INSTITUTO
UNIVERSITÁRIO
DE LISBOA

Collaborative Code Editing with Continuous Integration

Afonso Pinheiro Sampaio

Master's in Computer Science and Engineering

Supervisor:
PhD André Leal Santos, Assistant Professor,
Iscte - Instituto Universitário de Lisboa

Co-Supervisor:
PhD Sancho Moura Oliveira, Associate Professor,
Iscte - Instituto Universitário de Lisboa

October, 2024

TECHNOLOGY
AND ARCHITECTURE

Department of Information Science and Technology

Collaborative Code Editing with Continuous Integration

Afonso Pinheiro Sampaio

Master's in Computer Science and Engineering

Supervisor:
PhD André Leal Santos, Assistant Professor,
Iscte - Instituto Universitário de Lisboa

Co-Supervisor:
PhD Sancho Moura Oliveira, Associate Professor,
Iscte - Instituto Universitário de Lisboa

October, 2024

Acknowledgments

I have had a lot of support and help while working on this dissertation.

I would like to thank my supervisors, Professor André Santos and Professor Sancho Oliveira, for all their support during the development of this dissertation.

I would like to thank my parents, Pedro and Cláudia, for all their support and for giving me everything I could ever wish for during my academic journey.

Finally, I would like to thank my close friends for all their enthusiasm, support, reassuring words during the hardest times and, most importantly, for believing in me.

Resumo

No desenvolvimento de software contemporâneo, a colaboração entre programadores é um aspeto fundamental para a gestão e evolução de repositórios de código. Os sistemas de controlo de versões, como o Git, facilitam a colaboração, permitindo a integração de alterações independentes num ramo de desenvolvimento principal. No entanto, a ocorrência de modificações paralelas dá frequentemente origem a conflitos de *merge*, os quais podem perturbar os fluxos de trabalho e atrasar os prazos de desenvolvimento. Embora ferramentas de automatização possam resolver alguns conflitos, a intervenção manual é frequentemente necessária para resolver casos de alterações sobrepostas. Para evitar a acumulação destes conflitos, foi proposta a prática da integração contínua como forma de incentivar a integração regular do código, com o objetivo de melhorar a gestão dos conflitos.

Esta tese propõe um protocolo de propagação de alterações de código que permite a integração de práticas de integração contínua diretamente em ambientes de programação colaborativa. Esta abordagem permite a deteção precoce de conflitos, tendo potencial para simplificar o processo de *merging*. Foi desenvolvido o protótipo Javardair, com o objetivo de validar esta abordagem para Java, empregando uma metodologia baseada em transformações. Esta abordagem representa as modificações do código como transformações estruturadas a um nível semântico com base numa árvore de sintaxe abstrata, em vez de um nível textual. Isto permite identificar conflitos de forma mais precisa, uma vez que se baseia na semântica do código e não apenas em comparações linha a linha, permitindo também aplicar modificações de forma mais eficiente.

Palavras-Chave: *Integração contínua, programação colaborativa, deteção de conflitos, transformações*

Abstract

In contemporary software development, collaboration among developers is a fundamental aspect of the management and evolution of codebases. Version control systems such as Git facilitate collaboration by enabling the integration of version branches into a primary development branch. However, the occurrence of parallel modifications frequently gives rise to merge conflicts, which have the potential to disrupt workflows and delay development timelines. Although automated tools can address some conflicts, manual intervention is frequently necessary to resolve instances of overlapping changes. In order to prevent the accumulation of these conflicts, the practice of continuous integration has been proposed as a means of encouraging frequent code integration, with the aim of improving conflict management.

This dissertation proposes a protocol for propagation of code changes to facilitate the integration of continuous integration practices directly into collaborative coding environments. This approach enables the early detection of conflicts and streamlines the merge process. The Javardair prototype was developed with the objective of validating this approach for Java, employing a transformation-based methodology. In contrast to conventional version control systems, this approach represents code modifications as structured, semantic-level transformations within an abstract syntax tree instead of a textual level. This facilitates a more precise identification of conflicts, as it is based on code semantics rather than simple line-by-line comparisons and also enables a more effective application of modifications.

Keywords: *Continuous integration, collaborative coding, conflict detection, transformations*

Contents

List of Figures

List of Tables

List of Acronyms

AST: Abstract Syntax Tree

VCS: Version Control System

UUID: Universally Unique Identifier

CI: Continuous Integration

MVC: Model View Controller

OT: Operational Transformation

CRDT: Conflict-Free Replicated Data Type

IDE: Integrated Development Environment

JSON: JavaScript Object Notation

CLI: Command Line Interface

RPC: Remote Procedure Call

SOA: Service Oriented Architecture

GUI: Graphical User Interface

CHAPTER 1

Introduction

1.1. Context and Motivation

In the contemporary era of software development, software products are developed and maintained through the collaboration of a team of developers, typically affiliated with a company or an open-source community. The facilitation and simplification of this collaboration is made possible by the existence of Version Control System (VCS). VCS, such as Git [**9**], permit teams to oversee the product's development by enabling developers to implement modifications independently and subsequently integrating them by merging them with the primary development branch on a periodic basis.

