Faculty of Automation and Computers

Master Program: Software Engineering

DETECTING FEATURE ENVY USING SOFTWARE EVOLUTION

Dissertation Thesis

Andrei HEIDELBACHER

Supervisor:
Prof. dr. eng. Radu MARINESCU

Timișoara,
2021

Abstract

Software quality assurance is essential to ensure the flexibility and maintainability of software systems, which are necessary to allow them to evolve and adapt to business requirements. In order to tackle the complexity of large systems, many automated tools have been developed to aid the assessment of software quality. We leveraged prior work on analyzing the evolution of software systems and built a tool that detects instances of the Feature Envy anti-pattern using temporal coupling, a metric that indicates how often software entities (e.g., methods) change together, and co-change graphs, a visualization technique for the identified anti-patterns. We then analyzed several open-source systems and showcased the general patterns and interesting findings.

Contents

List of Figures

List of Tables

Listings

1 Introduction. Problem Statement

1.1 Motivation

Software systems must evolve and adapt to client requirements in order to remain relevant and satisfy business needs. In order to allow this, the systems must be designed with flexibility and maintainability in mind. Software quality assurance aims to assess and improve such characteristics of software. However, most systems grow and become difficult to be inspected manually. As a result, automated tools have been developed to aid software quality assessment.

Many such tools inspect the syntactical structure of the code to detect common anti-patterns (e.g., code duplication). Although such tools provide important information on the code quality, they are restricted to a single snapshot of the analyzed systems.

Modern software is developed with the aid of version control systems, which enable multiple developers to work concurrently, allow tracking changes, managing versions, etc. This historical information can be used to develop tools that assess the quality of a system based on how it changes and evolves over time, having the potential to offer deeper insights than tools that inspect only a single snapshot of a system.

Our goal is to utilize the history and evolution of software systems to detect design flaws and anti-patterns.

1.2 Contribution

We build on our prior work, ChronoLens [1], a framework that allows developing automated software evolution analyzers, and implement a *Feature Envy* [2] anti-pattern detector based on *temporal coupling* [3] and *co-change graphs* [4].

Specifically, we compute the temporal coupling (a metric that quantifies how often two software entities change together) between all pairs of methods in a system, and build cross-file co-change graphs (in which nodes represent methods, and weighted edges represent the temporal coupling between the endpoints). If the sum of temporal couplings of a method with all other methods in a different file is greater than the sum of temporal couplings with methods from its own file, then that method is reported as an instance of Feature Envy.

A sparse temporal coupling matrix with methods as indices is already provided by ChronoLens as part of the prior *Divergent Change* analyzer. However, cross-file co-change graphs require additional work, because we must figure out what files should be involved. We check every method in the system individually, and compute the sum of couplings to every other file. In a cross-file co-change graph (which will be a sub-graph of

the co-change graph of the entire system), we include the files that have a tighter coupling with a method than the method's own file. We compute these cross-file co-change graphs for every Feature Envy instance and append them to the analysis report.

Finally, we analyze several large open-source systems, and note patterns observed in the analysis reports.

1.3 Organization

Our contributions build up on our prior work, the ChronoLens software analysis tool [1]. The same fundamental concepts, software quality and evolution principles, and anti-patterns are required to understand how we extended the tool to develop our analysis and the insights we discuss as part of the experimental results. Additionally, we are exploring the same research space of assessing software quality by inspecting how a system changes over time, and thus, the same state of the art retains its relevance. Therefore, we included Chapter 2 and Chapter 3 from our prior work, *Detecting Anti-Patterns by Analyzing the Evolution of Software Systems* [1]. However, we expanded Chapter 3 with Section 3.5, which summarizes Chapter 4 and Section 5.3 of our prior work [1] and presents the recent changes to the source code and history model used by the tool. We also adjusted some paragraphs in the other sections to better reflect the connection to our current application.

In Chapter 4, we describe the core of our current application, the *Feature Envy* anti-pattern detector: metrics used to detect the design flaw, cross-file coupling graphs, detection algorithm, etc.

Chapter 5 presents the command-line interface of our detector and the techniques we used to visualize and inspect the results of our analyses. In Section 5.1, we include Section 6.1 from our prior work [1], and adjusted it for the *report format* and *force layouts* produced by our current analyzer. We then present experimental results and observed patterns following the analysis of several large open-source software systems.

We summarize our contributions in Chapter 6, reiterate directions for future research that we've mentioned in our previous work [1], as well as present additional ideas for future 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 programming, 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" [5]. 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" [6]. 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* [7], 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 [8]).

Liskov Substitution Principle
"Subtypes must be substitutable for their base types" (originally stated by Liskov [9]).

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" [5]. 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* [2], 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" [6]. 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.

Abstract Factory is a creational pattern used to "provide an interface for creating families of related or dependent objects without specifying their concrete classes" [6]. 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.

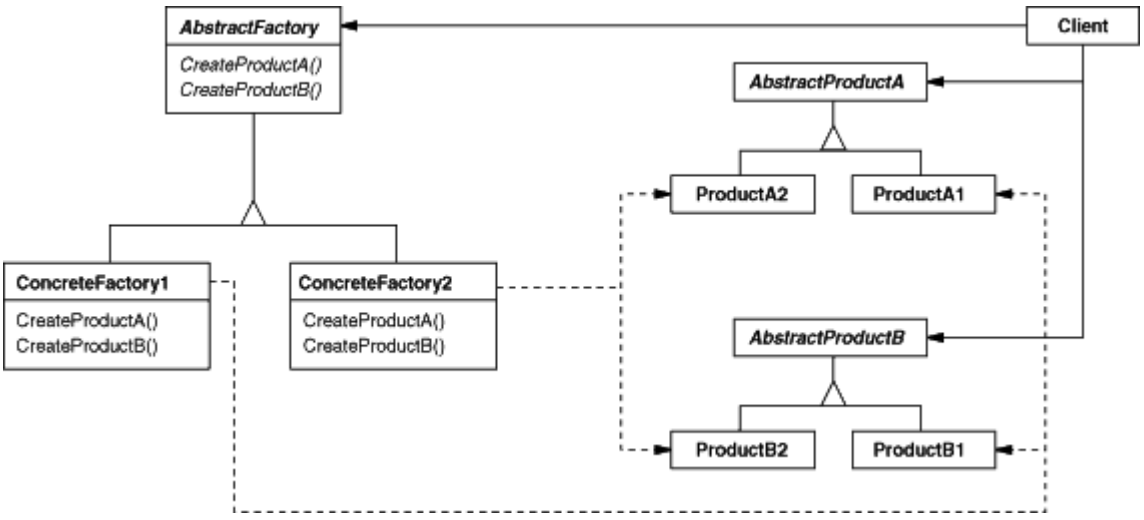

Figure 2.1: UML class diagram of the Abstract Factory pattern structure [6]

Singleton is another creational pattern used to "ensure a class only has one instance, and provide a global point of access to it" [6]. 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.

Proxy is a structural pattern used to "provide a surrogate or placeholder for another object to control access to it" [6]. 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

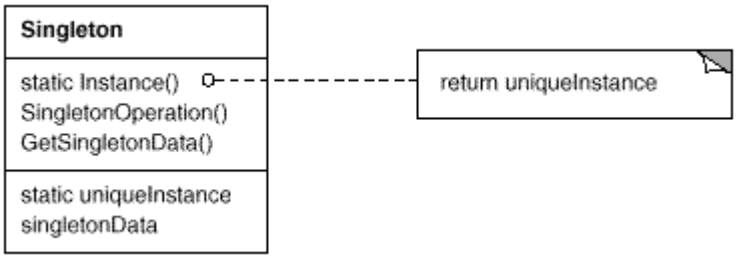

Figure 2.2: UML class diagram of the Singleton pattern structure [6]

operations to a remote object.

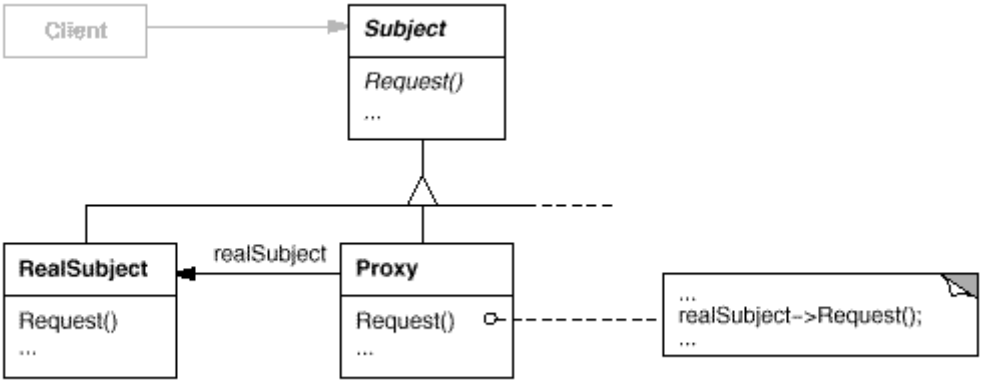

Figure 2.3: UML class diagram of the Proxy pattern structure [6]

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" [6]. 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.

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.

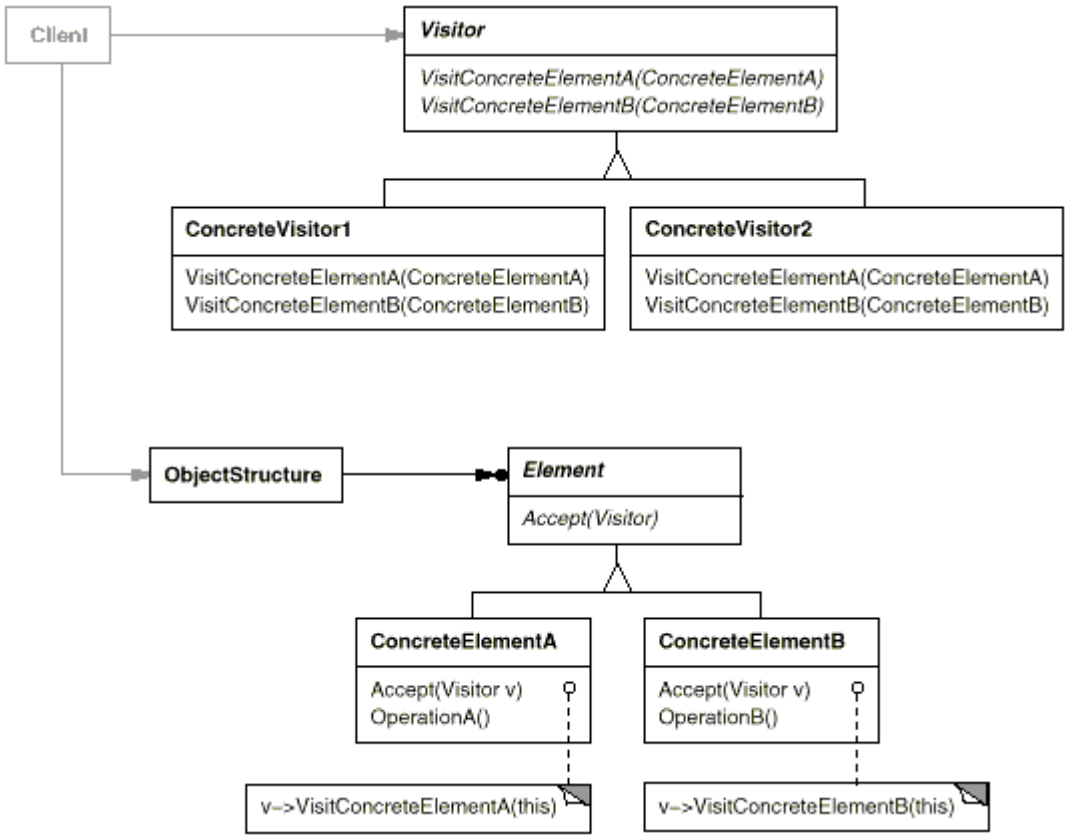

Figure 2.4: UML class diagram of the Visitor pattern structure [6]

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 [10] 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 [11] published a comparative study on the existing tools that detect bad smells.

Some of these tools, such as CHECKSTYLE [12], 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. The CHRONOLENS tool attempts to overcome these limitations through its *Encapsulation Breakers* analysis [1] by reporting fields that were initially hidden and exposed publicly at a later point in time.

Lanza and Marinescu [13] devised many metrics used to detect various anti-patterns. For example, their strategy for detecting instances of **Feature Envy** uses the **Access to Foreign Data (ATFD)**, **Locality of Attribute Accesses (LAA)**, and **Foreign Data Providers (FDP)** metrics, and is summarized in Figure 3.1.

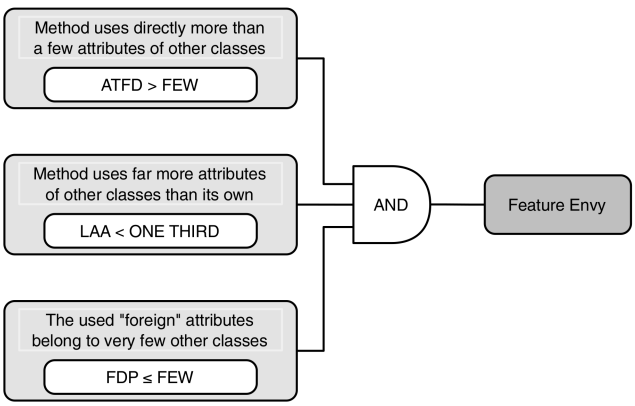

Figure 3.1: Detection strategy for Feature Envy [13]

All of these metrics are computed based on the syntactical structure of the code and describe the state of a software entity at a single point in time. In contrast, our current analysis detects instances of Feature Envy based on how often software entities change together, thus relying on a *semantic* coupling rather than a *syntactic* or *structural* coupling.

