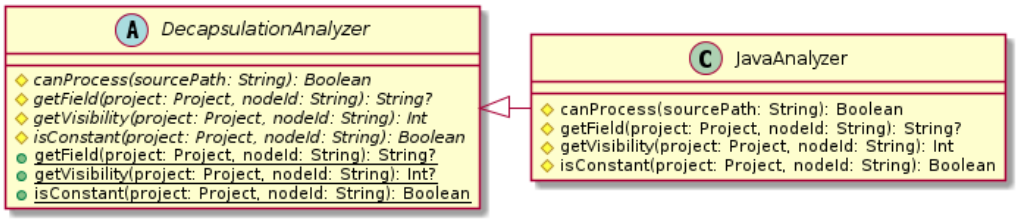


Figure 5.1: UML class diagram of the decapsulation analyzer plugin system

We provide a reference plugin implementation for Java. Visibility modifiers have values assigned as shown in Table 5.1. The smaller the value, the more restrictive the

visibility. Accessors for a field with a specified name `X` are the getter methods `getX` or `isX` and the setter method `setX`. Thus, a decapsulation is detected if the difference between the new visibility value in a revision and the old visibility value of a field or an accessor of a field is positive.

Table 5.1: Values assigned to Java visibility modifiers

Visibility modifier	Assigned value
private	1
package-private	2
protected	3
public	4

The data class that models a decapsulation is presented in Figure 5.2: the field with the specified `fieldId` was decapsulated in the revision with the specified `revisionId` by relaxing the visibility of the source node with the specified `sourceNodeId` (either the field itself or an accessor method); an accompanying `message` explains the context of the decapsulation.

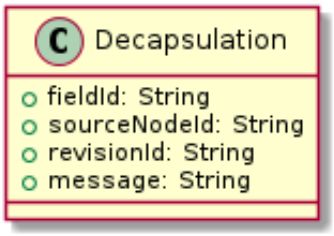


Figure 5.2: UML class diagram of the Decapsulation data class

When inspecting a transaction, the analyzer retrieves the set of edited source nodes and the visibility for every field and method before and after the transaction. Fields are analyzed according to the following listing:

Listing 5.1: Analyzing decapsulations by inspecting the visibility of a field

```
fun analyze(variable: Variable) {
    val old = oldVisibility[variable.id] ?: return
    val new = newVisibility[variable.id] ?: return
    if (new > old) {
        addDecapsulation(
            fieldId = variable.id,
            nodeId = variable.id,
            revisionId = transaction.revisionId,
            message = "Relaxed field visibility!"
        )
    }
}
```

Edited methods are verified to see if they are accessors for an existing field, and if yes, they are inspected for a potential decapsulation. An accessor decapsulates a field if it did not exist before and was added with a more relaxed visibility than the field, or if

the visibility of the accessor itself was relaxed. The analysis of an accessor is described in the following listing:

Listing 5.2: Analyzing decapsulations by inspecting the visibility of accessors

```
fun analyze(function: Function) {
    val fieldId = getField(function.id) ?: return
    val fieldOld = oldVisibility[fieldId] ?: return
    val old = oldVisibility[function.id]
    val new = newVisibility[function.id] ?: return
    if (old == null && new > fieldOld) {
        addDecapsulation(
            fieldId = fieldId,
            nodeId = function.id,
            revisionId = transaction.revisionId,
            message = "Added accessor with more relaxed visibility!"
        )
    } else if (old != null && new > old) {
        addDecapsulation(
            fieldId = fieldId,
            nodeId = function.id,
            revisionId = transaction.revisionId,
            message = "Relaxed accessor visibility!"
        )
    }
}
```

Finally, decapsulations are aggregated for all fields in a source file, as shown in the following listing:

Listing 5.3: Aggregating decapsulations of fields in a source file

```
fun analyze(history: Iterable<Transaction>): Report {
    history.forEach(::analyze)
    val fieldsByFile = decapsulationsByField.keys.groupBy(::getSourcePath)
    val sourcePaths = project.sources.map(SourceFile::path)
    val fileReports = sourcePaths.map { path ->
        val fields = fieldsByFile[path].orEmpty()
        val fieldReports = fields.map { id ->
            FieldReport(id, getDecapsulations(id))
        }.sortedByDescending { it.decapsulations.size }
        FileReport(path, fieldReports)
    }.sortedByDescending(FileReport::value)
    return Report(fileReports)
}
```

Note, however, that the analysis currently has two limitations:

- the reverse of a decapsulation, an "encapsulation" (i.e., the visibility of a field or accessor becomes stricter or an accessor is deleted), is ignored; handling such cases requires more complex logic that spans multiple revisions: if a field was `private`, then it was decapsulated to `public`, and then encapsulated to `protected`, there still is a decapsulation (going from `private` in a previous revision to `protected` in a later revision)

- ChronoLens does not currently detect refactorings; thus, changing the type of a field (e.g., from `int id` to `String id`) would lead to a change in the signature of the setter method (e.g., from `setId(int)` to `setId(String)`), which would be detected as the deletion of a method and addition of a new method, leading to a new decapsulation of the field (an accessor with a more relaxed visibility was seen to be added)

Still, we believe this analysis provides meaningful insights on the evolution of data encapsulation. It aids finding instances in which data was initially hidden and made visible later, and does not flag situations in which data was exposed by design.

5.2 Open-Closed Breakers

The **Open-Closed Principle** as stated by Robert C. Martin, "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification" [3], has two clauses that, combined, can be summed up to: modules should be extendable without being modified. Violating this principle is undesirable and indicates a design flaw. However, it is difficult to assert that a module breaks the principle just by looking at the syntactic structure of the system because the semantic information that captures the design of the system is missing.

In this regard, the evolution of the system can provide the required information to identifty modules that violated the principle in the past (and likely to violate it in the future as well). We say that a module breaks the Open-Closed Principle if it is repeatedly extended through modification. We believe this is the case for two reasons:

- the module cannot be extended without being modified (if it could be, clients would chose that approach because it is usually easier), breaking the first clause of the principle
- the module suffers modifications, breaking the second clause of the principle

In this context, a module can be a method, a field with a long initializer (such as an anonymous class instantiation), class or source file. Assessing that a module was extended through modification is a difficult task that requires objective answers to the following questions, which are subjective in nature:

- what is the threshold that separates the creation of the module from the extension or modification of the module?
- which modifications made to the module are extending it, and which are not (e.g., bug fixes, refactoring)?

In an attempt to answer these questions, we made several assumptions that enable us to quantitatively measure how much a module has been extended through modification:

- the creation of a source entity of a module occurs in the first `T` days after the entity was added to the system, where `T` is a configurable value (e.g., it is unlikely to develop half of a method in the first week after it was added to a class and the other half a month later)

- a module is extended through modification if it is edited by a significant amount (e.g., adding or changing over 50 lines of a method body)
- the behavior of a module is *not* extended in the following cases:
  - code is deleted from the module (e.g., deleting a method from a class or deleting lines from a method)
  - the module is edited by a small amount (e.g., adding or changing only a few lines of a method body)
  - a source entity is added to the module (e.g., adding a method to a class)

We decided to make the last assumption because of the multitude of situations that could be falsely flagged as an extension of the module: the module is complex and is still in its initial development phase, the module was refactored and helper methods were extracted, the module owner decided at a later point in time that the module interface should support new operations, etc.

5.2.1 Detection Strategy

Having these assumptions to reason about the evolution of a system, we extract several metrics for methods and fields, while classes and source files simply aggregate the metrics of their child source nodes. These metrics help detecting violations of the Open-Closed Principle.

The main metric we use is the **weighted code churn**, which measures how much and often a module had modified or added code over time. Empirically, we established a lower threshold of 750 for the *weighted churn* metric to flag an instance of an *Open-Closed Breaker*. This value approximates that a module had at least 300 lines added or modified in the first 10 revisions after the development of the module ended. The greater the *weighted churn*, the more code was added or modified over a longer period of time. Note that deleted code has no influence on the metric.

In the following sections, we describe in detail the concepts and reasoning that led us to define the *weighted churn* metric, a formal definition, as well as a computation algorithm for the metric.

5.2.2 Hotspots

In his book, *Your Code as a Crime Scene* [17], Tornhill defined **hotspots** as "complex parts of the codebase that have changed quickly," and indicates that "the overlap between complexity and effort signals a hotspot." He suggests "using change frequencies as a proxy for effort" and "lines of code as a proxy for software complexity." Therefore, we decided to extract the `size` of a method as the number of lines of code from its body, and the `changes` as the number of revisions in which that method was modified.

While Tornhill's hotspots narrow down the inspection effort to complex modules and track the frequency of change, they tell nothing about the amount of change. Small behavior adjustments are not an actual violation of the Open-Closed Principle because the module is not modified significantly, and not with the goal of extending the module.

5.2.3 Code Churn

We would like to detect modules that have their behavior extended through modification (not just minor corrections of behavior). **Code churn**, another metric that could aid us in this sense, was defined by Nagappan and Ball as "a measure of the amount of code change taking place within a software unit over time" [33].

One of the measures that is of particular interest to us is the *churned lines of code*, which represent the "sum of the added and the changed lines of code between a baseline version and a new version of the files comprising a binary" [33]. This metric indicates how much a module changes over time. Thus, we decided to extract the `churn` of a method as the sum between the number of added or modified lines throughout its history, complementing the previous metrics.

5.2.4 Initial Development

It is possible that a method will change more frequently and have a higher `churn` in the first days after its creation. We denote this phenomenon as **initial development**. Note that changes occurring in this period are not relevant for the Open-Closed Principle: although a module must be closed for modification, creating and implementing the module without making any changes is impossible.

Although dual, each tracing a different aspect of the evolution of a module, neither of the `changes` and `churn` metrics address the issue of initial development. To this end, we use a filter, `--skip-days`, to ignore any modifications that occur in the first specified number of days after the creation of the module (these revisions are ignored by both metrics).

5.2.5 Logarithmically Weighted Churn

We enhance the `churn` metric by applying logarithmically increasing weights to the revisions. Thus, we define the **weighted churn** of a method, in which the churn added in a specific revision is multiplied by a factor of $ln(1 + c)$, where c is the number of times the method already changed (note that the `--skip-days` filter also applies to the computation of the `weightedChurn` metric).

The resulting metric will no longer have a semantic meaning (as in the case of `size`, `churn` or `changes`), but we believe it will better highlight the modules that are extended more frequently and more significantly. Simple `churn` that is distributed in moderate amounts across many changes might not add up to a large value that needs attention from the programmer. However, such a volatile module with frequent changes is clearly undesirable. The `weightedChurn` tries to flag such modules as well and attempts to unify the `churn` and `changes` metrics into a single value.