Although the majority of commits proceed without incident, the simultaneous occurrence of parallel changes can result in overlapping scenarios, potentially leading to merge conflicts. Caius et al. [**2**] observed that in 143 open-source projects, 20% of the merges resulted in conflicts. Similarly, Kasi et al. [**15**] and Brun et al. [**3**] obtained comparable results, with approximately 19% of the merges resulting in conflicts.

While automated merging tools can resolve certain issues, manual intervention is necessary when changes overlap. The resolution process can be challenging, resulting in potential delays as developers must be taken outside of their development process to address and resolve the conflicts, which may disrupt the workflow, impact the efficiency of the team, and affect the timelines for a project.

In order to circumvent the potential for unnoticed conflicts to accumulate, it has been proposed that Continuous Integration (CI) [**6**] be introduced into teams practices [**13**]. The practice of Continuous Integration encourages teams to merge and check-in their work on a regular basis, thereby providing a more robust mechanism for the management of merge conflicts.

The implementation of CI serves to prevent teams from accumulating changes or commits within a specific version control system. The practice of saving a batch of changes and then committing them to the main version is discouraged, as it significantly increases the likelihood of merging issues, especially on a large scale. However, developers frequently avoid the merging process, prompting teams to avoid parallel work [**12**] and causing developers to hasten their task in order to avoid assuming responsibility for the merge [**4**].

1.2. Goals

The principal aim of this dissertation is to devise and construct a protocol for propagation of code changes that is capable of facilitating the early detection of conflicts in collaborative coding environments, in accordance with the principles of continuous integration. The

objective is to develop a structured approach that enables the prevention of conflicts and the seamless integration of change propagation into the collaborative coding experience.

We intend to develop a collaborative structured code editor that incorporates continuous integration functionality as a proof-of-concept for the proposed architecture, Javardair. The objective is to facilitate collaborative software development with seamless integration, while mitigating the challenges associated with merging changes from the VCS.

RQ1a. What are the challenges associated with maintaining consistent and conflict-free code integration in collaborative coding environments?

RQ1b. How can these challenges be effectively addressed?

RQ2. How can transformation-based conflict detection improve collaborative coding in collaborative coding environments compared to traditional Version Control Systems?

1.3. Research Method

The methodology used in this dissertation was designed to identify and analyze relevant literature pertaining to the research themes, namely, the use of continuous integration in a collaborative code editor. A comprehensive search for articles was conducted on platforms such as IEEE Xplore and Google Scholar.

Although no systematic literature review (SLR) was performed, the research was guided by specific criteria to ensure that the quantity and quality of the collected information were maintained. The literature review was divided into three distinct sections. The following sections were included in the literature review: Collaborative Editing, Collaborative Code Editors and Distributed Model View Controller architecture.

The methodology involved the following steps for each section: The initial phase of the literature review began with a careful review of abstracts, serving as a preliminary filter. Subsequently, the introductions and conclusions of the selected literature were examined to ensure a better understanding. Publications that met both the relevance and depth criteria proceeded to the next phase. The selected publications were then further analysed.

1.4. Document Structure

Subsequent to the present chapter, Chapter 2 presents a review of the state-of-the-art, exploring the theoretical foundation, principal tools and methodologies related to collaborative code editing and continuous integration. Chapter 3 introduces a communication protocol as a proposed solution to the issues presented. Chapter 4 provides a detailed overview of the implementation of the prototype proof of concept developed to demonstrate the proposed approach in practice. Finally, Chapter 6 presents the conclusions, limitations, and potential directions for future work.

CHAPTER 2

Literature Review

2.1. Theorical Background

2.1.1. Merging Challenges

In an empirical study conducted by Shane McKee et al. [**20**], a group of developers were interviewed with the objective of gaining deeper insight into the implications of merging conflicts in the software development environment. The investigation sought to ascertain how software practitioners approach merge conflicts, the impact of unmet requirements on conflict resolution complexity, and the effectiveness of tools in meeting practitioners' needs concerning merging conflicts.

The study demonstrated that the majority of practitioners primarily assess the complexity of conflicting lines of code, relying on their expertise within the specific conflict node to estimate the difficulty of the conflict. It was found that many developers do not employ tools or metrics to calculate complexity; instead, they rely on intuitive estimates drawn from personal experience. Furthermore, there is a prevailing tendency to eschew the utilisation of external tools.