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* [5] 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** [14] 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 3.2 shows a schematic evolution matrix.

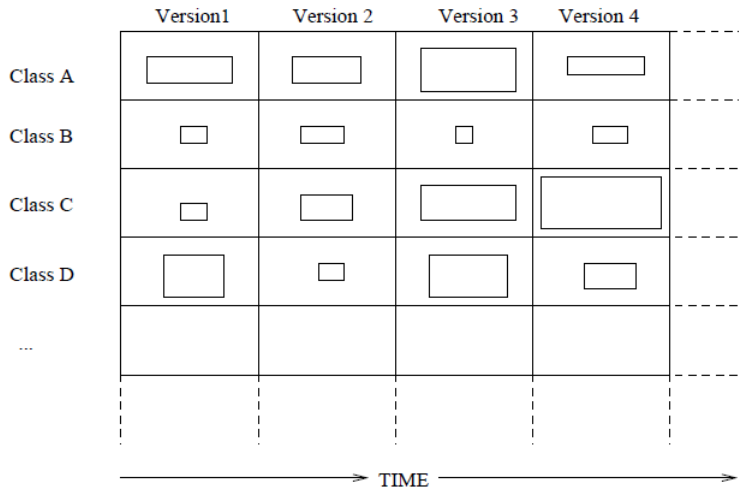

Figure 3.2: Schematic description of the evolution matrix [14]

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 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 by visualizing the evolution of a simple structural metric. Ball et al. [4] 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 3.3 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.

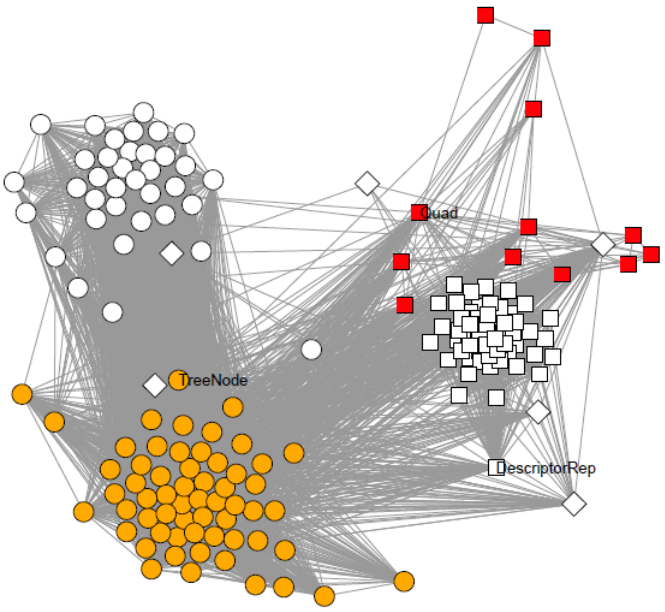

Figure 3.3: Example of a co-change graph [4]

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.

Hismo is a meta-model devised by Gîrba et al. [15] 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* [16] model to represent syntactical structure and a sequence of versions to represent the history of each entity, ChronoLens [1] defines a *history model* that makes use of a simpler, language-independent representation of source code, and integrates *changes* into the model. Thus, instead of maintaining multiple snapshots of the same entity, it maintains 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. [17] 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. [18] 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 [3] defined the concept of **change coupling** of a class with the entire system, unlike Ball et al. [4], 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 *Feature Envy* (and previously, *Single Responsibility Breakers*) analysis, but at a finer-grained level (between two methods) and with a different goal: finding methods related to the responsibilities of another class, not predicting software defects.

In his book, *Your Code as a Crime Scene* [19], Tornhill proposed many analyses that inspect the evolution of a system using version control systems. A 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 [3] as a preliminary measure to determine files that need a closer inspection. Then, he computes the *degree of coupling* as the percentage 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 modules in an attempt to identify methods related to responsibilities of a class different from their own.

Another concept defined by Tornhill is that of a **hotspot**, 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.

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

Palomba et al. [20] 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 **Feature Envy**, their approach will identify a method as being affected by this smell if it is "involved in commits with methods of another class of the system β% more than in commits with methods of their class" [20]. Although our analysis is similar, we identify instances of this smell through cumulative temporal coupling, and also provide a visualization in the form of co-change graphs for the affected source files.

3.5 ChronoLens

3.5.1 History Model

The ChronoLens tool analyzes the version control history of a software system and builds a language-agnostic **history model**, which can then be used to detect various anti-patterns. While this model is largely the same as in our original prior work [1], we enhanced it with a few minor changes to simplify some operations. We will present a summary of the current state of the model.

Figure 3.4 shows how source nodes (e.g., classes or methods) are modeled, and Figure 3.5 shows how changes are modeled in the evolution of a system.

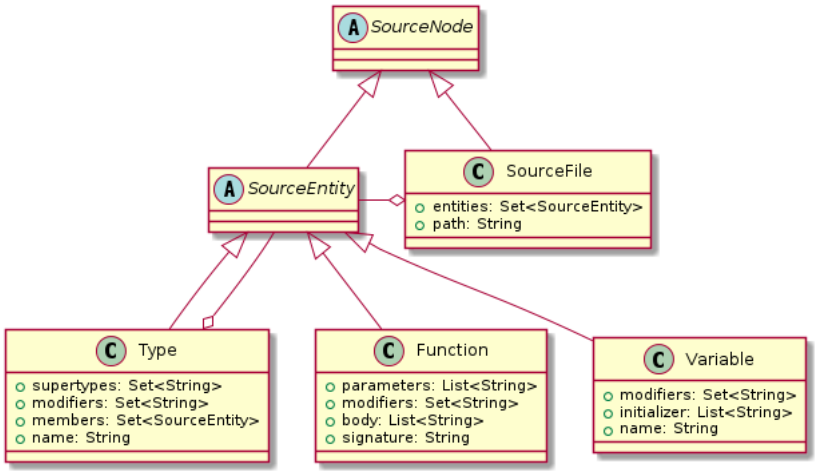

Figure 3.4: UML class diagram of the source model

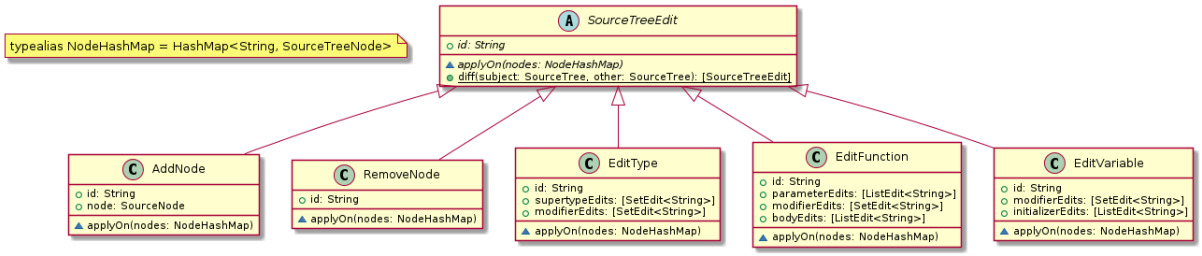

Figure 3.5: UML class diagram of the delta model

A **transaction** contains the set of changes applied in a single revision captured by the version control system, as presented in Figure 3.6. A **source tree**, described in Figure 3.7, contains the set of *source files* in a software system, decorates all *source nodes* with a globally unique identifier, and provides utilities to update the *source model* by efficiently applying a *transaction*.

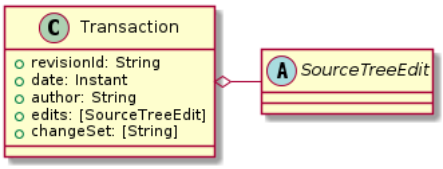

Figure 3.6: UML class diagram of `Transaction`

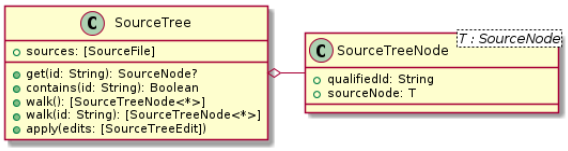

Figure 3.7: UML class diagram of `SourceTree`

3.5.2 Temporal Coupling. Co-Change Graphs

One of the analyses implemented as part of ChronoLens in our prior work [1] is the detection of *Single Responsibility Breakers*. Namely, a source file breaks the *Single Responsibility Principle* [7] if it changes for multiple reasons (equivalent to the *Divergent Change* code smell described by Fowler et al. [2]).

The detection strategy builds a **co-change graph**, in which nodes represent methods within a source file, and weighted edges represent the **temporal coupling** between two methods. Figure 3.8 presents an example graph. D'Ambros, Lanza and Robbes defined *change coupling* (also known as *temporal coupling*, as noted by Tornhill in his book [19]) 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" [3]. We define the temporal coupling C between two methods M_1 and M_2, where $R(M)$ is the set of revisions in which method M changed, as the *Jaccard similarity coefficient* [1, 21]:

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

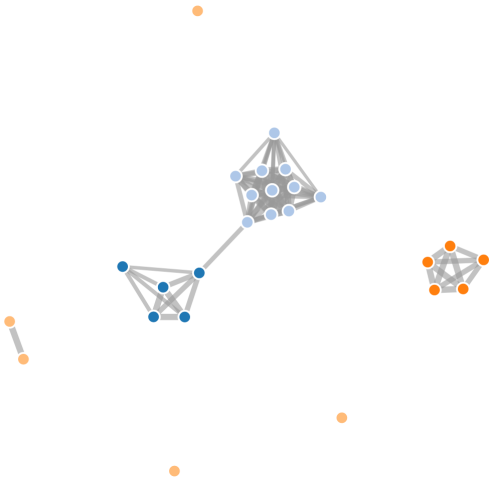

Figure 3.8: Co-change graph of `FluentIterable` from Guava [1]

Thus, the temporal coupling is a fractional number between 0 and 1 that indicates how often two methods changed together. Then, subsets of methods that are tightly coupled to each other (i.e., "a reason for change" [7]) are identified by using a simple

clustering algorithm. A source file is considered to be a *Single Responsibility Breaker* if it contains multiple clusters (i.e., fulfills multiple responsibilities).

In order to identify methods that are tightly coupled to files other than their own, we extend the *co-change graphs* to span methods from multiple files, or *cross-file co-change graphs*, which represent the key in detecting instances of Feature Envy in our current analysis.

3.6 Summary

Software evolution can offer deeper insights on design quality. We developed a *Feature Envy* detector on top of ChronoLens [1], a history analysis tool that extracts the history of a system, storing the changes from one version to another of a software entity.

We especially leverage concepts from the *Single Responsibility Breakers* analysis, which combines the co-change graph visualization developed by Ball et al. [4] and the Divergent Change detection strategy of Palomba et al. [20] (however, the analysis makes use of a different formula for temporal coupling, and uses a different clustering algorithm).

Our current analysis differs from that of Lanza and Marinescu [13] in the analyzed subject: their analysis inspects the syntactical structure of a single snapshot of the system to detect methods that access a disproportionately large number of fields and methods from another class with metrics such as *ATFD*, *LAA* or *FDP*, whereas ours inspects the history of a system, detecting methods that consistently change together with methods from another class through the sum of *temporal couplings*, thus establishing a semantic coupling.

Compared to Palomba et al. [20], who detect instances of Feature Envy simply by inspecting how often a *method* changes together with another *class*, we employ a stricter strategy: a *method* must consistently change together with *methods of another class*. In other words, instead of measuring the temporal coupling with another class, we measure the sum of temporal couplings with individual methods from another class. This gives us greater control over the detection algorithm, e.g., we can ignore pairs of methods that changed together fewer than T times (where T is a configurable threshold), allowing us to filter out "coincidences" (e.g., a *hotspot method* that changes very frequently throughout the lifetime of a system will inevitably change together with another class, but if it changes only once or twice together with individual methods of said class, it is less likely to exist a real coupling between them).

Additionally, we provide a co-change graph visualization to aid understanding the degree of coupling between the affected methods and classes and to direct potential refactoring efforts.

The goal of our analysis is to indicate the causes that lead to a flawed design (i.e., a method tightly coupled to another class' responsibility), not only symptoms of bad design (e.g., low cohesion), as well as to reinforce that the ChronoLens tool can be used to develop new analyses, enabling further research in the field of software evolution analysis.

4 Detecting Feature Envy

The main goal of our application is to detect *Feature Envy* instances by inspecting the evolution of software systems. To this end, we developed a specialized analyzer on top of the ChronoLens tool [1]. We leveraged the shared infrastructure provided by the tool to extract and query the history of a software system, as well as the tool's command-line interface. We developed our *Feature Envy* analyzer in the *Kotlin programming language* [22], and it runs on the *Java Virtual Machine* [23], just like the ChronoLens tool.

4.1 Detection Strategy

Fowler defined an instance of **Feature Envy** as "a method that seems more interested in a class other than the one it actually is in" [2]. Although he notes that the "most common focus of envy is the data" [2] (e.g., "a method that invokes half-a-dozen getting methods on another object to calculate some value" [2]), suggesting a structural or syntactic coupling with the envied class, he acknowledges that there are exceptions, which are not anti-patterns, but rather patterns to alleviate other anti-patterns (e.g., the Strategy and Visitor design patterns [6], which combat the Divergent Change anti-pattern [2]).