5.2.6 Detection Algorithm

Detecting Open-Closed Breakers is achieved through a dedicated analyzer. This analyzer inspects the history of the system extracted by ChronoLens and maintains an updated

snapshot of the system. At the same time, it stores information required to compute the metrics for every member (function or variable) in a map. The data class that holds this information is described in Figure 5.3:

- `creationDate` — the date when the associated source node was added to the system
- `revisions` — the set of revisions in which the source node was edited (including the revision in which it was added to the system)
- `changes` — the set of revisions in which the source node was edited after the initial development phase ended
- `churn` — the total number of added or modified lines of the source node across all revisions after the initial development phase ended
- `weightedChurn` — the logarithmically weighted churn of the source node across all revisions after the initial development phase ended

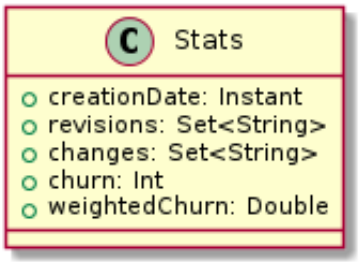


Figure 5.3: UML class diagram of the data class used to compute the metrics

When a member is edited, if it is still in the initial development phase, the revision is added to its set of revisions, leaving the other information unchanged. Otherwise, a certain amount of churn is added and the node's associated data class is updated as follows:

Listing 5.4: Updating the metrics of a source node

```
fun updated(revisionId: String, addedChurn: Int): Stats = copy(
    revisions = revisions + revisionId,
    changes = changes + revisionId,
    churn = churn + addedChurn,
    weightedChurn = weightedChurn + ln(changes.size + 1.0) * addedChurn
)
```

ChronoLens includes only additions and removals of lines in its *delta model* and does not directly support modifications of lines. However, every modified line corresponds to a removed line paired with an added line. Thus, we can compute the `churn` simply as the number of added lines (this will include the lines that were actually added and the lines that were modified as well).

Updating the accummulated information for a member with a certain `id` when a revision with a specific `revisionId` adds an amount of `churn` at a specific `date` is performed according to the following listing:

Listing 5.5: Updating the metrics for individual source entities

```
private fun updateStats(
    id: String,
    revisionId: String,
    date: Instant,
    churn: Int
) {
    val nodeStats = stats.getValue(id)
    val day = nodeStats.creationDate.until(date, DAYS)
    stats[id] =
        if (day < skipDays) nodeStats.updated(revisionId)
        else nodeStats.updated(revisionId, churn)
}
```

After all transactions have been analyzed, the metrics are aggregated at the source file level. All metrics except the `size` are directly available in the stored information for each source node. The `size` of a source node can be retrieved from the latest snapshot of the system:

- the number of lines of a method body in the case of methods
- the number of lines of a field initializer in the case of fields
- the sum of the sizes of member nodes in the case of types
- the sum of the sizes of the contained source entities in the case of source files

Aggregating the `size`, `churn` and `weightedChurn` metrics at the level of source files is achieved by summing up the values of the members. Aggregating the `revisions` and `changes` metrics is achieved by computing the union of the sets of the corresponding revisions of the members.

The results are returned as a report, which presents the source files descending by the value of the desired metric (one of `SIZE`, `REVISIONS`, `CHANGES`, `CHURN` or `WEIGHTED_CHURN`, specified through the `--metric` option). The metric aggregation and report creation are described in the following listing:

Listing 5.6: Aggregating and reporting the collected metrics

```
fun analyze(history: Iterable<Transaction>): Report {
    history.forEach(::visit)
    val membersByFile = stats.keys.groupBy(::getSourcePath)
    val sourcePaths = project.sources.map(SourceFile::path)
    val fileReports = sourcePaths.map { path ->
        val members = membersByFile[path].orEmpty()
        val memberStats = members.map(stats::getValue)
        val memberReports = members
            .map(::getMemberReport)
            .sortedByDescending(::getMemberValue)
        val revisions = memberStats.map(Stats::revisions).union().size
        val changes = memberStats.map(Stats::changes).union().size
        FileReport(path, metric, memberReports, revisions, changes)
    }.sortedByDescending(FileReport::value)
    return Report(fileReports)
}
```

Thus, we can use multiple metrics to detect anomalies in the change patterns of modules, indicating a lack of abstraction in the case of modules that keep changing during the lifetime of the system. The analyzer also provides an overview on how the implementation and maintenance effort is distributed among the methods of a module.

5.3 Single Responsibility Breakers

The **Single Responsibility Principle** is essential for the maintainability of a software system. Although it was originally stated by Robert C. Martin as "a class should have only one reason to change" [3], the term *class* can be replaced with the more general term *module*. For our purpose, by module we will denote *source files* instead of classes. We decided to extend the principle beyond object-oriented software in this manner for several reasons:

- inner classes that are tightly coupled with the containing class are not necessarily separate entities, they are usually used in the implementation of a specific responsibility by the containing class
- many programming languages (e.g., Javascript, Python, C++) combine concepts from multiple programming paradigms (e.g., procedural, object-oriented, functional), allowing the definition not only of classes, but also of top-level functions or variables in source files

Note that for object-oriented languages such as Java and C#, coding conventions require a one-to-one mapping between classes and source files (except for inner classes). Thus, the initial meaning of the principle is retained for these cases.

5.3.1 Detection Strategy

Assessing whether a module violates the Single Responsibility Principle is inherently tied to the history of the module: it must have changed for multiple reasons. This definition is equivalent to the **Divergent Change** code smell described by Fowler et al. [6].

Consequently, we introduce **temporal coupling**, a metric that indicates how often two source entities change together relative to the length of their history. We use this symmetric, bilateral metric between methods to build the **co-change graph** of a source file. In this graph, clusters of tightly coupled methods likely indicate responsibilities. Therefore, we define another metric, the number of **responsibilities**, as the number of clusters identified in the co-change graph. Naturally, a module is a *Single Responsibility Breaker* if the associated number of *responsibilities* is greater than one.

In the following subsections, we present detailed and formal descriptions of the introduced metrics, strategy refinements, as well as algorithms to compute the metrics, graphs and clusters.

5.3.2 Temporal Coupling

An essential concept to our approach for detecting modules that break the Single Responsibility Principle is **change coupling**, also known as **temporal coupling** (as noted by Tornhill in his book [17]), defined by D'Ambros, Lanza and Robbes as "the implicit and evolutionary dependency of two software artifacts that have been observed to frequently change together during the evolution of a software system" [16].

For the purpose of our analysis, we inspect the temporal coupling between methods in a source file. We believe that methods related to the same responsibility tend to change together, in the same revisions, while methods that implement different responsibilities do not. Palomba et al. also conjectured "that classes affected by Divergent Change present different sets of methods each one containing methods changing together but independently from methods in the other sets" [18], but we have a different detection strategy.

Although multiple measures to compute temporal coupling have been proposed, we chose the *Jaccard similarity coefficient* (originally coined by Jaccard as *coefficient de communauté* [34]) for its simplicity and ease of computation. Let M_1 and M_2 be two methods and $R(M)$ be the set of revisions in which method M was changed. Then, we define the temporal coupling C between M_1 and M_2 as follows:

$$C = \frac{|R(M_1) \cap R(M_2)|}{|R(M_1) \cup R(M_2)|}$$

Thus, the temporal coupling is a fractional number between 0 and 1 that indicates how often the two methods changed together. A coupling of 0 means the two methods never changed together and a coupling of 1 means the two methods always changed together. Note that it is possible for a method to have always changed together with another method, but to have coupling with it less than 1. This can happen if the method was created after the other method had already been modified multiple times. However, we believe that this is desirable: if a method did not exist for a part of the history of another method, it is not necessarily a part of the same responsibility. If, afterwards, the two methods systematically change together, their temporal coupling will increase, indicating that even if they were not created at the same time, they are still likely related to the same responsibility.

5.3.3 Method Clusters as Responsibilities

Using temporal coupling, we can state that two methods belong to the same responsibility if they are strongly coupled. But what is a responsibility? Robert C. Martin defined it as "a reason for change" [3], but that is a subjective concept and requires the semantic understanding of the system.

In the context of the syntactic and historical information extracted by ChronoLens, we define a **responsibility** of a module as a cluster of methods that have a strong temporal coupling with each other. We named this cluster of methods a **blob**. It is also possible for a module to have a set of methods that are very weakly coupled with the rest of the module (e.g., short utility methods), which we assign to a virtual "utility" responsibility. We named this set of unrelated methods an **anti-blob**. The remaining methods, which

we name **transition methods**, are not sufficiently coupled to be part of any blob, but are sufficiently coupled to not be part of the anti-blob. We do not assign transition methods to a virtual responsibility, as in the case of the anti-blob, because they most likely belong to an existing responsibility, but couldn't be included in it for various reasons. Thus, modules are split into several sets of methods: a possibly empty anti-blob, a possibly empty transition area, and zero or more non-empty blobs.

Having this topological decomposition of a module, we can assess whether the Single Responsibility Principle is violated by counting the number of non-empty sets of methods in which the module is split. If the module contains more than one such set, its methods implement more than one responsibility.

Decomposing a module into blobs, transition methods and an anti-blob requires formal definitions for these concepts. To this end, we interpret the module as a weighted, undirected graph, where nodes represent its methods and the weight of an edge represents the temporal coupling between the two methods it connects. We define the *sum of coupling* of a method as the sum of the weights of edges incident to its associated node. We call this graph the **module co-change graph**, or simply *module graph*.

A *blob* is a set of methods with the average sum of coupling in the subgraph induced by the associated nodes above a certain threshold `--min-blob-density`[1]. An *anti-blob* is a set of at least `--min-blob-size` methods that have the sum of coupling below a certain threshold `--max-anti-coupling`. A *transition method* is a method that does not belong to any blob or to the anti-blob of the module.

The quality of the decomposition can be improved with two more heuristic filters: `--min-revisions`, which indicates the minimum number of revisions for a node or edge to be a part of the module graph, and `--min-coupling`, which indicates the minimum weight of an edge to be a part of the module graph. Therefore, methods with a very short history (e.g., only one or two revisions) and edges that could be a coincidence (a pair of methods that changed together only once or twice) are filtered out from the graph.

5.3.4 Detection Algorithm

The detection algorithm works in three steps: compute the temporal coupling between methods, aggregate the collected data and build the module graphs, and decompose the graphs in blobs, anti-blobs and transition methods.

The temporal coupling C between two methods M_1 and M_1 can be computed if we count the number of revisions that changed each method individually ($|R(M_1)|$ and $|R(M_2)|$) and the number of revisions in which both methods were changed ($|R(M_1) \cap R(M_2)|$). The missing term from the defintion of C can be computed as follows:

$$|R(M_1) \cup R(M_2)| = |R(M_1)| + |R(M_2)| - |R(M_1) \cap R(M_2)|$$

Thus, the analyzer will inspect every transaction from the history of the system, updating the `changes` (number of individual revisions of a method) and `jointChanges` (number of revisions in which a pair of methods was modified together) as follows:

[1] A responsibility translates to a dense subgraph in the module graph, for a certain definition of density. Yanagisawa and Hara [35] explored various graph density metrics. After analyzing the properties of each metric, we decided to use the average degree of a node, which is the average sum of coupling in our case.

Listing 5.7: Updating the temporal coupling induced by a transaction

```
private fun analyze(transaction: Transaction) {
    val editedIds = visit(transaction)
    for (id in editedIds) {
        changes[id] = (changes[id] ?: 0) + 1
    }
    if (transaction.changeSet.size > maxChangeSet) return
    for (id1 in editedIds) {
        for (id2 in editedIds) {
            if (id1 == id2) continue
            val jointChangesWithId1 = jointChanges.getOrPut(id1, ::HashMap)
            jointChangesWithId1[id2] = (jointChangesWithId1[id2] ?: 0) + 1
        }
    }
}
```

Note that another another filter was used, `--max-change-set`, representing the maximum number of files that can be modified in a single revision to consider it towards temporal coupling. This not only limits the maximum computational effort for a single transaction, improving the performance, but also helps filter out irrelevant revisions when computing temporal coupling (it is unlikely to modify more than 100 files in a single revision; usually, this is the case when sources are automatically generated or a a different formatting is enforced in the project, e.g., the number of spaces used for indentation changed).