In instances where a merge conflict escalates in complexity, developers frequently choose to avoid resolution and revert changes. This disrupts the development flow and may inadvertently result in the deletion of potentially crucial code. A significant challenge for practitioners in resolving merging issues is the dearth of sufficient information at the conflicting nodes, coupled with the risk of overlooking the impact on other nodes due to isolated code changes.

2.1.2. Continuous Integration

Continuous Integration (CI) is a software development practice that encourages teams of developers to engage in frequent merges and check-ins during the day, with each integration verified by an automated build and automated tests [**23**].

Continuous Integration offers a number of potential benefits, including the early detection of integration issues, which can reduce the likelihood of bugs, and future merge conflicts. Additionally, it provides developers with the ability to receive prompt feedback on the impact of their changes. Furthermore, CI ensures the consistency and reliability of software builds.

In this practice, developers retrieve code from a shared repository, implement modifications in their local copies and then submit these modifications back to the repository.

Automated builds and tests are initiated after each submission to verify the code's functionality. In addition to regular submissions and automated tests, maintaining short build times is also considered a key practice in Continuous Integration [**6**].

2.1.3. Model View Controller Architecture

The Model View Controller (MVC) architectural pattern is a software design paradigm that is predominantly employed for the design and development of user interfaces. MVC divides the application into three distinct types of components, each with a specific function. The fundamental components of an MVC architecture are models, views and controllers [**25**]. The model constitutes the fundamental component of the application. In addition to representing the data and business logic of the application, it also manages the data, logic and the rules. It is independent of the user interface and the user input [**21**]. The view represents the user interface and is responsible for displaying the data of the model. Furthermore, the view receives input from the user and transmits it to the controller. It should be noted that an application may comprise multiple views, including a Graphical User Interface (GUI) view and a Command Line Interface (CLI) view [**21**]. The Controller serves as an intermediary between the model and the view, processing user input received from the view and updating the model in accordance with this input.

2.1.4. Distributed Model View Controller

The employment of the method-based MVC architecture within a distributed system gives rise to the necessity of utilising Remote Procedure Call (RPC), which in turn results in the components being tightly coupled [**19**]. The consequence of one system's components affecting all the other components following a change is the potential for flexibility problems and a reduction in the re-usability of the components.

This issue can be addressed through the implementation of alternative solutions:

- **Asynchronous Communication Patterns**: The adoption of models such as Message-Based or Publish-Subscribe [**14**,**19**] allows components to be independent, thereby facilitating greater flexibility and improved re-usability.
- **Micro-services Architecture**: Addresses the coupling problem by dividing the application into small, independent services. Each micro service operates autonomously, thereby minimizing the impact of changes on other components and improving overall flexibility.
- **Service Oriented Architecture (SOA)**: Ensures that the MVC components within each service remain isolated by breaking down the application into services, each with its dedicated functionality, and facilitating communication through APIs [**28**].

2.2. Collaborative Editing

2.2.1. Operational Transformations

Operational Transformation (OT) represents the most commonly employed methodology for ensuring real-time consistency of shared data in collaborative applications [**16**, **24**].

The operational transformation process is of great consequence in the management of collaborative environments, as it comprises two key components: the integration algorithm and the transformation functions.

The integration algorithm is responsible for the reception, transmission and execution of operations within the collaborative space. Concurrently, the transformation functions are engaged when two operations, defined on the same state are combined. The operations received undergo transformations based on local concurrent operations prior to execution.

David Sun et al. [**24**] propose an extension to the existing OT algorithms to facilitate the concurrent execution of update operations alongside the existing insert and delete operations in collaborative word processors.

Furthermore, the OT is divided into two layers: the high-level transformation control algorithms, and the low-level transformation functions. The authors' approach to extending the OT to support update operations entails maintaining the existing high-level control algorithms while introducing new transformation functions tailored to updates. This strategy serves to reduce the overall complexity of the system and localise the extension, thereby facilitating the integration of update operations into the collaborative framework.

Update operations do not affect the linear address space of the document, in contrast to the other operations. Consequently, OT emerges as the preferred algorithm for collaborative programming tools due to its effectiveness in maintaining consistency in shared data across real-time collaborative applications.

However, the biggest drawback regarding OT is the fact that it requires a centralised authority to mediate the edits, which not only restricts scalability but also prevents peer-to-peer decentralised sharing and constrains the flexibility of branching and merging [**18**].

2.2.2. Conflict-Free Replicated Data Type

Conflict-Free Replicated Data Types (CRDTs) constitute a family of algorithms designed for distributed systems, wherein multiple nodes need to update and replicate data without the necessity for a centralized authority. The use of CRDTs permits the updating of data in disparate nodes without the necessity for coordination or locking mechanisms, enabling each update to happen independently [**18**]. In contrast to OT, CRDTs facilitate concurrent operations in commutative manner, meaning that the order of applying operations is inconsequential with respect to the final state.