According to Fowler, the "fundamental rule of thumb is to put things together that change together. Data and behavior that references that data usually change together" [2], noting that we should "move the behavior to keep changes in one place" [2]. Thus, even though he defines *structural symptoms* of Feature Envy (a method that accesses many fields or methods of another class), the *cause* of Feature Envy is that a method is used to fulfill a responsibility different from the one it belongs to, i.e., a *semantic symptom* manifested by a method that changes together with a different class.

In this regard, Palomba et al. conjectured that "a method affected by feature envy changes more often with the envied class than with the class it is actually in" [20]. However, a *hotspot method* (i.e., "complex parts of the codebase that have changed quickly" [19]) that changed very frequently could have changed together with another class oftenly without necessarily participating in that class' responsibilities. Therefore, we believe that we can further refine the definition of a Feature Envy instance.

The *Single Responsibility Principle* states that "a class should have only one reason to change" [7]. Consequently, in our prior work we defined a **responsibility** as "a cluster of methods that have a strong temporal coupling with each other" [1]. Note that, as mentioned in our prior work, we extended the coupling between method and *class* to method and *file*. The rationale for this generalization is that multple classes can exist in a file (e.g., inner classes in object-oriented languages, such as Java or C#, or standalone classes or functions in languages that combine concepts from multiple programming paradigms,

such as Javascript or Python), but all software entities in a source file contribute towards the same purpose or responsibility.

Although in that context we analyzed responsibilities within a single file, it is possible for a responsibility to span methods across multiple files. If the majority of the methods that comprise a responsibility are located together in the same class, and only a few other methods involved in that responsibility are scattered in other files, then it follows that those other methods belong to the class that contains the majority of them.

Therefore, for the purpose of our current analysis, we define an instance of **Feature Envy** as a method that has stronger temporal coupling with the methods of another file than with the methods of its own file. This gives us greater control by allowing us to filter out temporal couplings below a configurable threshold T (i.e., methods that changed together only once or twice in the lifetime of a system are unlikely to be connected to the same responsibility), and thus, we should be able to filter out the aforementioned hotspot methods.

In the following sections, we present detailed and formal descrptions of the used metrics, as well as algorithms to compute the metrics, graphs, and to implement the detection strategy.

4.2 Temporal Coupling

The key **metric** used in our analysis is **temporal coupling** (coined by Tornhill [19]), originally defined as *change coupling* 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" [3].

In our prior work [1], we defined the temporal coupling C between two methods M_1 and M_2, where $R(M)$ is the set of revisions in which method M changed, as the *Jaccard similarity coefficient* [21]:

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

Therefore, the temporal coupling C will always be a number between 0 (indicating that methods M_1 and M_2 *never* changed together) and 1 (indicating that methods M_1 and M_2 *always* changed together).

In order to detect Feature Envy instances, we inspect each individual method and compute the temporal coupling to every source file as the sum of temporal couplings with all methods in the corresponding source file. Let M be the inspected method, and F be a source file. The temporal coupling of method M to file F is defned as:

$$C(M, F) = \sum_{M_i \in F, M_i \neq M} C(M, M_i)$$

We then define an **envy ratio** T between 0 and 1. Let F_M be the source file that contains method M. We say that method M *envies* file F if $C(M, F) > T * C(M, F_M)$. In other words, we define a Feature Envy instance as a method that changes more oftenly with more methods from another source file than its own source file. The rationale behind

the envy ratio T is that even if a method doesn't change more frequently with another file than its own, if it does so on the same order of magnitude, then it is still an anti-pattern worth looking into.

Note that it is possible for a method M to envy multiple files F_i. Although a Feature Envy instance is typically defined as a method that envies a *single* other file, we believe that a method that envies *multiple* files could be an interesting anti-pattern to investigate. Therefore, we introduce a **maximum envy count** to limit the number of reported envied files (in descending order by $C(M, F_i)$). This will allow us the flexibility to investigate such anti-patterns, while retaining relevance and reducing redundancy for our findings.

4.3 Co-Change Graphs

Inspired by Ball et al. [4], in our prior work, we defined a **module co-change graph** 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" [1]. In order to improve the quality of this decomposition, we used two heuristic filters, `--min-revisions` and `--min-coupling`, which filter out methods with a very short history and edges that could be a coincidence from the graph.

In the context of the Single Responsibility Breakers analysis, a *module* denoted a *source file*, and the co-change graph was used to aid in decomposing a module into *responsibilities* (specifically, by clustering methods in *blobs*, *transition methods*, and an *anti-blob*). For the purpose of our current *Feature Envy* analysis, however, we are interested in more than just one source file, and do not care about highlighting method clusters, but rather methods with tighter coupling to another file.

Therefore, we extend the prior *module co-change graph* into a **cross-file co-change graph**, which uses the same definitions for nodes, edges, and weights, and comprises methods from both the envying file and the envied files. This structure will allow visualizing the coupling between methods from different files, understanding which methods are involved in a specific Feature Envy instance and to what degree, and will help guide refactoring efforts.

4.4 Detection Algorithm

The detection algorithm works in three steps: compute the sparse temporal coupling matrix (which has methods as indices), identify the Feature Envy instances, and build the cross-file co-change graphs.

The **sparse temporal coupling matrix** is the core that powers our analysis. It uses globally unique identifiers as row and column keys, with the values representing the temporal coupling between the methods denoted by those identifiers (and is, therefore, a symmetric matrix). In the following listing, we present auxiliary aliases and utilities to simplify working with sparse matrices:

Listing 4.1: Sparse matrix definition

```
typealias SparseMatrix<K, V> = Map<K, Map<K, V>>
typealias SparseHashMatrix<K, V> = HashMap<K, HashMap<K, V>>

fun <K, V : Any> emptySparseHashMatrix(): SparseHashMatrix<K, V> =
    hashMapfOf()

operator fun <K, V : Any> SparseMatrix<K, V>.get(x: K, y: K): V? =
    this[x].orEmpty()[y]

operator fun <K, V : Any> SparseHashMatrix<K, V>.set(x: K, y: K, value: V) {
    if (x !in this) {
      this[x] = hashMapOf()
    }
    this.getValue(x)[y] = value
}
```

As in our prior work, "the temporal coupling C between two methods M_1 and M_2 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 definition of C can be computed as follows" [1]:

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

In order to identify which methods were changed in each revision, we traverse all the source subtrees rooted in edited source nodes for the analyzed transaction using the **Visitor pattern** [6]:

Listing 4.2: Computing changed methods in a revision

```
private val sourceTree = SourceTree.empty()

private fun visit(transaction: Transaction): Set<String> {
    val editedIds = hashSetOf<String>()
    for (edit in transaction.edits) {
        when (edit) {
          is AddNode -> editedIds += visit(edit) // smart-cast to AddNode
          is RemoveNode -> editedIds += visit(edit) // smart-cast to RemoveNode
          is EditFunction -> editedIds += edit.id
        }
        sourceTree.apply(edit)
    }
    return editedIds
}

// 'visit' overloads for AddNode/RemoveNode types have been omitted for brevity.
```

Note that the `visit(edit: RemoveNode)` overload is a stateful operation, as it removes previously filled entries in the `changes` map and `jointChanges` sparse matrix, which track how many revisions changed a particular method, and how many revisions changed a pair of methods together. Further, the analyzer will inspect every transaction from the history of the system, updating the `changes` and `jointChanges` as follows [1]:

Listing 4.3: Updating the revision count for methods and pairs of methods

```
private val changes = hashMapOf<String, Int>()
private val jointChanges = emptySparseHashMatrix<String, Int>()

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
            jointChanges[id1, id2] = (jointChanges[id1, id2] ?: 0) + 1
        }
    }
}
```

Lastly, the temporal coupling between two methods identified by `id1` and `id2` can be computed as follows:

Listing 4.4: Computing temporal coupling from changes and joint changes

```
private fun computeTemporalCoupling(id1: String, id2: String): Double {
    val countId1 = changes[id1] ?: return 0.0
    val countId2 = changes[id2] ?: return 0.0
    val countId1AndId2 = jointChanges[id1, id2] ?: return 0.0
    val countId1OrId2 = countId1 + countId2 - countId1AndId2
    return 1.0 * countId1AndId2 / countId1OrId2
}
```

As in our prior work, "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" [1], which limits not only the computational effort, but also filters out irrelevant revisions (e.g., formatting changes affecting a very large number of files, or comitting a large number of auto-generated files).

We make use of the `changes`, `jointChanges` and `temporalCoupling` data structures repeatedly across our analyzers, and thus we encapsulated them in a `TemporalContext`, which provides utility access methods:

Listing 4.5: The `TemporalContext` data holder

```
class TemporalContext(
    private val changes: Map<String, Int>,
    private val jointChanges: SparseMatrix<String, Int>,
    private val temporalCoupling: SparseMatrix<String, Double>,
) {
  val ids: Set<String> get() = changes.keys

  val cells: Collection<Pair<String, String>> by lazy {
      val c = arrayListOf<Pair<String, String>>()
      for ((x, row) in jointChanges) {
          for (y in row.keys) {
              cells += x to y
          }
```

```
14          }
15          c
16      }
17
18      fun revisions(id: String): Int = changes[id] ?: 0
19      fun revisionsOrNull(id: String): Int? = changes[id]
20      fun revisions(id1: String, id2: String): Int = jointChanges[id1, id2] ?: 0
21      fun revisionsOrNull(id1: String, id2: String): Int = jointChanges[id1, id2]
22
23      fun coupling(id1: String, id2: String): Double =
24          temporalCoupling[id1, id2] ?: 0.0
25
26      fun couplingOrNull(id1: String, id2: String): Double? =
27          temporalCoupling[id1, id2]
28  }
```

The next step in our algorithm is to find the Feature Envy instances, which will allow our tool to produce a report with the findings. For this purpose, we first compute the `functionToFileCoupling` sparse matrix, which is asymmetrical and has globally unique method and source file identifiers as indices:

Listing 4.6: Computing temporal coupling between functions and source files

```
1  private fun TemporalContext.computeFunctionToFileCouplings():
2      SparseMatrix<String, Double> {
3          val functionToFileCoupling = emptySparseHashMatrix<String, Double>()
4          for ((id1, id2, coupling) in cells) {
5              val function = id1
6              val file = id2.sourcePath
7              val coupling = coupling(id1, d2)
8              functionToFileCoupling[function, file] =
9                  (functionToFileCoupling[function, file] ?: 0.0) + coupling
10         }
11         return functionToFileCoupling
12     }
```

The Feature Envy instances will be processed and used repeatedly, and thus we encapsulate them in a data class:

Listing 4.7: Feature Envy instance data class

```
1  data class FeatureEnvy(
2      val function: String,
3      val coupling: Double,
4      val enviedFile: String,
5      val enviedCoupling: Double,
6  ) {
7      val file: String get() = function.sourcePath
8  }
```

We then inspect every method, sort the other source files in descending order by the coupling with the respective method, and detect a Feature Envy instance for every file that meets the coupling threshold (based on the **envy ratio**). Note that, according to the **maximum envy count**, only the tightest coupled `maxEnviedFiles` files will be reported:

Listing 4.8: Computing the tightest coupled source files for functions

```
private fun findFeatureEnvyInstances(
    functionToFileCoupling: SparseMatrix<String, Double>,
): List<FeatureEnvy> {
    val featureEnvyInstances = arrayListOf<FeatureEnvy>()
    for ((function, fileCouplings) in functionToFileCoupling) {
      fun couplingWithFile(f: String): Double = fileCouplings[f] ?: 0.0

        val file = function.sourcePath
        val selfCoupling = couplingWithFile(file)
        val couplingThreshold = selfCoupling * minEnvyRatio
        val enviedFiles =
            (fileCouplings - file).keys
                .sortedByDescending(::couplingWithFile)
                .takeWhile { f -> couplingWithFile(f) > couplingThreshold }
        featureEnvyInstances +=
            enviedFiles.take(maxEnviedFiles).map { f ->
                FeatureEnvy(function, selfCoupling, f, couplingWithFile(f))
            }
    }
    return featureEnvyInstances.sortedByDescending(FeatureEnvy::enviedCoupling)
}
```

Lastly, we compute the **co-change graphs** for the infringing files, and extend them to span the envied files as well. For this purpose, we reuse the graph data structures and utilities from our prior work [1], which compute a co-change graph given a `TemporalContext` and a subset of node ids (keys in the sparse coupling matrix):

Listing 4.9: Computing co-change graphs from `TemporalContext`s and node subsets

```
data class Graph(val label: String, val nodes: Set<node>, val edges: List<Edge>)
data class Node(val label: String, val revisions: Int)
data class Edge(
    val source: String,
    val target: String,
    val revisions: Int,
    val coupling: Double,
)

fun TemporalContext.buildGraphFrom(label: String, nodeIds: Set<String>): Graph {
    val nodes = hashSetOf<Node>()
    for (id in nodeIds) {
        val revisions = reivisionsOrNull(id) ?: continue
        nodes += Node(id, revisions)
    }
    val edges = arrayListOf<Edge>()
    for (id1 in nodeIds) {
        for (id2 in nodeIds) {
            if (id1 >= id2) continue
            val revisions = revisionsOrNull(id1, id2) ?: continue
            val coupling = couplingOrNull(id1, id2) ?: continue
            edges += Edge(id1, id2, revisions, coupling)
        }
    }
}
```

Note that the utility can be used to compute a graph for methods belonging to the same file (as was the case for the prior *Divergent Change* analysis [1], where the goal was finding clusters of methods within the same file to highlight multiple repsonsibilities), or for methods belonging to multiple files. In order to compute the appropriate graphs for our *Feature Envy* analysis, we identify all source files that contain Feature Envy instances, all the other files envied by each such infringing file, and build the resulting **cross-file co-change graphs**:

Listing 4.10: Computing cross-file co-change graphs for Feature Envy instances

```
private fun TemporalContext.buildGraphs(
    featureEnvyInstancesByFile: Map<String, List<FeatureEnvy>>,
): List<Graph> {
    val idsByFile = ids.groupBy(String::sourcePath)
    return featureEnvyInstancesByFile.map { (path, instances) ->
        val enviedFiles = instances.map(FeatureEnvy::enviedFile).toSet()
        val nodeIds = (enviedFiles + path).map(idsByFile::get).flatten().toSet()
        buildGraphFrom(path, nodeIds)
    }
}
```

In order to facilitate visualization, we assign a different color to methods belonging to each file, and also a unique color to every Feature Envy instance (note that we encode colors as unique indices — mapping indices to actual colors is up to the rendering component):

Listing 4.11: Assigning colors to files and Feature Envy instances

```
data class ColoredGraph(graph: Graph, colors: Map<String, Int>)

private fun Graph.colorNodes(
    featureEnvyInstancesByFile: Map<String, List<FeatureEnvy>>,
): ColoredGraph {
    val instances =
        featureEnvyInstancesByFile.getValue(label)
            .map(FeatureEnvy::function)
            .toSet()
    val fileGroups = nodes.map(Node::label).groupBy(String::sourcePath).values
    val groups = fileGroups + instances.map(::listOf)
    return colorNodes(groups) // assigns a unique color to every group of nodes
}

// Overload has been omitted for brevity.
fun Graph.colorNodes(groups: Collection<Iterable<String>>): ColoredGraph
```

Finally, we have all the data inferred from the history analysis available in the `featureEnvyInstances` and `coloredGraphs`, which will allow us to produce reports and visualize the anti-patterns in any necessary format.