Module graphs are modeled by several data classes described in Figure 5.4. Every graph has a `label` equal to the path of the source file (module), a set of `nodes` and a list of `edges`. Nodes consist of a `label` equal to the `id` of the associated method and a number of `revisions` in which the method was changed. Edges have a `source` and `target` consisting of the labels of the nodes it connects, a number of `revisions` in which the two associated methods changed together and the temporal `coupling` between them. Subgraphs used to represent blobs and anti-blobs have a set of labels correspoding to the subset of `nodes` that induced the subgraph, as well as a `density` (average weighted degree of a node) and `size` (number of nodes in the subgraph).

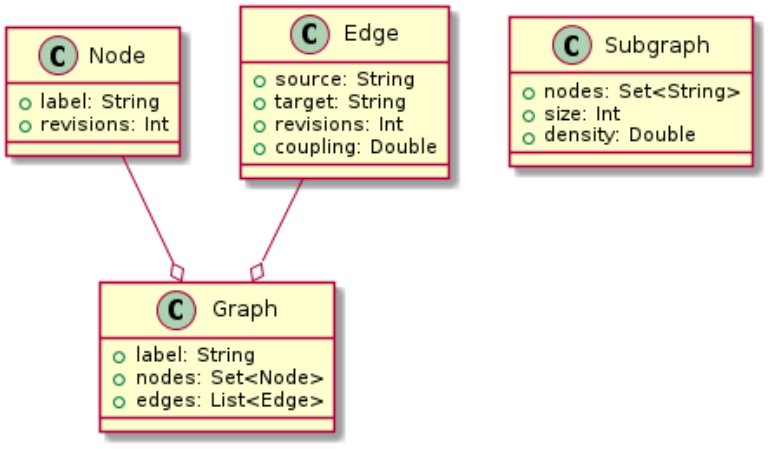


Figure 5.4: UML class diagram of the graph data structures

Aggregating the computed temporal couplings into module graphs is straightforward: group all methods by source files (thus extracting the nodes of the module graphs), retrieve the temporal coupling for all pairs of methods that belong to the same source file (thus extracting the edges of the module graphs) and apply the previously defined filters to keep only the relevant nodes and edges. This process is described in the following listing:

Listing 5.8: Computing the coupling graphs

```
private fun aggregate(): List<Graph> {
    val idsByFile = changes.keys.groupBy(::getSourcePath)
    val sourcePaths = project.sources.map(SourceFile::path)
    return sourcePaths.map { path ->
        val ids = idsByFile[path].orEmpty()
        val edges = ids.flatMap { id1 ->
            jointChanges[id1].orEmpty()
                .filter { (id2, _) -> id1 < id2 }
                .map { (id2, revisions) -> makeEdge(id1, id2, revisions) }
        }
        val nodes = (ids + edges.getEndpoints())
            .map { id -> Node(id, changes.getValue(id)) }
            .toSet()
        Graph(path, nodes, edges)
            .filterNodes(::takeNode)
            .filterEdges(::takeEdge)
    }
}
```

Decomposing a module graph into blobs requires identifying subgraphs (or clusters) of strongly coupled methods. Multiple such clusterings exist, some more suitable than others, and finding the fittest clustering is a complex task that requires a formal definition of a cluster, of the fittest clustering, etc. Balalau et al. [36] proposed an algorithm that allows computing K disjont subgraphs that maximize the total density of the subgraphs, where K is a chosen parameter. However, their approach is computationally intensive and provides an exact solution (i.e., no other partitioning into subgraphs can achieve a greater total density), whereas in our case, we do not need the best partitioning possible.

Charikar [37] presented a greedy algorithm that finds a subgraph with density no less than $\rho_{max}/2$, where ρ_{max} is the maximum density of any subgraph. In our case, we use this algorithm to find a dense subgraph, remove it from the graph and repeat the procedure as long as the density of the found subgraph is greater than the specified threshold `--min-blob-density`. The subgraphs extracted in this manner will constitute the blobs. This approach is more efficient timewise and produces suitable clusterings for our purpose.

Having all the module graphs available, this analysis detects modules that appear to fulfill multiple responsibilities and provides a hint on how to refactor the modules based on how responsibilities are distributed among the methods, showing the temporal coupling between them.

6 Usage. Experimental Results

6.1 Usage

6.1.1 Command-Line Interface

The developed tool and analyzers expose a simple **command-line interface** and interact with the repository detected in the current working directory. They are packaged and released as a Java application distribution.[1]

The ChronoLens tool connects to the repository through one of the version control system proxies and provides several commands:

- `help` — prints usage information
- `ls-tree` — prints the source files found in the specified *revision* that can be interpreted by one of the packaged parsers; listing these files indicates which files are analyzed and which are ignored
- `rev-list` — prints the ancestor revisions of the `head` revision in chronological order; listing these revisions helps follow the actual history that is being analyzed
- `model` — prints the model of the source node with the specified *id*, as it is found in the given *revision*; this aids debugging and understanding the results of certain analyses
- `persist` — writes the source and history model of the repository to disk for other analyzers to inspect; this greatly improves performance when multiple analyses are performed, because all the computationally intensive algorithms and VCS proxy subprocesses are executed only once
- `clean` — deletes the previously persisted history model

The specialized analyzers load the persisted history model from the current working directory and print to the standard output a JSON report containing the results. The various filters used by the analyses (e.g., the minimum temporal coupling between two methods) can be specified as options (e.g., `--min-coupling=0.25`). The analyzers were bundled together with the tool as separate commands:

- `decapsulations` — runs the *Encapsulation Breakers* analysis and prints a report containing the decapsulations that occurred throughout the history of the system
- `churners` — runs the *Open-Closed Breakers* analysis and prints a report containing the high-churn modules of the system
- `divergent-change` — runs the *Single Responsibility Breakers* analysis and prints a report containing the *Divergent Change* instances

[1] The application distribution can be downloaded from the following link: https://github.com/andreihh/chronolens/releases.

To showcase the various commands, consider the following scenario: Alice wishes to inspect the decapsulations that occurred in the *Apache Kafka* open-source project developed by the Apache Software Foundation. To this end, she runs the following commands:

Listing 6.1: Apache Kafka analysis commands

```
# Clone the remote repository into the current working directory.
git clone https://github.com/apache/kafka.git .

# Persist the history model corresponding to the repository root. The version
# control system is detected automatically as 'git'.
chronolens persist

# Run the 'Encapsulation Breakers' analysis and output the resulting report to
# the specified file.
chronolens decapsulations > kafka-decapsulations.json

# Manually inspect the analysis results.
cat kafka-decapsulations.json | less

# Delete the persisted history model from the current working directory.
chronolens clean
```

While looking at the analysis report, she notices that the `partitioner` field from the `Produced.java` source file suffered a decapsulation in the revision with the short id `779714c`. In order to understand the context and what triggered the decapsulation, Alice can inspect the source model of the decapsulated field before and after the decapsulation occurred with the following commands:

Listing 6.2: Investigating decapsulations in Apache Kafka

```
# Find the previous revision relative to the revision when the detected
# decapsulation occurred. The command below will print
# 'c5464edbb7a6821e0a91a3712b1fe2fd92a22d68'.
chronolens rev-list | grep -B 1 "779714c08bc16fcdd6fe7c39e92a7f73ebebdb71"

# Visualize the model of the field before the detected decapsulation occurred.
chronolens model \
    --id "streams/src/main/java/org/apache/kafka/streams/kstream/Produced.java\
:Produced#partitioner" \
    --rev "c5464edbb7a6821e0a91a3712b1fe2fd92a22d68"

# Visualize the model of the field after the detected decapsulation occurred.
chronolens model \
    --id "streams/src/main/java/org/apache/kafka/streams/kstream/Produced.java\
:Produced#partitioner" \
    --rev "779714c08bc16fcdd6fe7c39e92a7f73ebebdb71"
```

With the presented debugging commands, Alice saw that the field changed its visibility modifier from `private` to `protected`, which triggered the decapsulation.

6.1.2 Data Visualization

The analyzers produce **JSON reports** containing the detected anti-patterns in a specific format described in Listing 6.3. The report consists of a list of file records describing

the affected `file` path, the `category` of anti-patterns, the specific `name` of the detected anti-pattern, and the `value` for the metric used to detect it.

Listing 6.3: JSON report template produced by the analyzers

```
[
  {
    "file": <path>,
    "category": <anti-pattern-category>
    "name": <anti-pattern>,
    "value": <metric-value>,
    ...
  },
  ...
]
```

Additionally, records may contain more fine-grained information about the file, depending on the performed analysis:

- *Encapsulation Breakers* — besides the nominal `decapsulations` metric field (equal to the `value` field), the record contains a list of `fields` with their associated decapsulations, in the same data format as described by Figure 5.2
- *Open-Closed Breakers* — the record contains the nominal metric fields `revisions`, `size`, `changes`, `churn`, and `weightedChurn`, as well as a list of `members` (functions or variables), each having the same nominal metric fields with their corresponding values; the `value` field is equal to one of the nominal metric fields, depending on the provided `--metric` option (which is also stored in the `metric` field of the report)
- *Single Responsibility Breakers* — besides the nominal `responsibilities` metric field (equal to the `value` field), the record contains a list of `blobs` and the `anti-blob`, each describing a cluster of methods in the `Subgraph` format presented in Figure 5.4

Although the JSON reports generated by the analyzers are readable and provide detailed information about a specific anti-pattern instance, it is difficult for a user to grasp the state of the entire system. For a broader view of the system based on our analyses, we use Chronos[2], a commercial tool developed by Endava.

Chronos parses the VCS history of a system and JSON reports in the same format as those produced by our analyzers and builds a **tree-map**[3] of the system based on the reported metric values. Nodes in the tree-map have the reactangle area proportional to the number of lines of code in the associated file and are colored according to two configurable threshold values for the metric, `lower` and `upper`:

- a node is white (ignored) if there is no metric value provided (e.g., it was filtered out by the `--min-metric-value` option or the associated file contains source code written in an unsupported programming language)
- a node is blue (retired) if more than six weeks passed from the last modification

[2] Chronos was described in *Quality Assessment in the Tower of Babel*, a presentation given by Radu Marinescu at *ITCamp 2016* [38].

[3] Shneiderman described tree-maps as "a representation designed for human visualization of complex traditional tree structures: arbitrary trees are shown with a 2-d space-filling representation" [39].

- a node is green (active) if it was modified in the last six weeks
- if a node is blue or green, it has assigned one of three shades: a lighter shade if its metric value is less than the `lower` threshold, a medium shade if its metric values is between `lower` and `upper` and a darker shade if its metric value is greater than `upper`

Consider the tree-map produced by CHRONOS after running the *Open-Closed Breakers* analysis with the *weighted churn* metric on the Apache Kafka project. The resulting image is shown in Figure 6.2. The used thresholds are `lower = 200` and `upper = 500`. Hovering the cursor above a specific rectangle will show a tooltip with the corresponding file name. Thus, users have a broad view of the state of the system.

The *Single Responsibility Breakers* analysis provides more information than the simple value of a metric, it provides the topological decomposition of a module co-change graph into responsibilities.

We implemented a visualization method for module graphs that makes use of the **force layouts** from the *D3.js* third-party library [40]. Unique colors are assigned to the nodes in each blob, in the anti-blob, and those corresponding to transition methods. The stronger the temporal coupling between two nodes, the thicker and shorter the edge connecting them is. Hovering the cursor above a certain node will show a tooltip with the id of the corresponding method.