5 Usage. Experimental Results

5.1 Usage

5.1.1 Command-Line Interface

Our application has been developed as an analyzer on top of the ChronoLens tool [1], which exposes a simple **command-line interface** and interacts with the repository detected in the current working directory. The tool and analyzer are packaged and released as a Java application distribution.[1]

The ChronoLens tool connects to the repository through a version control system proxy and provides several commands, of which the most relevant for our application are:

- `help` — prints usage information
- `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

Our specialized analyzer loads the persisted history model from the current working directory and prints to the standard output a report containing the results. The various filters used by the analysis (e.g., the minimum temporal coupling between two methods) can be specified as options (e.g., `--min-coupling=0.25`). The analyzer was bundled together with the tool as a separate command:

- `feature-envy` — runs the *Feature Envy* analysis, prints a report containing the Feature Envy instances, and persists the associated cross-file co-change graphs

To showcase these commands, consider the following scenario: we wish to inspect the Feature Envy instances that occurred in the *Kafka* [24] open-source project developed by the Apache Software Foundation. To achieve this, we run the following commands:

Listing 5.1: Kafka analysis commands

```
1 # Clone the remote repository into the current working directory.
2 git clone https://github.com/apache/kafka.git .
3
4 # Persist the history model corresponding to the repository root. The version
5 # control system is detected automatically as 'git'.
6 chronolens-cli persist
7
```

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

```
# Run the 'Feature Envy' analysis and output the resulting report to the
# specified file.
chronolens-cli feature-envy > kafka-feature-envy.json

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

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

The analyzer produces **JSON reports** [25] containing the detected anti-patterns in a specific format described in Listing 5.2. The report consists of a list of file records describing the affected `file` path, the `category` of anti-patterns, the `name` of the detected anti-pattern, and the `value` for the metric used to detect it (in our case, the metric represents the number of methods in the file that were detected as Feature Envy instances).

Additionally, records contain finer-grained information about the file: besides the nominal `featureEnvyCount` metric (equal to the `value` field), the record contains the list of all Feature Envy instances in that file (identified by the unique `function` id), alongside the `enviedFile` (representing the file path envied by that method), the `coupling` with its own file, and the `enviedCoupling` (temporal coupling with the envied file).

Listing 5.2: JSON report template produced by the analyzer

```
[
  {
    "file": <path>,
    "category": "Temporal Coupling Anti-Patterns",
    "name": "Feature Envy",
    "value": <metric-value>,
    "featureEnvyCount": <number-of-feature-envy-instances-in-file>
    "featureEnvyInstances": [
      {
        "function": <function-unique-id>,
        "coupling": <self-coupling>,
        "enviedFile": <envied-file-path>,
        "enviedCoupling": <coupling-with-envied-file>
      },
      ...
    ]
  },
  ...
]
```

5.1.2 Data Visualization

Although the JSON reports generated by the analyzer 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 analysis, we use CHRONOS[2], a commercial tool developed by Endava.

[2] CHRONOS was described in *Quality Assessment in the Tower of Babel*, a presentation given by Radu Marinescu at *ITCamp 2016* [26].

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 rectangle 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 no metric value is 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
- 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 value 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 *Feature Envy* analysis on the *Kafka* project. The resulting image is shown in Figure 5.1. The used thresholds are `lower = 2` and `upper = 4`. 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 *Feature Envy* analysis provides more information than the simple value of a metric, it provides the topological decomposition of a **cross-file co-change graph**. We leveraged a visualization method for cross-file graphs that makes use of the **force layouts** from the *D3.js* third-party library [28]. Unique colors are assigned to the nodes in each file, and to each Feature Envy instance. 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 5.2 presents the force layout for the cross-file co-change graph corresponding to the `ConsumerCoordinator` infringing Java file from the *Kafka* project. Odd-colored nodes (not present in the color legend) are Feature Envy instances (i.e., methods of `ConsumerCoordinator` that have a tighter coupling with one of the three other files present in the cross-file co-change graph).

CHRONOS tree-maps highlight the files with methods that envy other files, whereas D3 force layouts of cross-file co-change 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 subset of files (envying and envied) by themselves.

[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" [27].

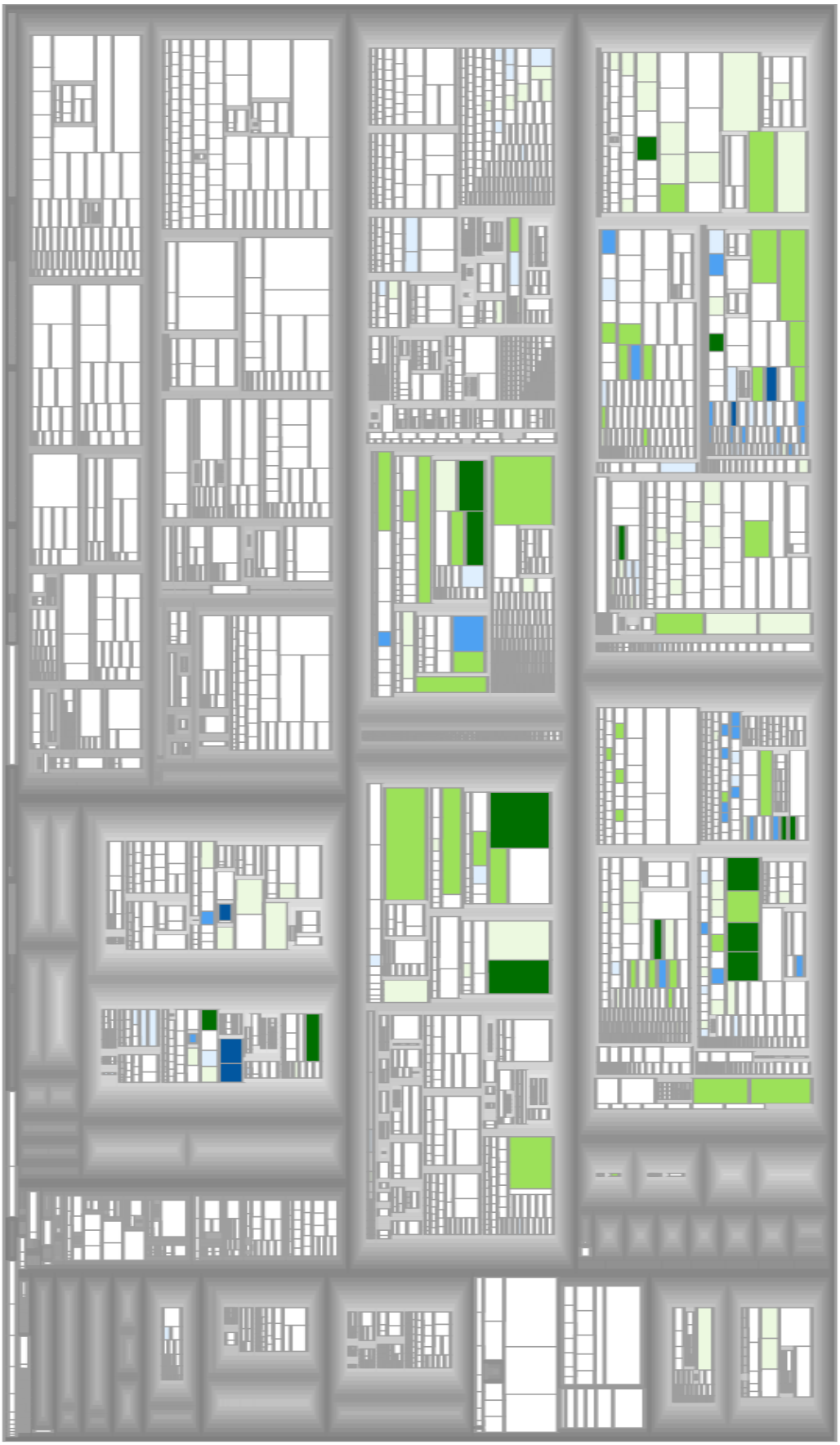

Figure 5.1: Chronos tree-map of Kafka Feature Envy instances

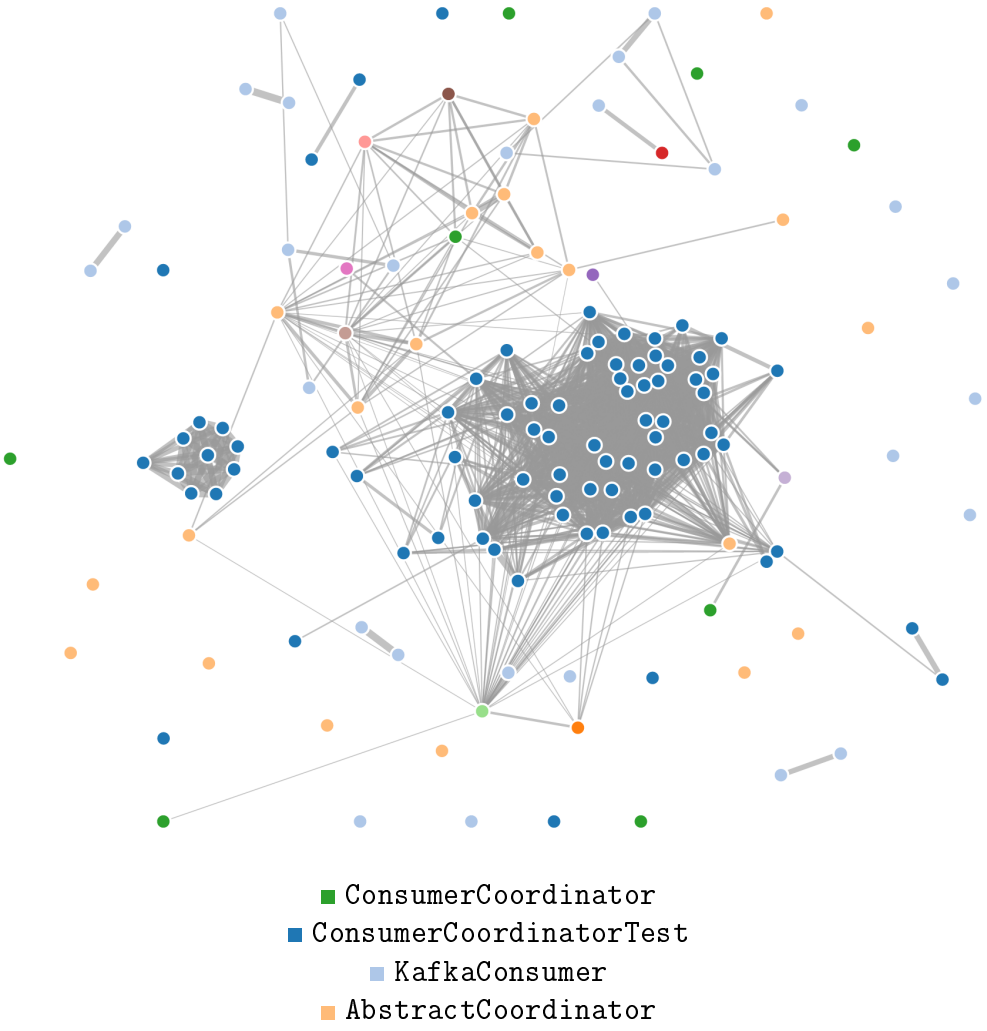

Figure 5.2: D3 force layout of the `ConsumerCoordinator` cross-file co-change graph

5.2 Experimental Results

5.2.1 Analyzed Systems

We analyzed several large open-source projects written in *Java* [23] and versioned in *Git* [29]. We extracted the number of lines of code for each project using the CLOC tool [30]. Table 5.1 presents the main characteristics of the analyzed systems: number of lines of code, number of analyzed revisions, and the short hash code of the head revision (i.e., the latest snapshot at the time of analysis). We focused on the most relevant Feature Envy instances, and filtered out noise and short-lived methods with no meaningful history. Therefore, we applied the ChronoLens filters presented in Table 5.2 across all of our experiments. Additionally, we used the Chronos thresholds presented in Table 5.3 for rendering the tree-maps.

Table 5.1: Analyzed systems' characteristics

System	Lines of Java code	Revisions	Head short hash
Kafka	425256	8418	ad91c5e
Elasticsearch	2033492	15212	4f22f43
Tomcat	348726	22958	76da8de
Hadoop	1808161	22104	0c7b951
Spring	697092	19319	8da049b

Table 5.2: ChronoLens parameters used for experiments

Parameter	Value	Description
`--min-envy-ratio`	1	the *envy ratio* defined in Section 4.2
`--max-envied-files`	1	the *maximum envy count* defined in Section 4.2
`--max-change-set`	100	filters out revisions that change more than 100 files
`--min-revisions`	5	filters out methods and couplings with less than 5 revisions
`--min-coupling`	0.1	filters out temporal couplings less than 0.1
`--min-metric-value`	1	filters out files without Feature Envy instances

Table 5.3: Chronos parameters used for experiments

Parameter	Value	Description
`lower`	2	exclusive maximum envying methods for mild Feature Envy
`upper`	4	exclusive minimum envying methods for severe Feature Envy

Kafka

Let us inspect the Feature Envy instances in the *Kafka* [24] project. Figure 5.1 presents the Chronos tree-map, and Table 5.4 lists the source files with the most methods that envy other files. Most of these files contain methods that envy their corresponding test files, which could be an indication of brittle or fragile tests. In order to gain a better understanding of the results, we will take a closer look at the top "offenders".

Table 5.4: Kafka top Feature Envy instances

File	Feature Envy	Most envied file
ConsumerCoordinator	9	ConsumerCoordinatorTest
DistributedHerder	8	DistributedHerderTest
StreamTask	6	StreamTaskTest
StandaloneHerderTest	5	DistributedHerderTest
SessionWindowedKStreamImpl	5	TimeWindowedKStreamImpl

Table 5.5 presents every Feature Envy instance in the `ConsumerCoordinator` file, and Figure 5.2 presents the associated cross-file co-change graph. By the sum of couplings of methods envying each file, the associated test file, `ConsumerCoordinatorTest`, is the most envied. However, some of the methods (e.g., `sendOffsetCommitRequest` and `sendOffsetFetchRequest`) envy the superclass `AbstractCoordinator`. This is an interesting result, as it indicates a stronger coupling between abstraction and implementation. At a closer inspection, both of these files have over 1000 lines of code, and thus `AbstractCoordinator` is very much part of the implementation. Favoring composition

over inheritance [6] and extracting the pieces of code related to the shared functionality of the superclass might have prevented this cross-file coupling.

Table 5.5: `ConsumerCoordinator` Feature Envy instances

Method	Self-coupling	Envied file	Envied coupling
onJoinComplete	0.58	ConsumerCoordinatorTest	4.69
#handle	1.08	AbstractCoordinator	2
sendOffsetCommitRequest	0.79	AbstractCoordinator	1.98
sendOffsetFetchRequest	0.82	AbstractCoordinator	1.47
commitOffsetAsync	0.35	ConsumerCoordinatorTest	0.81
onJoinPrepare	0.47	ConsumerCoordinatorTest	0.76
refreshCommittedOffsetsIfNeeded	0	KafkaConsumer	0.55

Table 5.6 presents the Feature Envy instances in the `DistributedHerder` file, and Figure 5.3 presents the associated cross-file co-change graph. As in the previous case, most methods envy test files. However, it is curious that some methods envy test files other than their own (e.g., the `StandaloneHerderTest`), and some methods envy sibling classes in the inheritance hierarchy (e.g., `startTask` envies `StandaloneHerder`).

Table 5.6: `DistributedHerder` Feature Envy instances

Method	Self-coupling	Envied file	Envied coupling
#onRevoked	0	DistributedHerderTest	1.61
startTask	0	StandaloneHerder	1
reconfigureConnector	0	WorkerTest	0.91
restartConnector	0	StandaloneHerderTest	0.83
tick	0	DistributedHerderTest	0.58
putConnectorConfig	0	StandaloneHerderTest	0.26
handleRebalanceCompleted	0	DistributedHerderTest	0.23
connectorInfo	0	DistributedHerderTest	0.23

Elasticsearch

Next, let us inspect the Feature Envy instances in the *Elasticsearch* [31] project. Figure 5.4 presents the CHRONOS tree-map, and Table 5.7 lists the source files with the most methods that envy other files. In this case, we note that many of the top offenders are test methods themselves. Others appear to be duplicated, but not identical source files (e.g., `client/` and `x-pack/TrainedModelConfig`, both of which appear to be data classes with similar fields and methods, but the `x-pack` version contains more logic, most likely to satisfy the `Writeable` interface, which it implements in addition to the `client/` version). However, `DatafeedConfig` doesn't fit either of the previously mentioned patterns, therefore we will inspect it separetely.

Table 5.7: Elasticsearch top Feature Envy instances

File	Feature Envy	Most envied file
TextFieldMapperTests	8	RangeFieldMapperTests
KeywordFieldMapperTests	8	RangeFieldMapperTests
DatafeedConfig	7	DatafeedUpdate
ScaledFloatFieldMapperTests	5	RangeFieldMapperTests
client/TrainedModelConfig	4	x-pack/TrainedModelConfig
client/JobUpdate	4	x-pack/JobUpdate
TokenService	4	TokenServiceTests
MachineLearningIT	3	Classification

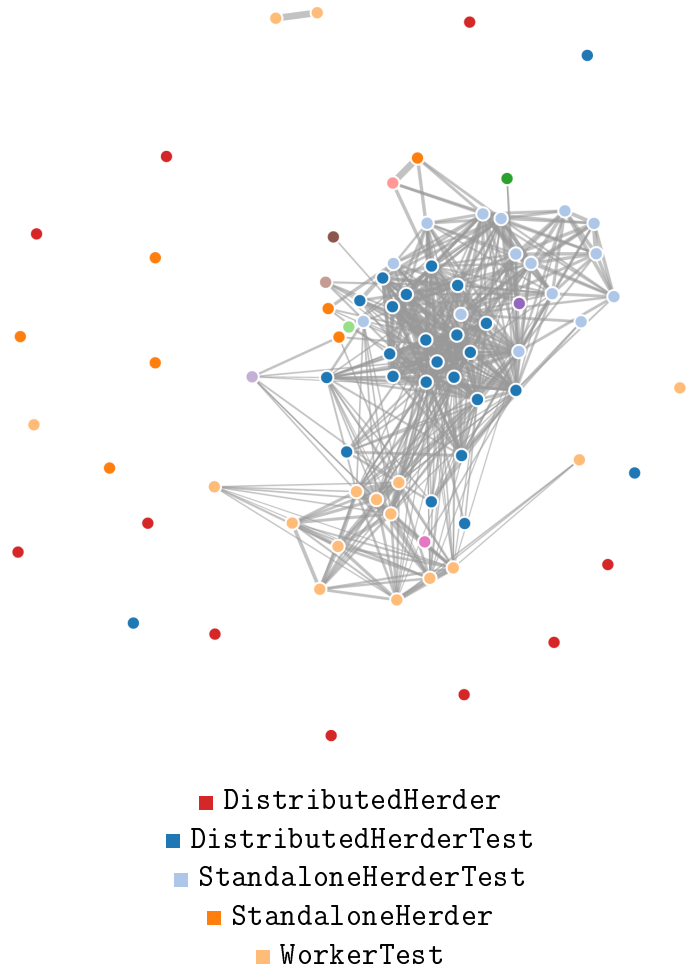

Figure 5.3: Cross-file co-change graph of `DistributedHerder`