Figure 6.1 presents the force layout for the module graph corresponding to the `ServletContextHandler.java` file from the *Jetty* open-source project developed by the Eclipse Foundation. Orange nodes form a blob, the light-blue node corresponds to a transition method, and the dark-blue nodes are part of the anti-blob.

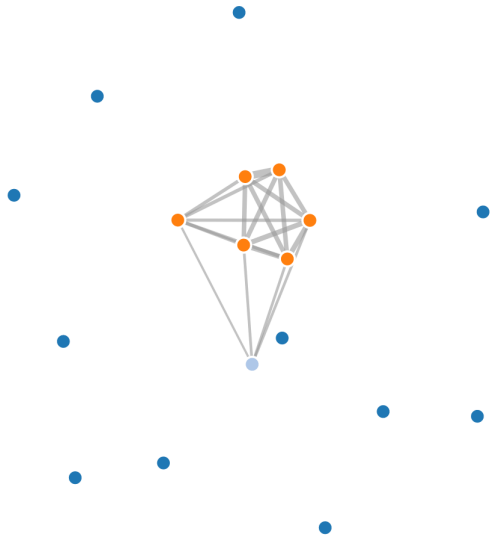


Figure 6.1: D3 force layout of the Jetty ServletContextHandler module graph

CHRONOS tree-maps highlight the modules with multiple responsibilities, whereas D3 force layouts of module graphs provide a detailed view of the temporal coupling between methods and the topology of responsibilities. Thus, users have a complete picture of both the system as a whole and of each module by itself.

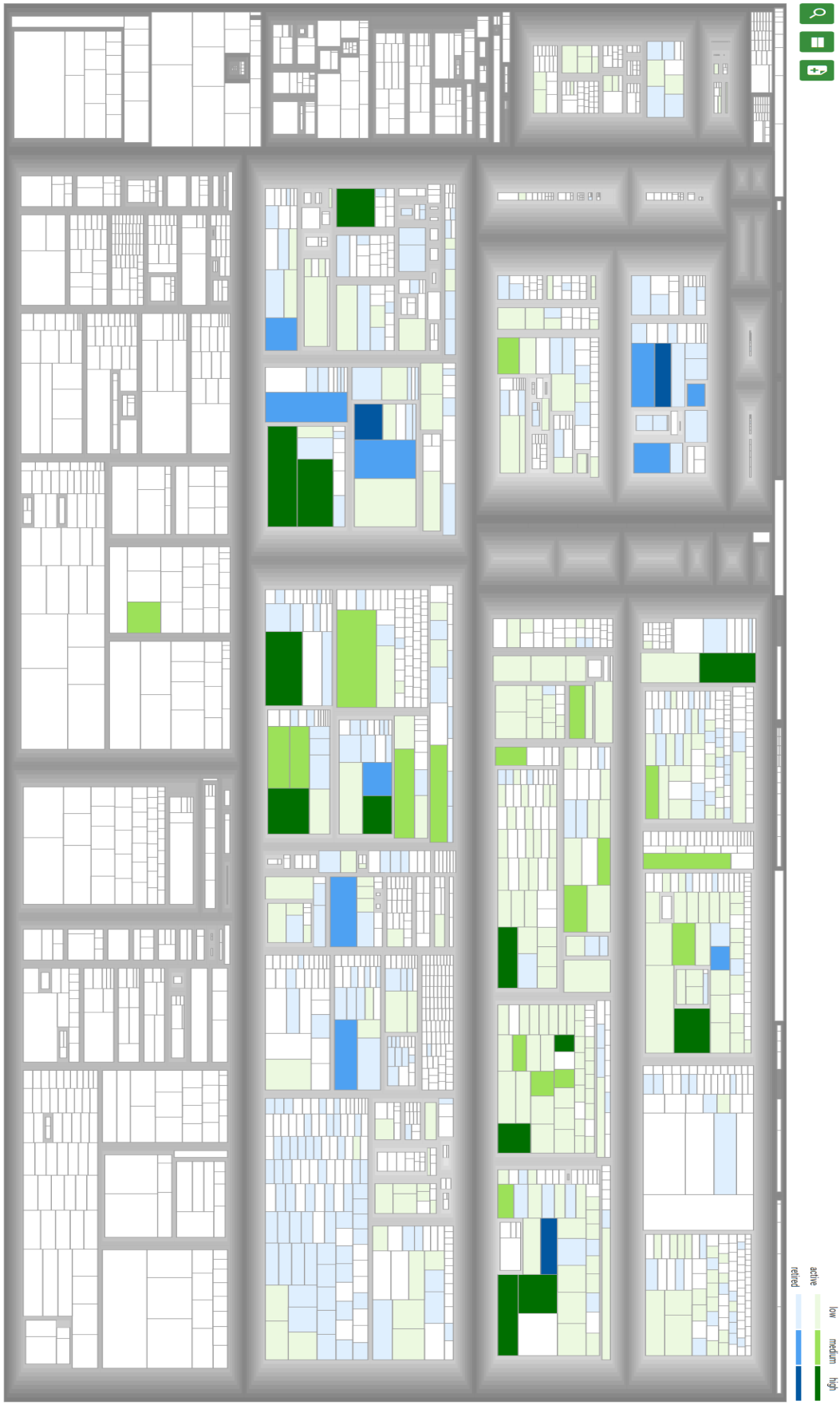


Figure 6.2: CHRONOS tree-map of Apache Kafka Open-Closed Breakers

6.2 Experimental Results

We analyzed several medium and large open-source projects written in *Java* and stored in *Git*. Table 6.1 presents the main characteristics of the chosen systems. We extracted the number of lines of Java code using the CLOC tool [41] and the number of analyzed revisions from the persisted history model. We included in the table the short hash of the analyzed `head` revision to uniquely identify the considered system snapshot and history.

In the case of individual source files, we retrieved the number of revisions that modified it with the following command:

Listing 6.4: Retrieving the number of revisions

```
1 git rev-list --first-parent HEAD -- "$path" | wc -l
```

Table 6.1: Analyzed systems' characteristics

System	Lines of Java code	Revisions	Head short hash
Kafka	205117	4404	564311f
Guava	339224	4696	1477e56
Groovy	183842	12369	5443e87
Jetty	361742	6600	d25fa7d
Tomcat	313919	18898	8ad7db3
Cassandra	350041	11063	42827e6
Spring	600951	14428	795e5d3
Wildfly	490480	17914	13c786e
SWT	375905	25936	a29eed0
Elasticsearch	1045629	16232	eed8a3b
Hadoop	1500509	15581	e87be8a

In order to highlight only relevant source files in CHRONOS tree-maps, we set the `--min-metric-value` filter to 1 for all the performed analyses (i.e., files with no decapsulations, zero weighted churn or no detected responsibilities will not be highlighted).

6.2.1 Encapsulation Breakers

In the case of Encapsulation Breakers, two systems stand out by the most number decapsulations for a single module: Tomcat, and Hadoop. We set the CHRONOS thresholds to `lower = 5` and `upper = 10`.

Figure 6.3 presents the resulting CHRONOS tree-map after analyzing Tomcat, and Table 6.2 lists the source files with the most decapsulations.

The top "offender" is the `JspC.java` source file, a rather large file with a long history, but not as large or long-living as other files. The `trimSpaces` field has three decapsulations: one relaxed the field visibility from `private` to `protected`, and two false-positives when the type of the field changed from `boolean` to `String` and from `String` to `TrimSpacesOption`. All other decapsulations occurred in the same revision, each relaxing the visibility modifier of a field from `private` to `protected`. The `StandardWrapper.java` file has 26 decapsulated fields, all in a single revision, having their visibility modifiers relaxed from `private` to `protected`. All decapsulations of `JspCompilationContext.java`

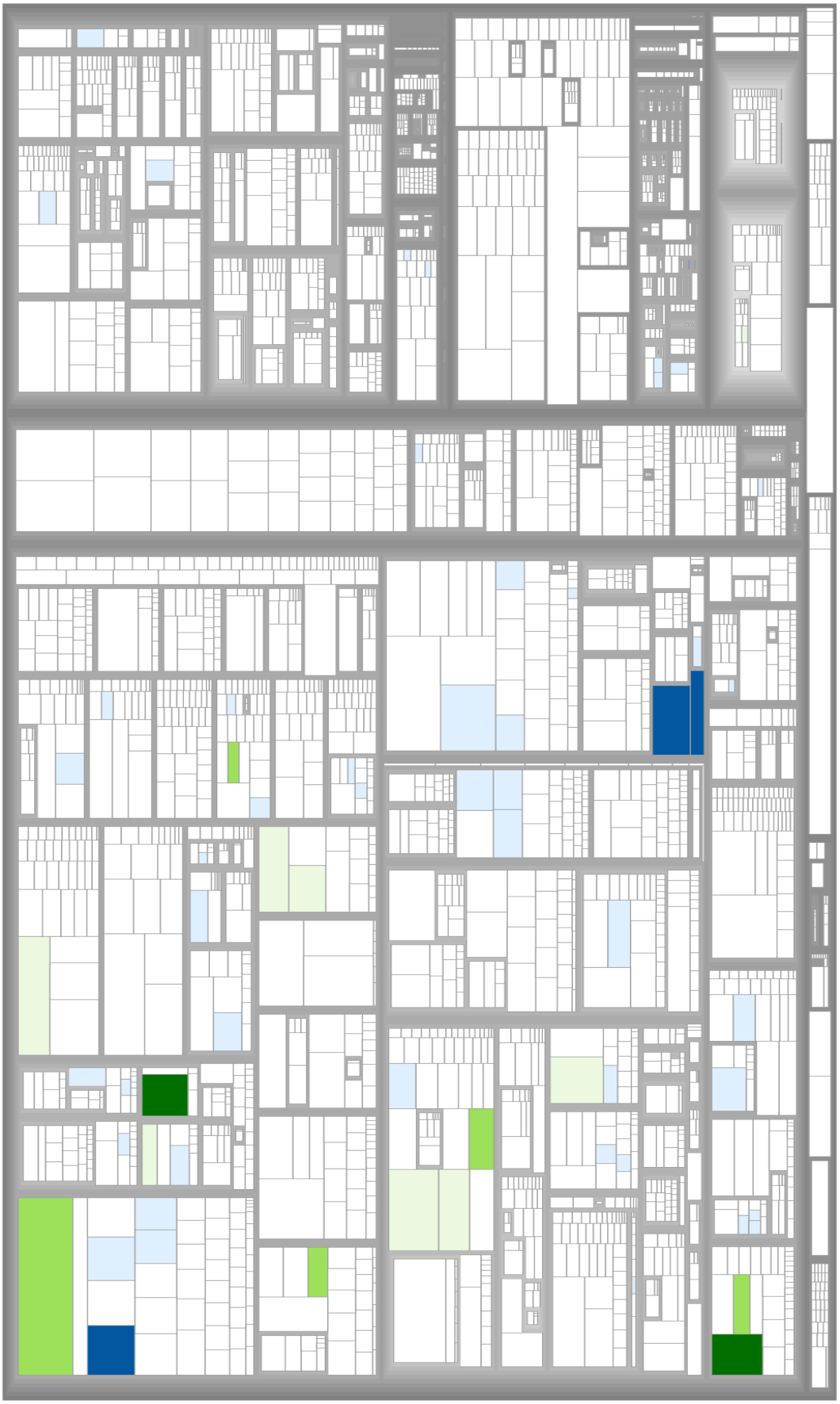


Figure 6.3: CHRONOS tree-map of Tomcat Encapsulation Breakers

Table 6.2: Tomcat top Encapsulation Breakers

Module	Decapsulations	Lines of code	Revisions
JspC.java	47	1758	86
StandardWrapper.java	26	1753	127
JspCompilationContext.java	25	770	70
Http2UpgradeHandler.java	20	1597	219
AbstractReplicatedMap.java	17	1640	134
Http2Parser.java	8	749	72
SocketWrapperBase.java	7	1077	97
AccessLogValve.java	5	704	110
StandardContext.java	5	6634	381
NumberGuessBean.java	5	99	4
AbstractArchiveResourceSet.java	5	337	20

also occurred in a single revision. The `derivedPackageName` field had its visibility modifier relaxed from `private` to `protected` and had the visibility modifier of the associated getter method relaxed from `private` to `protected`. All the other fields had a single decapsulation consisting of relaxing the visibility modifier from `private` to `protected`.