Table 5.8 presents the Feature Envy instances in the `DatafeedConfig` file, and Figure 5.5 presents the associated cross-file co-change graph. Note that unlike the previously investigated infringing files, only a single other file is envied by all the Feature Envy instances: `DatafeedUpdate`. According to its documentation, a "datafeed update contains partial properties to update a {@link DatafeedConfig}. The main difference between this class and {@link DatafeedConfig} is that here all fields are nullable". Therefore, the strong coupling is unsurprising: these are data classes holding the same data, with different nullability constraints applied to one of them, which is also used to modify existing object instances of the other.

Table 5.8: `DatafeedConfig` Feature Envy instances

Method	Self-coupling	Envied file	Envied coupling
hashCode	3.15	DatafeedUpdate	4.07
equals	3.15	DatafeedUpdate	4.07
#Builder	2.97	DatafeedUpdate	3.71
#build	2.72	DatafeedUpdate	2.9
createParser	2.37	DatafeedUpdate	2.66
writeTo	2.36	DatafeedUpdate	2.59
DatafeedConfig	2.22	DatafeedUpdate	2.38

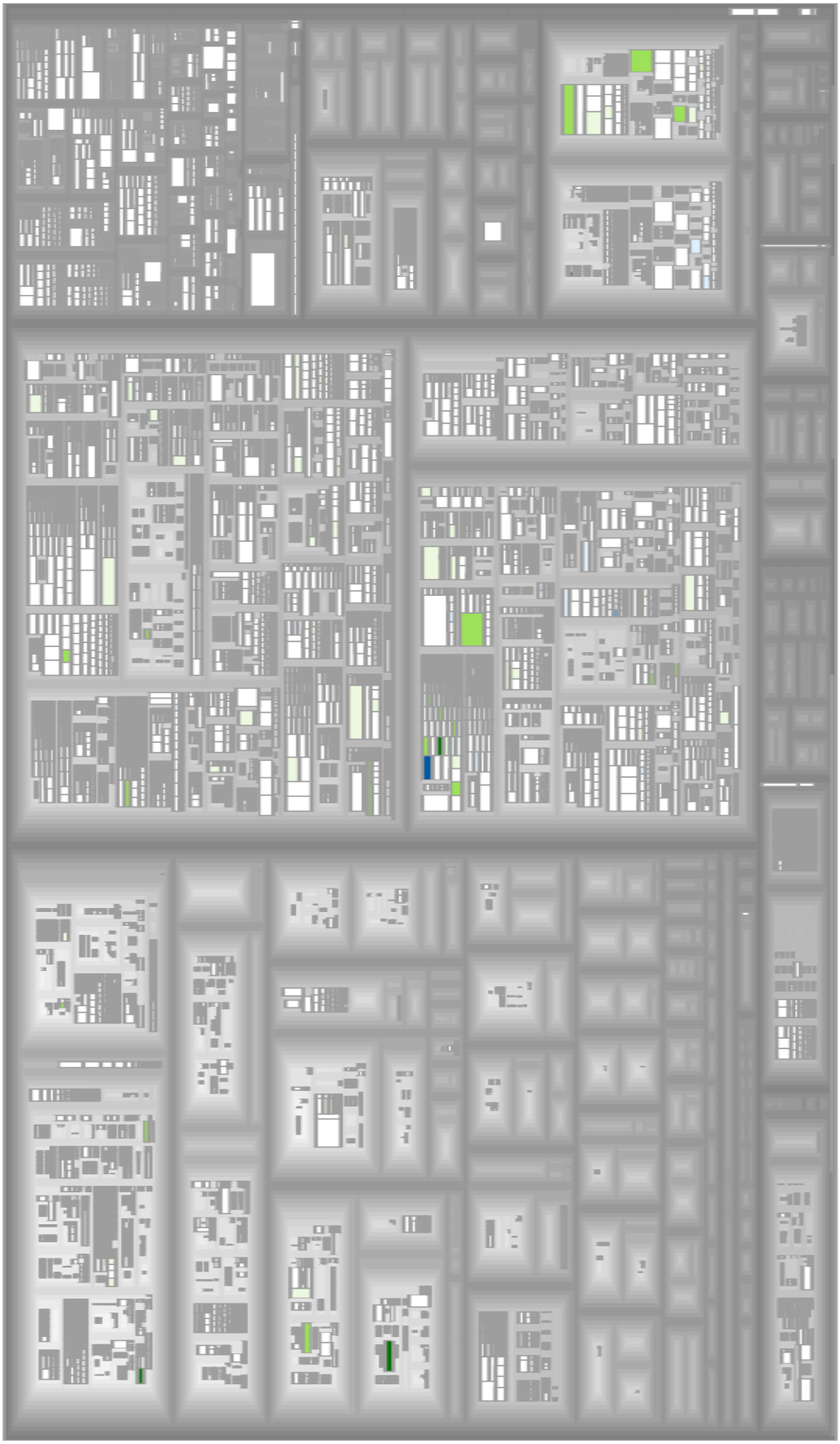

Figure 5.4: CHRONOS tree-map of Elasticsearch Feature Envy instances

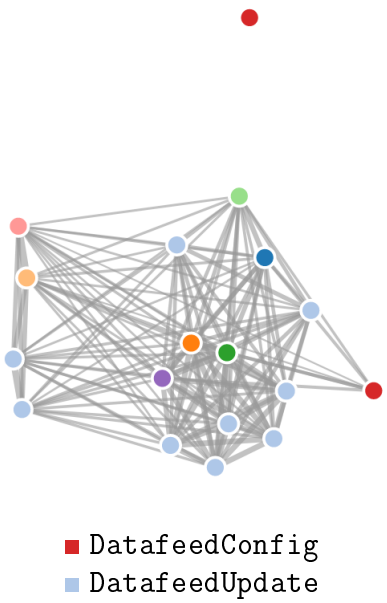

Figure 5.5: Cross-file co-change graph of `DatafeedConfig`

Tomcat

Now, let us inspect the Feature Envy instances in the *Tomcat* [32] project. Figure 5.6 presents the CHRONOS tree-map, and Table 5.9 lists the source files with the most methods that envy other files. Note that the top offender is an auto-generated file, which envies another auto-generated file. We ignore this instance, because the programmer has no control over how these files change, or how they are structured. In the case of `NioEndpoint` and `Nio2Endpoint`, both are thread pools handling socket operations, with the difference that one uses a synchronous interface, and the other uses an asynchronous interface. Therefore, it is expected that some methods could envy the other file (e.g., a change in the socket implementation could span changes across both the synchronous and asynchronous interfaces). A more interesting file is `StandardContext`, thus we will take a closer look at it.

Table 5.9: Tomcat top Feature Envy instances

File	Feature Envy	Most envied files
ELParserTokenManager	8	ELParser
StandardContext	5	ContainerBase
NioEndpoint	5	Nio2Endpoint
GenericKeyedObjectPool	4	GenericObjectPool
AprEndpoint	4	NioEndpoint
GenericObjectPool	3	GenericKeyedObjectPool
TestParser	3	TestCompiler
Nio2Endpoint	3	NioEndpoint
ManagerServlet	3	HostManagerServlet

Table 5.10 presents the Feature Envy instances in the `StandardContext` file, and Figure 5.7 presents the associated cross-file co-change graph. The most envied file is the

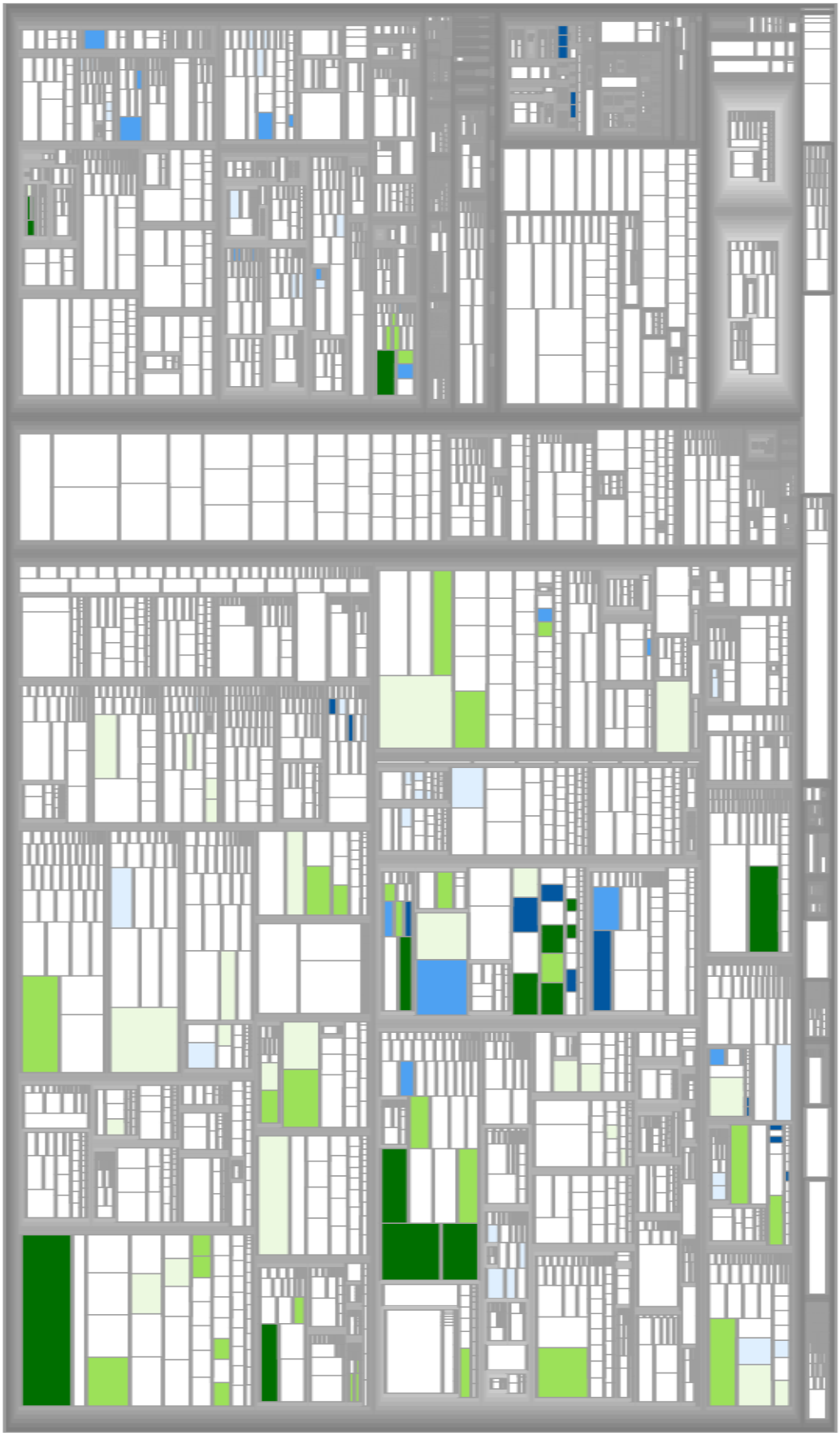

Figure 5.6: CHRONOS tree-map of Tomcat Feature Envy instances

superclass `ContainerBase`, as was the case for a few methods of `ConsumerCoordinator` from Kafka. However, a few other methods have a mild envy for other files. For example, the `listenerStart` method envies the sibling `StandardWrapper` class. Another interesting instance is the `resourceStart` method, which starts up the `WebResourceRoot`, which implements the `Lifecycle` interface, which, in turn, notifies `LifecycleListeners` of various events. The `ContextConfig` is one such `LifecycleListener` implementation. It is difficult to assess the semantics or relationships between these classes, but a plausible hypothesis is that changes in the `resourceStart` method implementation could trigger a change in how the `ContextConfig` implementaton reacts to the corresponding lifecycle event.

Table 5.10: `StandardContext` Feature Envy instances

Method	Self-coupling	Envied file	Envied coupling
destroyInternal	0.65	ContainerBase	1.06
stopInternal	0.4	ContainerBase	0.57
listenerStop	0.28	ApplicationFilterConfig	0.49
listenerStart	0.28	StandardWrapper	0.29
resourcesStart	0	ContextConfig	0.15

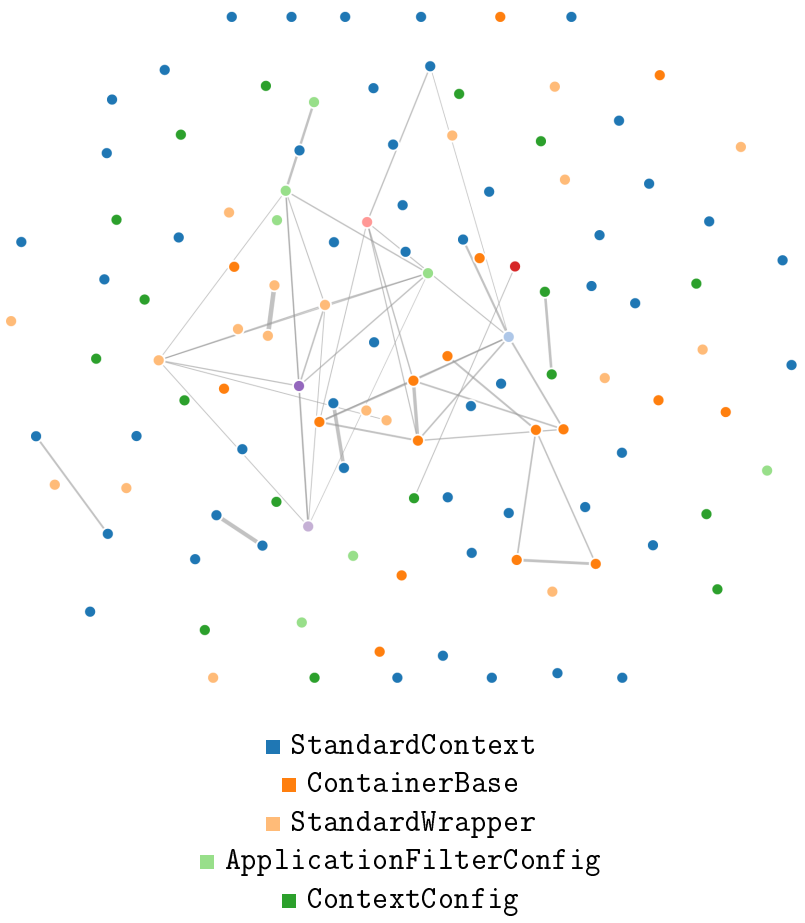

Figure 5.7: Cross-file co-change graph of `StandardContext`

Hadoop

Next, we will inspect the Feature Envy instances in the *Hadoop* [33] project. Figure 5.8 presents the CHRONOS tree-map, and Table 5.11 lists the source files with the most methods that envy other files. As was the case for the Elasticsearch project, some test utilities are among the top offenders. There are also cases where a class envies a sibling class (e.g., `ParentQueue` and `LeafQueue`, or `FifoScheduler` and `CapacityScheduler`). An interesting file, which doesn't fit any of the previously observed patterns, is `FSNamesystem`. Therefore, we will take a closer look at it.

Table 5.11: Hadoop top Feature Envy instances

File	Feature Envy	Most envied file
TestReservations	7	TestLeafQueue
ParentQueue	7	LeafQueue
TestYarnCLI	7	NodeCLI
FSNamesystem	6	BlockIdManager
FifoScheduler	6	CapacityScheduler
RMAppAttemptImpl	6	TestRMAppAttemptTransitions

Table 5.12 presents the Feature Envy instances in the `FSNamesystem` file, and Figure 5.9 presents the associated cross-file co-change graph. Once again, we notice the pattern where some methods envy test files (although not the test files associated to their own class). However, the most interesting instance is the `nextGenerationStamp` method, which has the strongest coupling with `BlockIdManager`. This method delegates its implementation to the `BlockManager`, which contains a `BlockIdManager`, which, according to its documentation, "allocates the generation stamps and the block ID". Thus, we can conclude that the method is clearly related to the class it envies. Curiously, the method is quite short at the head revision (less than 20 lines of code). At some point, the method's implementation resided entirely in the `FSNamesystem` (e.g., in commit `6770de7`), and only later was its implementation delegated to the `BlockIdManager` (in commit `14a9ff1`). Although it is difficult to inspect the entire history of this method (its containing file has many revisions that touched it), we believe a plausible hypothesis is that the method did envy the `BlockIdManager` at some point, and was later on refactored and suffered few changes afterwards. In fact, this highlights one of the drawbacks of our analyzer: even if a method is refactored such that the envying logic is moved to the envied file, the history (and temporal coupling with the envied file) persists. If the method suffers few changes afterwards because it became stable as a result of refactoring and does not offset its previous envying history, it will be falsely reported as a Feature Envy instance.

Table 5.12: `FSNamesystem` Feature Envy instances

Method	Self-coupling	Envied file	Envied coupling
nextGenerationStamp	0	BlockIdManager	2.59
internalReleaseLease	0.14	BlockManager	0.45
#clearCorruptLazyPersistFiles	0.1	NamenodeFsck	0.22
updatePipelineInternal	0.13	TestCommitBlockSynchronization	0.2
clear	0.11	FSImageFormat	0.15
getLiveNodes	0	TestNameNodeMxBean	0.13

Figure 5.8: CHRONOS tree-map of Hadoop Feature Envy instances

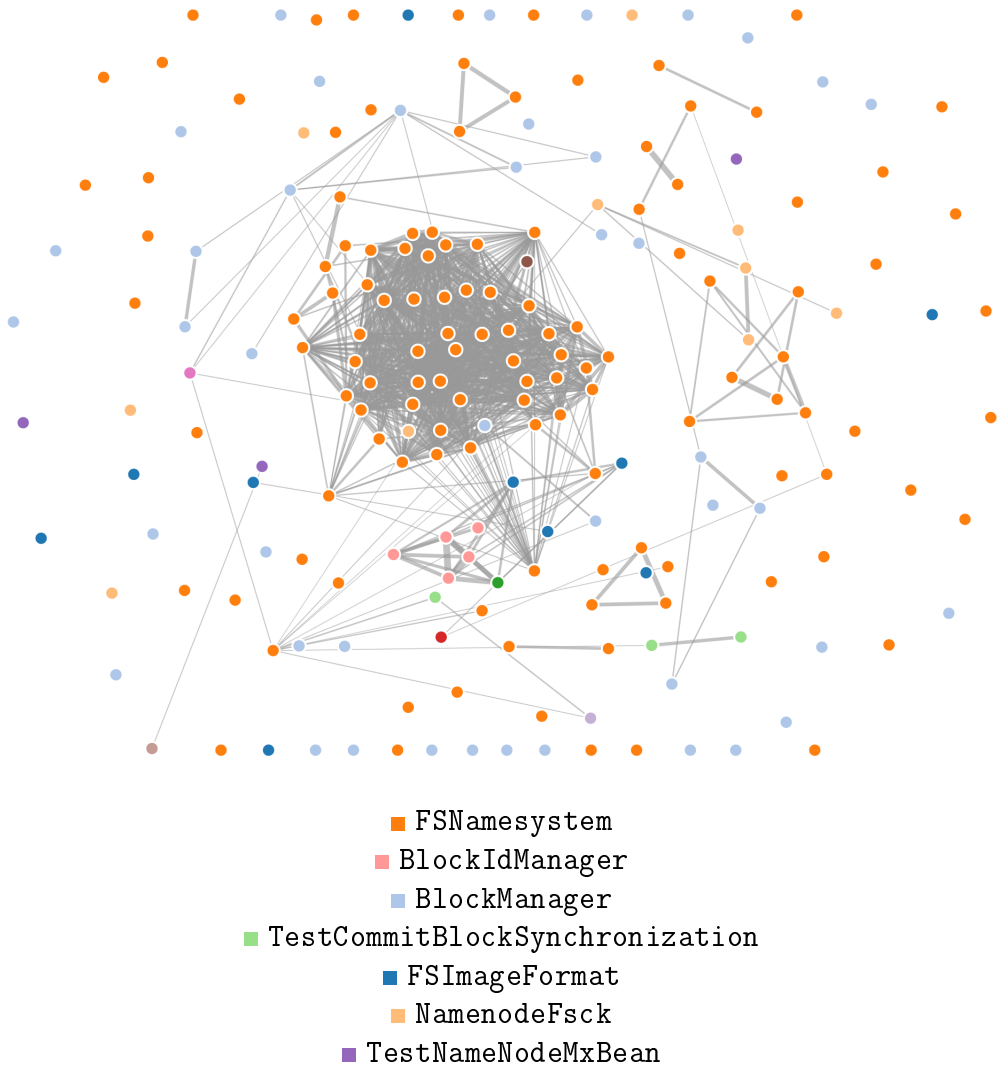

Figure 5.9: Cross-file co-change graph of `FSNamesystem`

Spring

Finally, we will inspect the Feature Envy instances in the *Spring Framework* [34] project. Figure 5.10 presents the CHRONOS tree-map, and Table 5.13 lists the source files with the most methods that envy other files. As previously noted, there are test files both among the top offenders and among envied files. There are, however, some instances that seem related to the corresponding envied file. Two of them are `AnnotationUtils` and `AbstractListenerReadPublisher`, therefore we will inspect them more closely.

Table 5.14 presents Feature Envy instances from `AbstractListenerReadPublisher`, and Figure 5.11 presents the associated cross-file co-change graph. There is a very strong coupling with the only envied file. In fact, the majority of the methods with a significant history (i.e., at least 5 revisions) are Feature Envy instances. They are closely related, as they model consumers/producers of HTTP requests/responses. Some of the revisions that contributed to the envy handled logging (commits `42e4ca1`, `2874dd7`, `5dc49b1`, `c1b191e`),

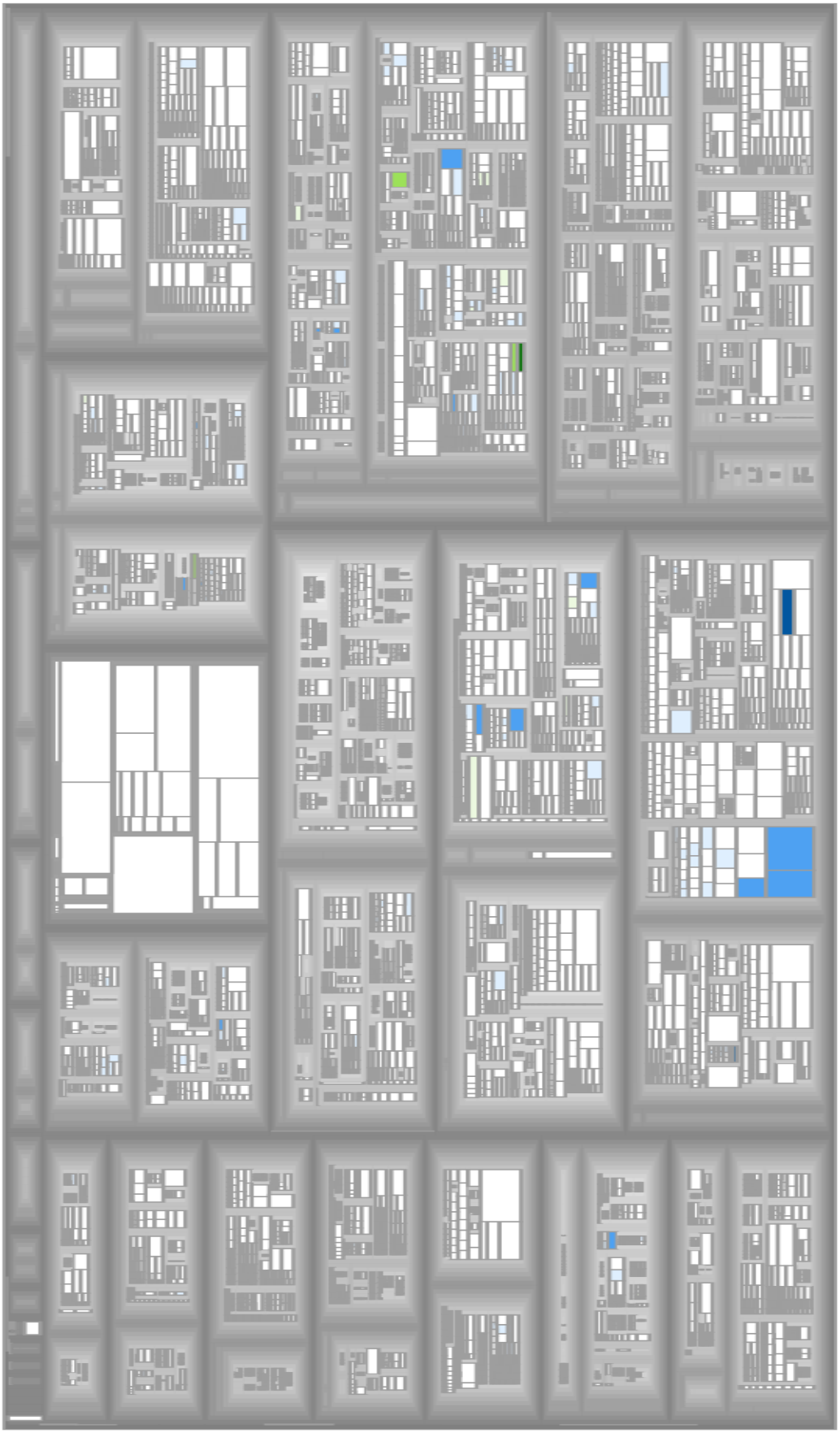

Figure 5.10: CHRONOS tree-map of Spring Feature Envy instances

Table 5.13: Spring top Feature Envy instances

File	Feature Envy	Most envied file
AbstractListenerReadPublisher	6	AbstractListenerWriteProcessor
AnnotationUtils	5	AnnotatedElementUtils
StringDecoderTests	4	Jaxb2XmlDecoderTests
ViewResolutionResultHandlerTests	3	RequestMappingInfoTests
ClassWriter	3	MethodWriter
MethodWriter	3	Type
PatternsRequestCondition	3	PatternsRequestConditionTests
AnnotationDrivenBeanDefinitionParser	3	WebMvcConfigurationSupport

releasing resources (commit `862dd23`), etc. It seems that a significant part of the temporal coupling was instilled by cross-cutting concerns and refactorings. We acknowledge that it is possible for such a cross-cutting concern to impact fewer methods in the envying file than in the envied file, thus leading to a disproportionate contribution to the temporal couplings within and across files, and incorrectly labeling this situation as Feature Envy.

Table 5.14: `AbstractListenerReadPublisher` Feature Envy instances

Method	Self-coupling	Envied file	Envied coupling
onError	3.09	AbstractListenerWriteProcessor	4.23
onAllDataRead	3.09	AbstractListenerWriteProcessor	4.23
request	2.79	AbstractListenerWriteProcessor	3.95
cancel	2.79	AbstractListenerWriteProcessor	3.95
readAndPublish	1.86	AbstractListenerWriteProcessor	2.2
onDataAvailable	1.42	AbstractListenerWriteProcessor	1.54

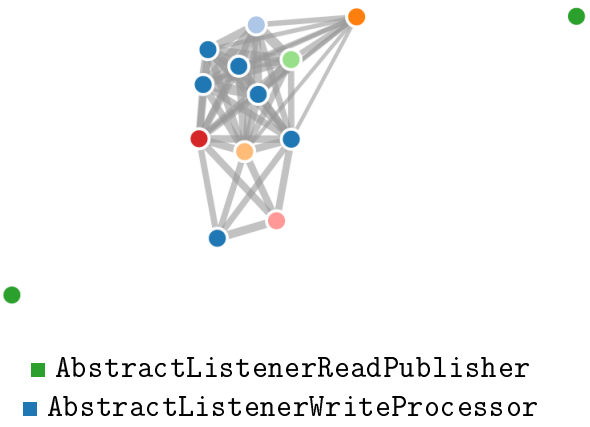

Figure 5.11: Cross-file co-change graph of `AbstractListenerReadPublisher`

Table 5.15 presents the Feature Envy instances in the `AnnotationUtils` file, and Figure 5.12 presents the associated cross-file co-change graph. There is a very strong coupling with the only envied file, `AnnotatedElementUtils`, just as was the case for `AbstractListenerReadPublisher`. A difference, however, is that both files have many methods with a significant history that do not envy the other file (even if there is non-trivial coupling). Two revisions stand out: commit `a14bfe9` and commit `37255af`, which migrate a significant number of utilities from these two files to `MergedAnnotation` and `MergedAnnotations`, thus incurring changes across many methods of the files. The files

have a significant history, which makes an accurate assessment difficult, but it is plausible that the reported Feature Envy instances were genuine at some point, and were later refactored, just as was the case for the `nextGenerationStamp` method of `FSNamesystem` from Hadoop.

Table 5.15: `AnnotationUtils` Feature Envy instances

Method	Self-coupling	Envied file	Envied coupling
isAnnotationDeclaredLocally	1.66	AnnotatedElementUtils	3.19
synthesizeAnnotation	2.87	AnnotatedElementUtils	3.19
findAnnotationDeclaringClassForTypes	2.47	AnnotatedElementUtils	2.9
findAnnotationDeclaringClass	2.26	AnnotatedElementUtils	2.6
isAnnotationInherited	1.3	AnnotatedElementUtils	2.3

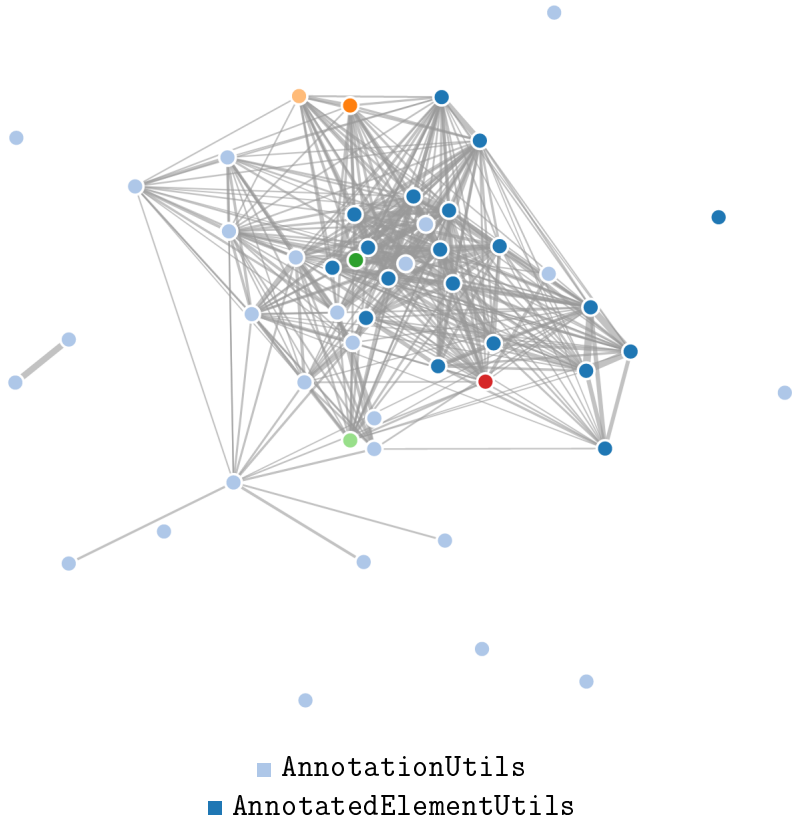

Figure 5.12: Cross-file co-change graph of `AnnotationUtils`

5.2.2 Multiple Envied Files

We mentioned in Section 4.2 that methods that envy multiple files could be an interesting anti-pattern to investigate, and introduced the **maximum envy count** filter to control how many envied files are reported for a method. Therefore, we repeated the analyses with the `--max-envied-files=20` option.

A pattern that we observed with this option is that significantly many more test files envy others and are envied, although this seems to be more pronounced in Kafka,

Hadoop and Elasticsearch than in Spring and Tomcat. As an example, Table 5.16 presents how the top offenders change for the Kafka project. In comparison, Table 5.17 presents the previous top offenders (when only one envied file was reported per method) with the results where multiple envies per method are allowed. We note that for these files, although a few methods do envy multiple files, the differences are minor, whereas the new test files that top the offenders list have a significant increase in the number of Feature Envy instances.

Table 5.16: Kafka top Feature Envy instances with multiple envies

File	Feature Envy
KStreamTransformValuesTest	34
KStreamMapValuesTest	32
KStreamFlatMapValuesTest	31
KStreamTransformTest	28
KStreamFilterTest	28
KTableMapValuesTest	26
AbstractStreamTest	22
KStreamKStreamLeftJoinTest	21
KStreamSelectKeyTest	20
KStreamMapTest	20
KStreamFlatMapTest	20
KTableMapKeysTest	20
KStreamBranchTest	20
KStreamForeachTest	20

Table 5.17: Kafka previous top Feature Envy instances with multiple envies

File	Feature Envy	Most envied file
ConsumerCoordinator	12	ConsumerCoordinatorTest
DistributedHerder	16	DistributedHerderTest
StreamTask	10	StreamTaskTest
StandaloneHerderTest	7	DistributedHerderTest
SessionWindowedKStreamImpl	7	TimeWindowedKStreamImpl

Let us take a closer look at the `DistributedHerder` file, which had the most significant increase in the number of Feature Envy instances among the previous top offenders. Table 5.18 presents the Feature Envy instances (bold lines indicate new instances). The associated cross-file co-change graph is unchanged from Figure 5.3, because no new envied files are introduced, and the temporal couplings for the additional envies were already captured. We note that 7 out of 8 new Feature Envy instances are methods that envy test files.

In the case of the Tomcat project, all of the new top offenders (highlighted in bold in Table 5.19) consist of a single method that accounts for almost all of the Feature Envy instances. The previous top offenders are nearly unchanged.

Let us take a closer look at the `PoolableConnectionFactory` file, which had the most significant increase. Table 5.20 presents the Feature Envy instances, and Figure 5.13 presents the associated cross-file co-change graph. The red and blue nodes are the only methods of `PoolableConnectionFactory`, both of which are Feature Envy instances. The class is quite peculiar, as most of its methods have an insignificant history (fewer than 5 revisions), and the only methods with a significant history envy other files. This could be explained by the fact that the class was relatively stable, having been changed in less than 20 commits, with most of its methods unaffected, and thus excluded from

Table 5.18: `DistributedHerder` Feature Envy instances with multiple envies

Method	Self-coupling	Envied file	Envied coupling
#onRevoked	0	DistributedHerderTest	1.61
startTask	0	StandaloneHerder	1
reconfigureConnector	0	WorkerTest	0.91
startTask	**0**	**StandaloneHerderTest**	**0.83**
restartConnector	0	StandaloneHerderTest	0.83
tick	0	DistributedHerderTest	0.58
tick	**0**	**StandaloneHerderTest**	**0.52**
startTask	**0**	**DistributedHerderTest**	**0.5**
restartConnector	**0**	**DistributedHerderTest**	**0.5**
reconfigureConnector	**0**	**StandaloneHerder**	**0.35**
reconfigureConnector	**0**	**DistributedHerderTest**	**0.35**
putConnectorConfig	0	StandaloneHerderTest	0.26
reconfigureConnector	**0**	**StandaloneHerderTest**	**0.24**
handleRebalanceCompleted	0	DistributedHerderTest	0.23
connectorInfo	0	DistributedHerderTest	0.23
putConnectorConfig	**0**	**DistributedHerderTest**	**0.21**

Table 5.19: Tomcat top Feature Envy instances with multiple envies

File	Feature Envy	Most envied files
PoolableConnectionFactory	**10**	**BasicDataSource**
ELParserTokenManager	8	ELParser
DelegatingStatement	**8**	**DelegatingResultSet**
DelegatingConnection	**8**	**InstanceKeyDataSourceFactory**
NioEndpoint	7	Nio2Endpoint
PoolableCallableStatement	**7**	**DelegatingConnection**
StandardContext	6	ContainerBase
AprEndpoint	6	NioEndpoint
PoolingDataSource	**5**	**PStmtKey**

temporal coupling computation. This led to the two outlier methods unable to "couple" to any other method in their own file, and thus, implicitly to envying any other (possibly unrelated) file that happened to change at the same time.

Table 5.20: `PoolableConnectionFactory` Feature Envy instances with multiple envies

Method	Self-coupling	Envied file	Envied coupling
makeObject	0	BasicDataSource	1.63
makeObject	0	PStmtKey	1
activateObject	0	BasicDataSource	0.62
makeObject	0	SharedPoolDataSource	0.55
makeObject	0	DriverAdapterCPDS	0.55
makeObject	0	BasicDataSourceFactory	0.5
makeObject	0	InstanceKeyDataSource	0.45
activateObject	0	BasicDataSourceFactory	0.41
makeObject	0	PoolableCallableStatement	0.41
makeObject	0	DelegatingStatement	0.38

Allowing multiple envies per method leads to interesting results and patterns, which vary across projects: many test files with a significant increase in the number of Feature Envy instances, some files that consist of only a few methods that envy many other files, and some top offenders that remain largely unchanged (perhaps with the exception of envying some additional test files). However, we do not believe the results were more practically relevant than when reporting only the most envied file for each method.

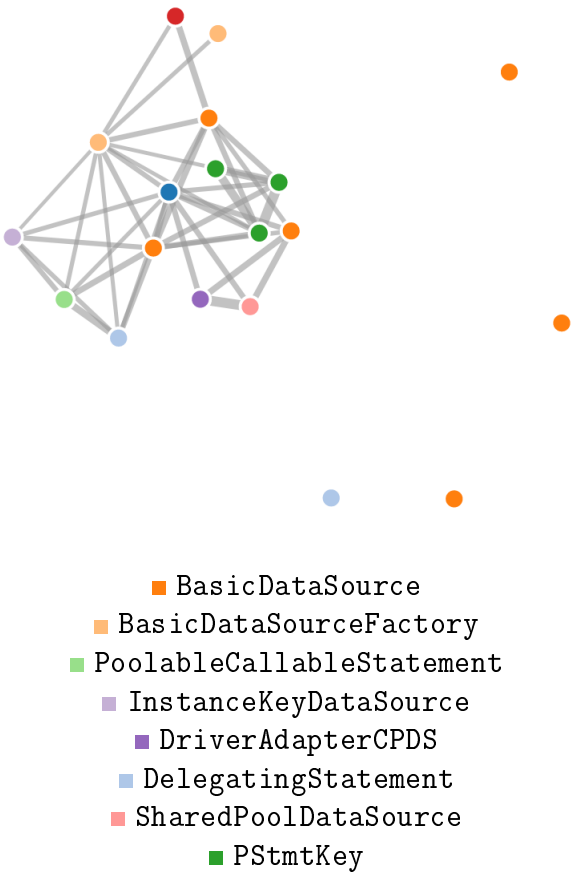

Figure 5.13: Cross-file co-change graph of `PoolableConnectionFactory`

5.2.3 Observed Patterns