This pattern suggests three possible reasons for the decapsulations: preparing the classes for inheritance in the future, instantiating anonymous classes, or accessing their fields from other classes within the same package. Considering that none of `JspC` and `JspCompilationContext` have any subclasses in the latest revision and that the first subclass of `StandardWrapper` was added more than one year later since the decapsulations, the first considered reason was either not the cause of the decapsulations or a case of over-abstraction. The three files are quite large, two of them having over 1700 lines of code including comments and white spaces. Thus, understanding the class and its contracts requires significant effort, leading us to the conclusion that extension through anonymous classes was either a bad design decision or not the cause of the decapsulations. The third considered cause is a clear case of breaking encapsulation, but is difficult to probe, as it requires tracking references to the decapsulated fields across multiple revisions.

The case of `Http2UpgradeHandler.java` is also interesting. The `pingManager` field has two decapsulations in the same revision: its visibility modifier was relaxed from `private` to `protected` and a `protected` getter method was added. The `protocol` field had its visibility modifier relaxed from `private` to `protected` in another revision. All the other fields had their visibility modifier relaxed from `private` to `protected` in the same revision. However, unlike the previously analyzed source files, the `Http2UpgradeHandler` had a subclass added in the same revision as most of the decapsulations. Thus, in this case, the decapsulations occurred because a class that was not originally designed for inheritance was extended, confirming the idea that inheritance breaks encapsulation.

In this system, all of the decapsulations occurred in the first two years after the corresponding file was created.

Figure 6.4 presents the resulting CHRONOS tree-map after analyzing Hadoop, and Table 6.3 lists the source files with the most decapsulations.

All decapsulations in `ClusterMetricsInfo.java` occurred in the same revision. Each decapsulated field had its visibility modifier changed from `private` to `protected` and

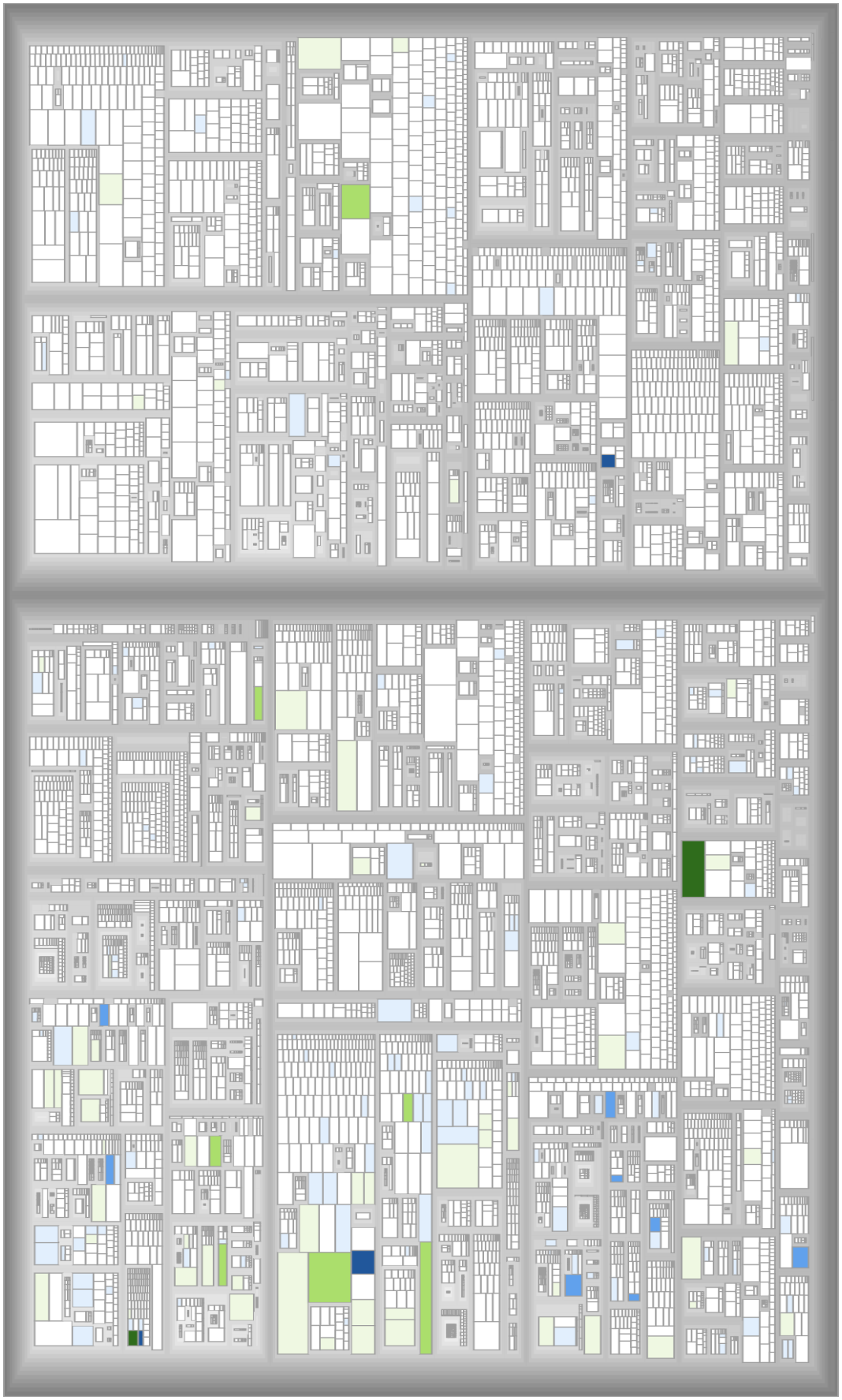


Figure 6.4: CHRONOS tree-map of Hadoop Encapsulation Breakers

Table 6.3: Hadoop top Encapsulation Breakers

Module	Decapsulations	Lines of code	Revisions
ClusterMetricsInfo.java	25	337	14
NamenodeFsck.java	18	1424	96
AppInfo.java	17	629	38
LoadGenerator.java	11	747	9
Server.java	10	3512	132
RMContainerAllocator.java	8	1656	75
BlockChecksumHelper.java	8	537	7
ApplicationMaster.java	7	1737	92
ContainerLaunch.java	7	1843	74
InputSampler.java	7	415	7
Fetcher.java	7	761	35

had a `public` setter method added. The decapsulations occurred five years after the creation of the file. In the case of `NamenodeFsck.java`, all decapsulations occurred in the same revision: the fields of the `private` nested class `Result`, along with the class itself, had their visibility modifiers relaxed from `private` to `package-private`. Moreover, the `@VisibleForTesting` annotation, which makes a `package-private` source entity visible to tests, was added to the nested class. This indicates that the decapsulations occurred for testing purposes. The `AppInfo.java` decapsulation pattern is almost identical to that of `ClusterMetricsInfo.java`.

Having analyzed these systems, there is a recurring pattern among the top offenders: most fields are decapsulated in a single revision, usually having their `private` visibility modifiers relaxed.

6.2.2 Open-Closed Breakers

In the case of Open-Closed Breakers, several systems have an exceptionally high total churn, but we will take a closer look at two of them: Hadoop and SWT. We set the CHRONOS thresholds to `lower = 750` and `upper = 1500`. For this particular analysis, we compute the number of `revisions` for each file differently than for the other analyses: it is provided by the analyzer, but only revisions that bring detectable changes to the *source model* of functions and variables. Therefore, changes to, e.g., documentation comments, or the set of implemented interfaces by a class, are ignored. To avoid confusion, we specify the number of such revisions as *effective revisions*. The `changes` metric is also provided by the analyzer and represents the number of revisions that modified the source model of functions or variables after their initial development phase ended. We decided to set the default value of the `--skip-days` filter to 14 days, as it is a common length of a development iteration.

Figure 6.5 presents the resulting CHRONOS tree-map after analyzing Hadoop, and Table 6.4 lists the source files with the highest weighted churn.

The total weighted churn of the entire system is 186939. Just 52 files out of over 9000 analyzed files have a weighted churn of over 750, and they account for over 35% of the total weighted churn of the system. The weighted churn to churn ratio varies between 1.5

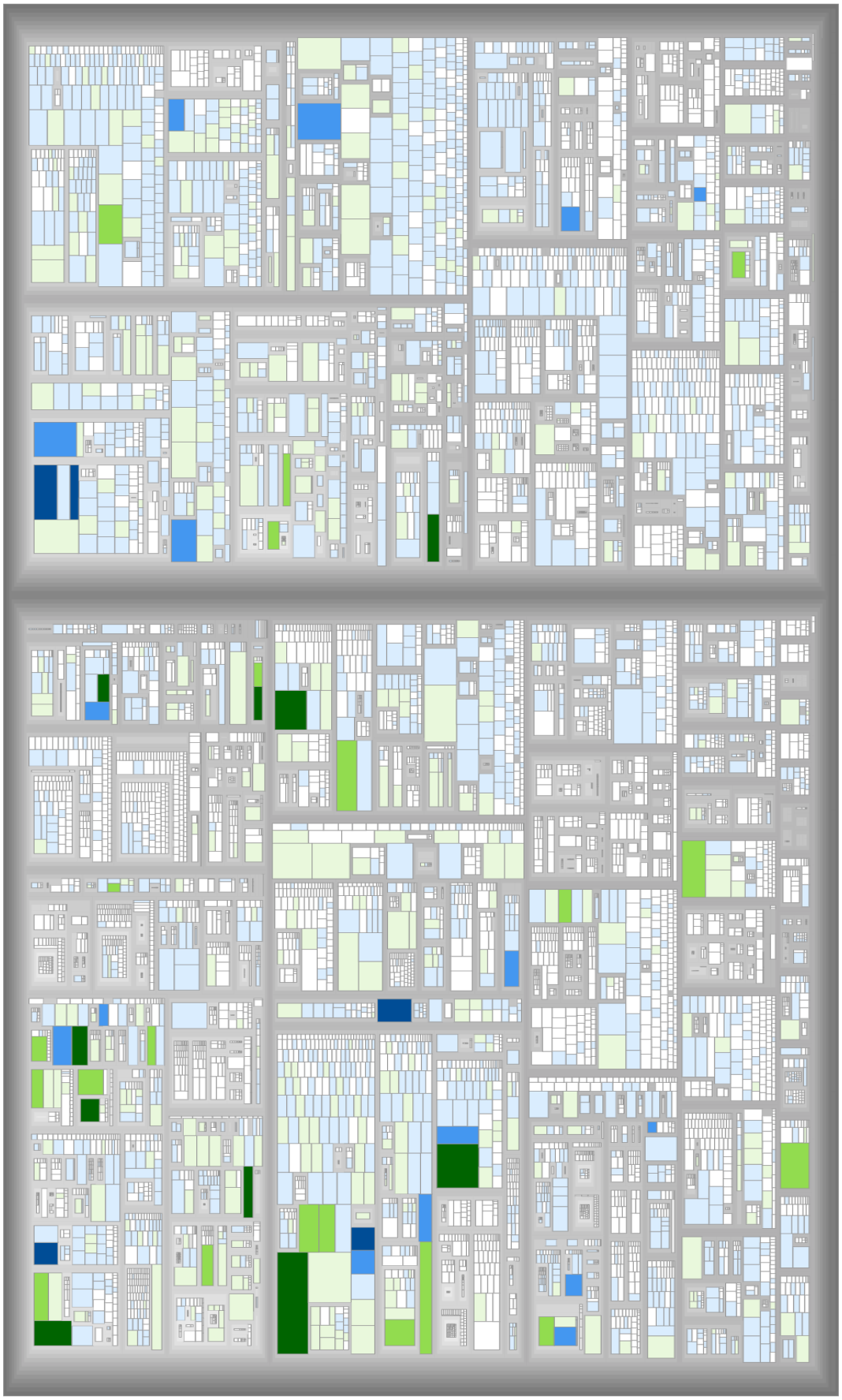


Figure 6.5: CHRONOS tree-map of Hadoop Open-Closed Breakers

Table 6.4: Hadoop top Open-Closed Breakers

Module	Weighted churn	Churn	Lines of code	Changes	Effective revisions
FSNamesystem.java	4535	3212	8031	361	426 (594)
TestLeafQueue.java	3934	1948	4131	76	87 (92)
CapacityScheduler.java	2055	1236	3018	111	124 (168)
FSEditLogLoader.java	1662	504	1385	84	91 (143)
BlockManager.java	1652	1398	4975	166	205 (337)
TestReservations.java	1615	904	1587	29	30 (32)
S3AFileSystem.java	1610	1176	3322	74	81 (90)
TestYarnCLI.java	1586	1264	2345	68	74 (76)
FairScheduler.java	1584	1322	1864	128	148 (197)
NodeStatusUpdaterImpl.java	1577	760	1263	79	83 (93)
DFSAdmin.java	1566	1003	2473	85	87 (96)

and 3, indicating that most of the modifications were performed in the first 15 revisions after the initial development phase of each method ended.

The top offender, `FSNamesystem.java`, is exceptionally large, having over 8000 lines, and has an exceptionally long history with over 400 revisions. More than 50% of the weighted churn is located in only 15 methods, out of a total of nearly 500 methods. The same 15 methods account for more than 30% of the churn. Less than 50 methods have a weighted churn greater than 25, whereas the top churner method has a weighted churn of 601 and a churn of 220.

The `TestLeafQueue.java` test file has a similar distribution of churn and weighted churn: 11 methods account for over 75% of the weighted churn and 70% of the churn, out of a total of 70 methods. Only 20 methods have a weighted churn greater than 25, whereas the top churner method has a weighted churn of 518 and a churn of 204.

In the case of `CapacityScheduler.java`, only two methods account for almost 50% of the total weighted churn and 30% of the total churn. Out of the total of 132 methods, only 17 have a weighted churn of over 25.

The `FSEditLogLoader.java` file is particularly interesting: its top churner method has a weighted churn of 1648 and a churn of 477, accounting for over 94% of the total weighted churn and churn. The other 41 methods have a weighted churn of less than 5.

The three previously inspected files are large, having over a thousand lines of code, and a long history of about 100 revisions.

Figure 6.6 presents the resulting CHRONOS tree-map after analyzing Tomcat, and Table 6.5 lists the source files with the highest weighted churn.

The total weighted churn of the entire system is 81741. Just 19 files out of over 2000 analyzed files have a weighted churn of over 750, and they account for over 35% of the total weighted churn of the system. Again, the weighted churn to churn ratio varies between 1 and 2.5, indicating that most of the modifications occurred in the first 15 revisions.

The `AprEndpoint.java` has 159 methods, out of which only 16 have a weighted churn of over 25. Only five methods account for over 80% of the total weighted churn and over 65% of the total churn. Its top churner method alone has a weighted churn of 1796 and a churn of 564.

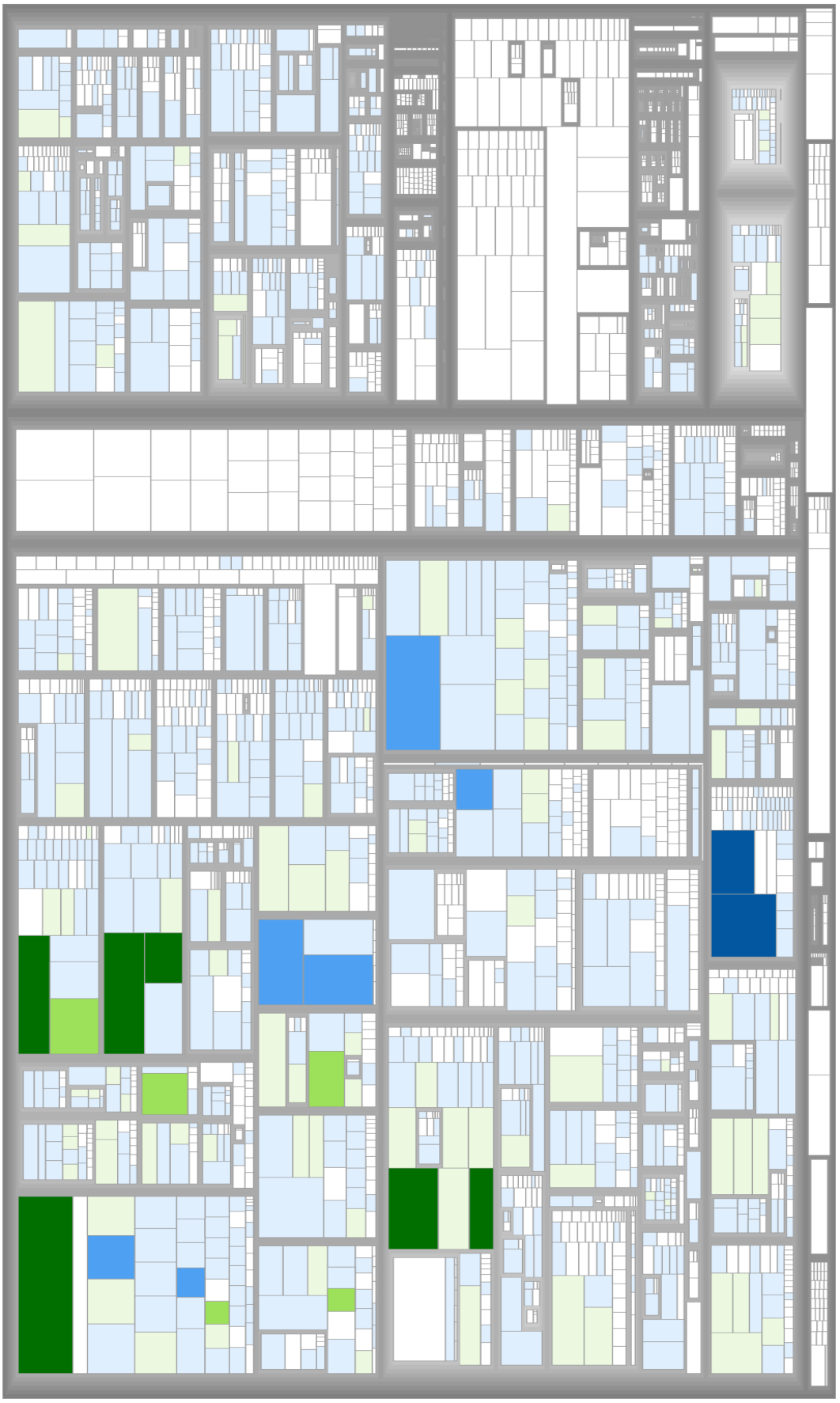


Figure 6.6: CHRONOS tree-map of Tomcat Open-Closed Breakers

Table 6.5: Tomcat top Open-Closed Breakers

Module	Weighted churn	Churn	Lines of code	Changes	Effective revisions
AprEndpoint.java	4382	1900	2931	242	277 (436)
ELParser.java	2670	2019	3143	23	26 (31)
ELParserTokenManager.java	2406	2067	2134	15	16 (23)
CoyoteAdapter.java	2064	796	1312	155	171 (256)
StandardContext.java	1815	1254	6634	235	287 (381)
NioEndpoint.java	1738	920	1418	213	244 (469)
Request.java	1635	1157	3482	184	207 (285)
ContextConfig.java	1634	929	2671	171	182 (294)

The `ELParser.java` and `ELParserTokenManager.java` files, unlike the rest of the offenders, have a short history of about 25 revisions. However, they are uninteresting for the purpose of our analysis, because the comment placed on the first line in each of these files indicates that they are automatically generated.

The `CoyoteAdapter.java` file has two of its methods account for over 90% of the total weighted churn and over 80% of the total churn. Only five methods out of the total of 27 have a weighted churn greater than 25. The `StandardContext.java` file has three of its methods account for over 50% of the total weighted churn and over 30% of the total churn. Only 14 out of the total of 515 methods have a churn greater than 25. Both files are large with over a thousand lines of code and have a long history of over 150 revisions, although the `StandardContext.java` is five times as large and has a history twice as long as `CoyoteAdapter.java`.

There is a clear pattern in the Open-Closed Breakers of these systems. All of them are large files with a long history. The distribution of weighted churn, as well as churn, is exponential both at the system level and at the module level: a small number of files (methods) account for most of the (weighted) churn in the system (files). The great majority of files and methods have an unsignificant amount of (weighted) churn. This hints at the fact that these large and intensely modified and extended entities lack abstraction. With a proper design, they would not grow as large and would be extendable through other means (e.g., polymorphism, or passing lambda functions as method parameters).

6.2.3 Single Responsibility Breakers

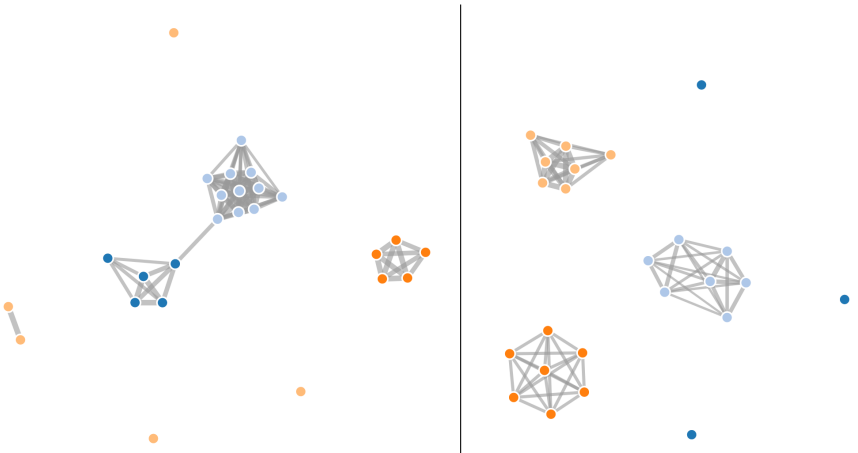
We identified instances of Single Responsibility Breakers in four systems: Guava, Hadoop, Jetty and SWT. We set the CHRONOS thresholds to `lower = 2` and `upper = 3`, and use the following values for the analysis filters:

- `--max-change-set=100`
- `--min-revisions=5`
- `--min-coupling=0.1`
- `--min-blob-density=2.5`
- `--max-anti-coupling=0.5`
- `--min-anti-blob-size=10`

Table 6.6 lists the source files with more than one identified responsibility from Guava, and Figure 6.7 shows the module graphs for the top two offenders.