We analyzed several large open-source systems and presented some of the interesting findings. We conclude by summarizing the patterns we observed.

Test Envy

Test files envy and are envied. When a method envies its associated test file, it may be an indication of fragile or brittle tests. Interestingly, some methods envy test files other than their own, suggesting an indirect coupling (a hypothesis could be that a change in the envying method leads to a different behavior in a different class that depends on the method, which in turn requires changing the associated tests). This pattern is by far the most common one, and occurred across all the analyzed systems.

Interestingly, if we report multiple files envied by a single method, this pattern is amplified significantly in some projects (particularly in Kafka, Hadoop and Elasticsearch).

Examples: `ConsumerCoordinator` and `DistributedHerder` from the Kafka project, `FSNamesystem` from Hadoop.

Sibling Envy

Some methods envy sibling classes in the inheritance hierarchy. This could be an indication of a missing abstraction, where some logic could be extracted and reused across the envying and envied classes to eliminate the coupling between the sibling implementations.

Examples: `ParentQueue` and `FifoScheduler` from Hadoop, `DistributedHerder` from Kafka, `StandardContext` from Tomcat.

Parent Envy

Some methods envy their superclasses, indicating an abnormal coupling between abstraction and implementation. We noticed that for these instances, the superclass is large (over 1000 lines of code), thus blurring the line between abstraction and implementation. It is possible that this coupling could be eliminated by favoring composition over inheritance [6] for reusing logic shared between the parent and children classes.

Examples: `ConsumerCoordinator` from Kafka, `StandardContext` from Tomcat.

Cross-Cutting Concerns

Cross-cutting concerns represent functionalities that affect many parts of a software system and cannot be decomposed from the affected parts (e.g., logging). The implementation of such a cross-cutting concern could contribute significantly to a stable method's history. If these changes affect a few methods in one file and many methods in another file, they will lead to a disproportionate contribution to the temporal couplings within and across the files. Thus, some methods may be mislabeled as Feature Envy instances.

Examples: `AbstractListenerReadPublisher` from Spring.

Refactored Envy

It is possible for a method to genuinely envy another class initially, and to later be refactored, having its envying logic moved to the appropriate location. However, the history of the method persists, and if it becomes stable as a result of the refactoring (i.e., does not change significantly in order to offset the previous history), it will be mislabeled as a Feature Envy instance. This highlights one of the drawbacks of our analysis. In order to mitigate this in the future, we may consider adjusting the temporal coupling metric. For example, when computing the history of a method or the history of two methods that change together, instead of simply counting the number of revisions, we may assign a weight between 0 and 1 for every revision that would decay over time. This would maintain the temporal and semantic aspect of our analysis, but would focus on methods that changed recently (thus reporting "acute" Feature Envy instances), and would "forget" the older history that may be less relevant. A potential risk with this approach is filtering out "chronic" Feature Envy instances (i.e., methods that changed significantly with another file a long time ago, but then became stable, even if they were not refactored). This may be an acceptable risk, as it is questionable if stable methods are problematic, even if they exhibit some design anti-patterns. The decay function would have to be chosen and tuned carefully in order to maximize the relevance of the findings.

Examples: `FSNamesystem` from Hadoop, `AnnotationUtils` from Spring.

Auto-Generated Envy

Methods from auto-generated files sometimes envy other auto-generated files. We ignore these instances because the programmer has no control over how these files are generated, how they are structured, or how they change. At best, the result indicates that the code generation tool does not apply best practices for its generated code.

Examples: `ELParserTokenManager` from Tomcat.

6 Conclusions, Contributions and Future Work

6.1 Summary

We aimed to detect design flaws by inspecting the history and evolution of software systems, and in particular, to detect instances of the *Feature Envy* [2] anti-pattern. We leveraged our prior work [1] to build an automated analyzer that reports methods that envy other source files using temporal coupling, and visualizes the relationship between these methods, their own source file, and the envied source files.

In order to validate our analysis, we inspected several large open-source systems, and presented the general patterns and interesting findings. In particular, we noticed that it is common for test files to envy and to be envied. Another pattern we noticed is envy between sibling classes in an inheritance hierarchy. We discovered some limitations of our analysis, such as when a method is refactored and becomes stable, it will still be reported as a Feature Envy instance even if the issue has been fixed, or that cross-cutting concerns could impact specific groups of methods in multiple otherwise stable classes, leading to mislabeling as Feature Envy. If we report multiple files envied by a single method, we noticed that oftentimes the top "offenders" do not change their metrics significantly, but in some projects, other files can reach the top through few methods that envy many files, and the pattern where test files envy and are envied may become much more significant.

6.2 Future Work

The ChronoLens framework [1] has great potential in facilitating the development of software evolution analyzers, and to conclude, we propose several research directions that can continue to be pursued:

- use other metrics or formulas for temporal coupling when detecting Feature Envy instances and compare the quality of the findings
- considering that Feature Envy occurs when a method envies one other class in particular, we may consider excluding methods that envy multiple files and investigate how the analysis results change
- develop further analyzers based on temporal coupling, such as for detecting instances of the *Shotgun Surgery* anti-pattern [2], in which a method frequently changes together with many other methods spread throughout many source files
- develop further analyzers, not necessarily related to temporal coupling, for example tracking down code duplication to its origin, how it spread across multiple methods over time, how duplication instances diverged, etc.

References

[1] Andrei Heidelbacher and Radu Marinescu. *Detecting Anti-Patterns by Analyzing the Evolution of Software Systems*. Timişoara, Timiş, Romania, 2018.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] *Checkstyle*. Version 8.44. July 25, 2021. URL: https://github.com/checkstyle/checkstyle.

[13] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. 1st edition. Springer Publishing Company, Incorporated, 2010. ISBN: 3642063748.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] JetBrains s.r.o. *Kotlin Language Documentation 1.5.20*. 2021. URL: https://kotlinlang.org/docs/kotlin-reference.pdf (visited on 07/25/2021).

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

[24] Apache Software Foundation. *Kafka*. Aug. 10, 2021. URL: https://github.com/apache/kafka.

[25] ECMA International. *The JSON Data Interchange Syntax*. Dec. 2017. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (visited on 07/25/2021).

[26] 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).

[27] 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.

[28] Mike Bostock. *D3.js*. Version 4.13.0. July 25, 2021. URL: https://github.com/d3/d3.

[29] *Git reference manual*. URL: https://kernel.org/pub/software/scm/git/docs/ (visited on 07/25/2021).

[30] Al Danial. *CLOC*. Version v1.90. May 18, 2021. URL: https://github.com/AlDanial/cloc.

[31] Elastic. *Elasticsearch*. Aug. 10, 2021. URL: https://github.com/elastic/elasticsearch.

[32] Apache Software Foundation. *Tomcat*. Aug. 10, 2021. URL: https://github.com/apache/tomcat.

[33] Apache Software Foundation. *Hadoop*. Aug. 10, 2021. URL: https://github.com/apache/hadoop.

[34] Pivotal Software. *Spring Framework*. Aug. 10, 2021. URL: https://github.com/spring-projects/spring-framework.