Table 6.6: Guava Single Responsibility Breakers

Module	Responsibilities	Lines of code	Revisions
ClassPathTest.java	3	646	45
FluentIterable.java	3	851	71
SplitterTest.java	2	777	44
Futures.java	2	1581	199
MoreExecutors.java	2	1006	70
FuturesTest.java	2	3971	110



(a) FluentIterable.java (b) ClassPathTest.java

Figure 6.7: Module graphs of top Guava Single Responsibility Breakers

The `FluentIterable.java` file has three blobs. The orange blob consists of five overloads of the `concat` method. The blue blob consists of the `cycle`, `limit`, `skip` methods, and two overloads of the `filter` method. The light blue blob consists of the `contains`, `last`, `anyMatch`, `firstMatch`, `toArray`, `transform`, `first`, `isEmpty`, `get`, `allMatch`, and `size` methods. The `FluentIterable` class is a wrapper that adds operations on top of `Iterable`, and implements them using a delegate. Overloaded methods are semantically coupled, implementing the same operation with slightly different behavior (only the method signatures differ). Thus, the existence of a blob containing the `concat` overloads is expected.

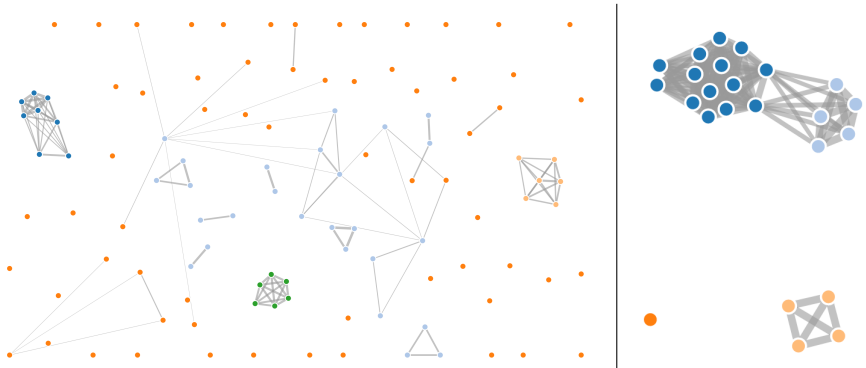
The `ClassPathTest.java` file also has three blobs. The light orange one consists of the `testClassPathEntries_*` series of methods. The orange one consists of the `testGetClassPathFromManifest_*` series of methods. The light blue one consists of the `testScan_*` and `testScanFromFile_*` series of methods, and the `resourceInfo`, `classInfo`, `testGetSimpleName`, `testGetPackageName`, methods. As we would expect, related test cases are strongly coupled with each other. Other test cases and utility methods that make use of a common subset of operations provided by the tested class are also coupled together, most likely because they are affected by changes in those operations.

The Single Responsibility Breakers are rather large files, all having between 500 and 4000 lines of code. They also have a substantial history, all having above 40 revisions.

Table 6.7 lists the source files with more than one identified responsibility from Hadoop, and Figure 6.8 shows the module graphs for the top two offenders.

Table 6.7: Hadoop Single Responsibility Breakers

Module	Responsibilities	Lines of code	Revisions
FSNamesystem.java	4	8031	594
TNAFSA.java	3	2098	16
FileSystemContractBaseTest.java	2	1017	21
TestFairScheduler.java	2	5417	193
TestLeafQueue.java	2	4131	92



(a) FSNamesystem.java (b) TNAFSA.java

Figure 6.8: Module graphs of top Hadoop Single Responsibility Breakers

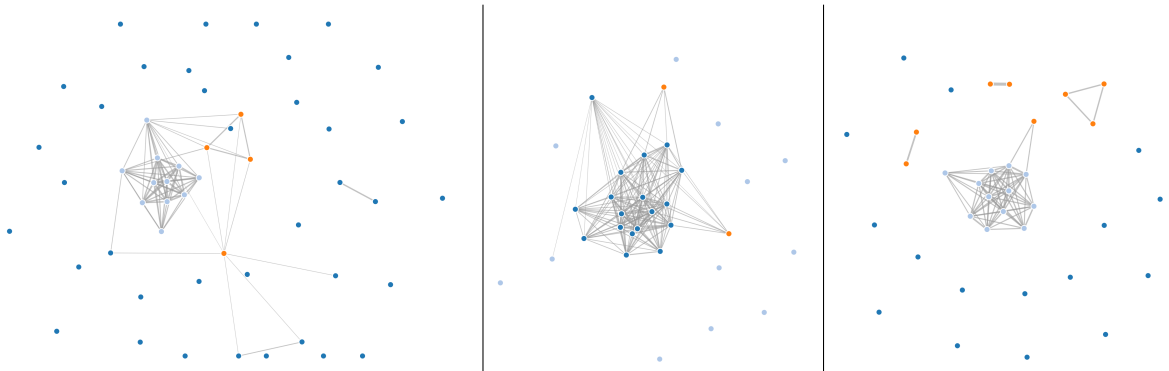
The `FSNamesystem.java` file has three blobs and an anti-blob. It is a very large file with over 8000 lines and an exceptionally long history of almost 600 revisions. The `TNAFSA.java`[4] file is also large, but contrasts the previous offender by having a very short history. Once again, related test cases are coupled together, which is expected. In this system, all Single Responsibility Breakers have above 1000 lines of code, but the history is varied: two have a short history, and three have a long history.

Table 6.8 lists the source files with more than one identified responsibility from Jetty, and Figure 6.9 shows the module graphs for the top three offenders.

Table 6.8: Jetty Single Responsibility Breakers

Module	Responsibilities	Lines of code	Revisions
TestABCase5.java	2	633	33
Request.java	2	2514	148
HttpParserTest.java	2	2251	89
Response.java	2	1490	134
ServletContextHandler.java	2	1591	63

[4] TestNativeAzureFileSystemAuthorization.java



(a) Request.java (b) Response.java (c) HttpParserTest.java

Figure 6.9: Module graphs of top Jetty Single Responsibility Breakers

In this case, the size of the Single Responsibility Breakers ranges from 500 to over 2500 lines of code, while the history ranges from an average number of 33 revisions to a large number of over 130 revisions. Except for `TestABCase5.java`, which has two blobs, all the other files have one blob and an anti-blob. Both regular files and test files have been identified with multiple responsibilities, as in the previous cases.

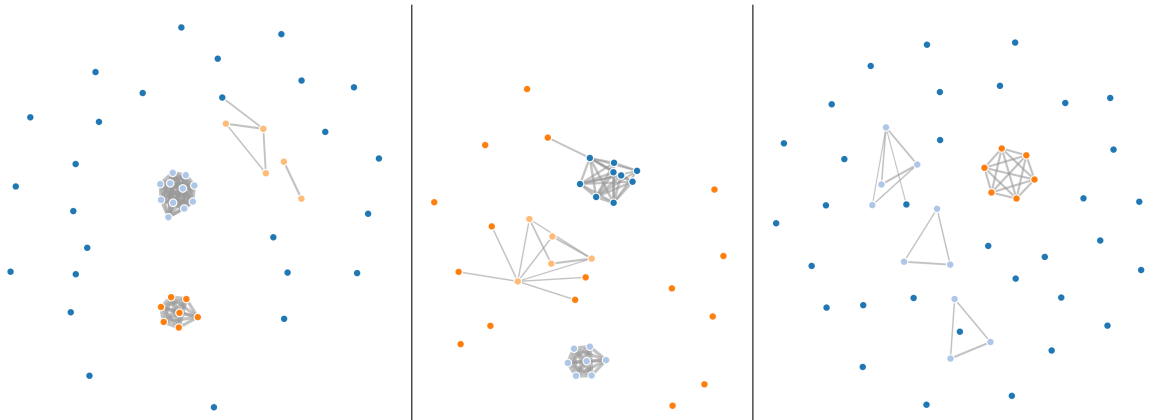
Table 6.9 lists the source files with more than one identified responsibility from SWT, and Figure 6.10 shows the module graphs for the top three offenders.

Table 6.9: SWT Single Responsibility Breakers

Module	Responsibilities	Lines of code	Revisions
win32//Accessible.java	3	5778	153
cocoa//Accessible.java	3	3604	73
gtk//GC.java	2	3881	253
gtk//Accessible.java	2	1101	59
test//Browser.java	2	2340	80
test//Display.java	2	1507	69
gtk//Control.java	2	6480	631
gtk//TextLayout.java	2	2357	141

The three `Accessible.java` files are very large with a long history. They implement a common accessibility API across different platforms. The `win32` and `cocoa` versions each have two blobs and an anti-blob. The `gtk//GC.java` file has one blob and an anti-blob. The blob consists of the `drawOval`, `fillArc`, `drawArc`, `drawRoundRectangle`, `fillOval`, `fillRoundRectangle` methods. These are fundamental drawing methods. In the case of SWT, unlike the previous systems, none of the Single Responsibility Breakers are test files. Moreover, there is little variance in the size or history of the offenders: all of them are very large, having at least 1000 lines of code, with a long history of over 55 revisions.

Having analyzed these systems, we can identify a few patterns. Single Responsibility Breakers tend to be rather large files, ranging from 500 lines of code to multiple thousands. Their history varies, from 15 revisions to hundreds. Except for SWT, all systems had instances among test files. We do not believe the test files to be an inherently bad design: it is expected that related test cases will change together, usually driven by a



(a) win32//Accessible.java (b) cocoa//Accessible.java (c) gtk//GC.java

Figure 6.10: Module graphs of top SWT Single Responsibility Breakers

change in the tested module. However, if this happens repeatedly, it might indicate that either the public interface of the tested module changes frequently, or that the internal implementation is tested instead of the module behavior, leading to changes in the tests everytime the module implementation changes.

There were rather few Single Responsibility Breakers compared to the other two analyses. We believe the reason is the formula used to compute temporal coupling (the Jaccard similarity coefficient), which leads to the detection of tightly and consistently coupled clusters of methods. Using a different formula that takes into account "occasional" coupling as well might lead to more diverse results.

7 Conclusions, Contributions and Future Work

7.1 Summary

Our goal was to detect design flaws related to the evolution of software systems by analyzing their history. In this sense, we developed the ChronoLens tool, which extracts a language-independent model that encompasses both the source structure and the changes performed on a system. Consequently, analyses can be built on top of it, focusing on the evolution itself. The proof-of-concept analyzers we developed showcase how ChronoLens can be used and offer a glimpse into the insights we can get by inspecting the evolution of systems.

We believe our application pushed further the state of the art in the field of software evolution analysis. The *Encapsulation Breakers* analysis overcomes the limitations of simple static analyzers such as Checkstyle [9], highlighting only exposed fields that were initially hidden. The *Open-Closed Breakers* analysis correlates a new metric with an actual design flaw, indicating a lack of abstraction. Thus, the cause is flagged, unlike Tornhill's *hotspots* [17], which indicate only a symptom. The *Single Responsibility Breakers* analysis combines the *co-change graph* visualization technique of Ball et al. [11] with the *Divergent Change* detection idea of Palomba et al. [18]. Therefore, we introduced a new approach that detects modules with multiple responsibilities, offering both a textual and visual representation of the identified subsets of methods making up the responsibilities.

We conducted a study on several large open-source systems, which revealed interesting insights on their design and evolution:

- *Encapsulation Breakers* — modules with many decapsulations usually have most of their fields decapsulated by relaxing the `private` visibility modifier; sometimes, this is to enable inheritance, confirming the idea that inheritance breaks encapsulation
- *Open-Closed Breakers* — most of the added and modified lines of code are accummulated by a few methods, making the distribution of (weighted) churn exponential both at system-level and at module-level; these churner methods likely lack abstraction and should be refactored to allow extension either through polymorphism, by passing lambda functions as parameters or some other mechanism that does not require modifying the method itself
- *Single Responsibility Breakers* — a larger group of overloaded methods usually form a responsibility cluster; related test methods are usually clustered together, as well; large sets of utility methods, weakly coupled with the rest of the module, are also frequent

7.2 Future Work

We conclude by proposing several research directions in which our work could be continued.

Implementing parsers for other programming languages besides Java, especially dynamic ones (e.g., Python), would allow us to compare the evolution of systems written in static and dynamic languages. This would also enable assessing the quality of software written in dynamic languages, a difficult task because of the weak guarantees static analysis can provide in this context.

The quality of the extracted history model can be improved by implementing refactoring detection in ChronoLens. We are interested in detecting *move refactorings* (renames are included in this category). Thus, the history of a source entity is not lost if it is merely renamed or moved to a different place. Other refactorings are less important, because they do not maintain the idendity of the affected source entities (e.g., an extracted method should not share the history of the method from which it was extracted).

The *Encapsulations Breakers* analysis could be refined to overcome its current limitations. Defining an algorithm that handles *encapsulations* (restricting the visibility of a field or accessor) would help flag only the fields that remained decapsulated compared to the original design. Detecting refactorings would also avoid false-positives reported when the type of a field, and implicitly the signature of its setter, change.

The *Open-Closed Breakers* analyzer aggregates several metrics besides the *weighted code churn*. Computing correlations between these metrics might produce interesting results, proving either the effectiveness and novelty or the equivalence of pairs of metrics. Moreover, inspecting anomalies in the context of correlated metrics could uncover interesting patterns (e.g., modules that change significantly a few times, or modules that change a little bit very often).

The results of the *Single Responsibility Breakers* analysis were rather scarce, and part of them were (predictably) test files. We believe the reason for this is the formula used to compute temporal coupling. In particular, a method that changed for its entire (shorter) lifetime together with another method with a long history can have an arbitrarily weak temporal coupling, depending on the difference in lifespan of the two methods. Experimenting with different formulas for temporal coupling (e.g., dividing by the minimum number of revisions of the two methods instead of the size of the union of the revision sets) could lead to more interesting results.

Towards the detection of new anti-patterns, the co-change graphs used in the *Single Responsibility Breakers* analysis could be extended for the *Feature Envy* and *Shotgun Surgery* code smells.

There is much work left to be done in the field of software evolution analysis. Although it could still be improved, ChronoLens is a solid tool upon which many analyses can be developed, focusing on the evolution of a system, on what and how changes.

References

[1] Meir M. Lehman. "Programs, life cycles, and laws of software evolution". In: *Proceedings of the IEEE* 68.9 (Sept. 1980), pages 1060–1076. ISSN: 0018-9219. DOI: 10.1109/PROC.1980.11805.

[2] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[3] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003. ISBN: 0-13-597444-5.

[4] Bertrand Meyer. *Object-Oriented Software Construction*. 1st edition. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988. ISBN: 0136290493.

[5] Barbara Liskov. "Keynote Address - Data Abstraction and Hierarchy". In: *SIGPLAN Not.* 23.5 (Jan. 1987), pages 17–34. ISSN: 0362-1340. DOI: 10.1145/62139.62141.

[6] Martin Fowler et al. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-48567-2.

[7] John C. Reynolds. "User-defined Types and Procedural Data Structures As Complementary Approaches to Data Abstraction". In: *Theoretical Aspects of Object-oriented Programming*. Edited by Carl A. Gunter and John C. Mitchell. Cambridge, MA, USA: MIT Press, 1994, pages 13–23. ISBN: 0-262-07155-X.

[8] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. "Automatic detection of bad smells in code: An experimental assessment". In: *Journal of Object Technology* 11.2 (Aug. 2012), 5:1–38. ISSN: 1660-1769. DOI: 10.5381/jot.2012.11.2.a5.

[9] *Checkstyle*. Version 8.10.1. May 28, 2018. URL: https://github.com/checkstyle/checkstyle.

[10] Michele Lanza. "The Evolution Matrix: Recovering Software Evolution Using Software Visualization Techniques". In: *Proceedings of the 4th International Workshop on Principles of Software Evolution*. IWPSE '01. New York, NY, USA: ACM, 2001, pages 37–42. ISBN: 1-58113-508-4. DOI: 10.1145/602461.602467.

[11] Thomas Ball et al. "If Your Version Control System Could Talk". In: *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*. Volume 11. 1997.

[12] Tudor Gîrba and Stéphane Ducasse. "Modeling History to Analyze Software Evolution". In: *Journal of Software Maintenance and Evolution: Research and Practice* 18.3 (May 2006), pages 207–236. ISSN: 1532-060X. DOI: 10.1002/smr.325.

[13] Sander Tichelaar, Stéphane Ducasse, and Serge Demeyer. "FAMIX and XMI". In: *Proceedings of the Seventh Working Conference on Reverse Engineering*. WCRE '00. Washington, DC, USA: IEEE Computer Society, 2000, pages 296–298. DOI: 10.1109/WCRE.2000.891485.

[14] Michele Tufano et al. "When and Why Your Code Starts to Smell Bad". In: *Proceedings of the 37th International Conference on Software Engineering*. Volume 1. Piscataway, NJ, USA: IEEE Press, May 2015, pages 403–414. DOI: 10.1109/ICSE.2015.59.

[15] Daniel Rațiu et al. "Using History Information to Improve Design Flaws Detection". In: *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering*. CSMR '04. Washington, DC, USA: IEEE Computer Society, 2004, pages 223–232. DOI: 10.1109/CSMR.2004.1281423.

[16] Marco D'Ambros, Michele Lanza, and Romain Robbes. "On the Relationship Between Change Coupling and Software Defects". In: *Proceedings of the 2009 16th Working Conference on Reverse Engineering*. WCRE '09. Washington, DC, USA: IEEE Computer Society, Oct. 2009, pages 135–144. DOI: 10.1109/WCRE.2009.19.

[17] Adam Tornhill. *Your Code as a Crime Scene*. Pragmatic programmers. Pragmatic Bookshelf, 2015. ISBN: 9781680500813.

[18] Fabio Palomba et al. "Mining Version Histories for Detecting Code Smells". In: *IEEE Transactions on Software Engineering* 41.5 (May 2015), pages 462–489. ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2372760.

[19] JetBrains s.r.o. *Kotlin Language Documentation, version 1.2*. 2017. URL: https://kotlinlang.org/docs/kotlin-docs.pdf (visited on 02/07/2018).

[20] Tim Lindholm et al. *The Java Virtual Machine Specification, Java SE 8 Edition*. 1st edition. Addison-Wesley Professional, 2014. ISBN: 978-0133905908.

[21] James R. Driscoll et al. "Making Data Structures Persistent". In: *J. Comput. Syst. Sci.* 38.1 (Feb. 1989), pages 86–124. ISSN: 0022-0000. DOI: 10.1016/0022-0000(89)90034-2.

[22] ECMA International. *The JSON Data Interchange Syntax*. Dec. 2017. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (visited on 06/02/2018).

[23] FasterXML. *Jackson*. Version 2.9.4. Feb. 7, 2018. URL: https://github.com/FasterXML/jackson.

[24] Robert A. Wagner and Michael J. Fischer. "The String-to-String Correction Problem". In: *J. ACM* 21.1 (Jan. 1974), pages 168–173. ISSN: 0004-5411. DOI: 10.1145/321796.321811.

[25] Hal Berghel and David Roach. "An Extension of Ukkonen's Enhanced Dynamic Programming ASM Algorithm". In: *ACM Trans. Inf. Syst.* 14.1 (Jan. 1996), pages 94–106. ISSN: 1046-8188. DOI: 10.1145/214174.214183.

[26] Google Trends. URL: https://www.google.com/trends (visited on 05/03/2017).

[27] *Git reference manual*. URL: https://kernel.org/pub/software/scm/git/docs/ (visited on 03/18/2018).

[28] TIOBE. URL: https://www.tiobe.com/tiobe-index/programming-languages-definition/ (visited on 06/02/2017).

[29] TIOBE. URL: https://www.tiobe.com/tiobe-index/ (visited on 06/06/2017).

[30] Eclipse Foundation. *Eclipse JDT Core*. Version 3.13.100. Feb. 7, 2018. URL: https://github.com/eclipse/eclipse.jdt.core.

[31] Grady Booch et al. *Object-Oriented Analysis and Design with Applications*. 3rd edition. Redwood City, CA, USA: Addison Wesley Professional, 2007. ISBN: 0-201-89551-X.

[32] Martin Fowler. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321127420.

[33] Nachiappan Nagappan and Thomas Ball. "Use of Relative Code Churn Measures to Predict System Defect Density". In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. New York, NY, USA: ACM, May 2005, pages 284–292. DOI: 10.1145/1062455.1062514.

[34] Paul Jaccard. "Étude comparative de la distribution florale dans une portion des Alpes et du Jura". In: *Bulletin de la Société Vaudoise des Sciences Naturelles* 37 (Jan. 1901), pages 547–579.

[35] Hiroki Yanagisawa and Satoshi Hara. "Discounted average degree density metric and new algorithms for the densest subgraph problem". In: *Networks* 71.1 (Aug. 2017), pages 3–15. DOI: 10.1002/net.21764.

[36] Oana Denisa Balalau et al. "Finding Subgraphs with Maximum Total Density and Limited Overlap". In: *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*. WSDM '15. New York, NY, USA: ACM, 2015, pages 379–388. ISBN: 978-1-4503-3317-7. DOI: 10.1145/2684822.2685298.

[37] Moses Charikar. "Greedy Approximation Algorithms for Finding Dense Components in a Graph". In: *Proceedings of the Third International Workshop on Approximation Algorithms for Combinatorial Optimization*. APPROX '00. Berlin, Heidelberg: Springer-Verlag, 2000, pages 84–95. ISBN: 3-540-67996-0.

[38] Radu Marinescu. "Quality Assessment in the Tower of Babel". Presentation given at ITCamp. Cluj-Napoca, Romania, May 2016. URL: https://vimeo.com/170923261 (visited on 06/02/2018).

[39] Ben Shneiderman. "Tree visualization with tree-maps: 2-d space-filling approach". In: *ACM Trans. Graph.* 11.1 (Jan. 1992), pages 92–99. ISSN: 0730-0301. DOI: 10.1145/102377.115768.

[40] Mike Bostock. *D3.js*. Version 4.13.0. Jan. 29, 2018. URL: https://github.com/d3/d3.

[41] Al Danial. *CLOC*. Version v1.72. Jan. 15, 2017. URL: https://github.com/AlDanial/cloc.