The algorithms associated with CRDTs can be classified into two principal categories: operation-based CRDTs and state-based CRDTs. Operation-based CRDTs represent data as a sequence of operations that can be applied to an initial state to achieve the current state. Each node independently processes the operations and merges them with its local

state [5]. In contrast, state-based CRDTs represent data as a shared state that can be modified by different nodes. These nodes periodically exchange their states, and any differences are merged to ensure consistency [5].

2.3. Real Time Collaborative Coding Editors

2.3.1. Visual Studio Code Live Share

Visual Studio Code's "Live Share" is a collaborative development tool that enables developers to work together in real time without being in the same physical location, supporting a diverse range of programming languages. In order for developers to utilise this tool, it is necessary for them to install the relevant extension in Visual Studio Code.

Subsequently, one of the developers can initiate a Live Share session, which generates a link that others can utilise to join the collaborative session. The initiator of the session can then grant read-only or read-write access to the remaining of the participants. Live Share offers a number of other significant features, including integrated terminal sharing, debugging collaboration and also audio call integration, which collectively enhance communication between developers [7].

2.3.2. IntelliJ IDEA Code with Me

IntelliJ IDEA "Code With Me" is a collaborative coding tool that enables real-time collaboration, making it suitable for pair programming, code reviews and troubleshooting. It employs the infrastructure of JetBrains to facilitate the collaboration between developers, eliminating the need for a dedicated server. Additionally, "Code With Me" is designed to operate across different operating systems [1].

2.3.3. Jimbo

Soroush et al. [8] designed Jimbo, a collaborative web Integrated Development Environment (IDE), with the principal objective of improving the pair programming experience in both educational and professional contexts. The use of three communication channels facilitate issue resolution while maintaining a steadfast focus on the code. Furthermore, Jimbo has elevated the visibility of code modifications through the integration of a notification system. This system ensures that developers are promptly informed about relevant developments, such as alterations to the code or comments within a thread.

Additionally, the platform prioritises fostering collaboration between developers and designers through the incorporation of a live preview feature. To guarantee the consistency of shared data during real-time collaboration, Jimbo employs operational transformation algorithms, which play a crucial role in maintaining the integrity of the code when multiple users are concurrently working on the same file.

2.3.4. Collabode

Max Goldman et al. [10, 11] developed a collaborative web-based IDE for Java, Collabode, with the objective of gaining insight into how a programming environment that supports close collaboration can improve the quality of the development and collaboration. Collabode

is compatible with any editor, as it has been integrated with EtherPad, a tool that enables real-time text editing.

The software utilizes the Eclipse platform to oversee the management of projects and to facilitate the provision of standard IDE services, including syntax highlighting, continuous compilation, compilation error and warning notifications, code formatting, refactoring and execution.

To address the challenges associated with collaborative editing semantics, each developer is allocated an independent and persistent working copy of the program, while Collabode maintains a disk version and a union version. The union version is the one accessible to users, integrating the edits from all developers. The edits are only shared with the disk version once the code is error free, ensuring that the disk version is always free of compilation errors, and thus enabling programmers to execute the program with minimal disruption.

2.3.5. CodeR

In their studdy, Aditya Kurniawan et al. [**17**] introduce CodeR, a collaborative coding web application that supports C, C++ and Java programming languages. The objective of this study is twofold: firstly, to address the issue of synchronization when multiple developers are working on the same file and secondly, to improve the pair programming experience.

CodeR enables users to engage in real-time collaboration, including the execution and display of results, via a terminal interface. CodeR incorporates Facebook tools for authentication and collaboration, facilitating immediate communication and collaboration among the users. The platform enables users to extend invitations to colleagues or other contacts to collaborate on a given project, with communication features such as real-time chat. Furthermore, the platform offers a user-friendly interface for managing files and folders within the workspace, enabling users to perform fundamental operations such as creation, deletion, renaming, moving, opening, downloading and uploading files.

CodeR employs an Operational Transformation algorithm to maintain data consistency during the real time collaboration, guaranteeing the integrity of the code when multiple users are working on a single file.

2.3.6. Comparison

This subsection presents a comparative analysis of the collaborative coding tools discussed in this chapter, with a particular focus on their respective features, consistency management methods and suitability for collaborative coding scenarios (Table 2.1). The objective is to provide an objective assessment of the strengths and limitations of each tool.

TABLE 2.1. Comparison between existing collaborative tools and Javardair

Collaborative Tool	**Platform**	**Programming Language Support**	**Key Features**	**Collaboration Model**	**Consistency Management**	**Advantages**	**Drawbacks**
VS Code Live Share [**7**]	Desktop (VS Code Extension)	Multiple Languages	Debugging collaboration; Pair programming; Code reviews; Terminal sharing; Audio call integration	Link-based access with read-only or read-write permissions	Textual based	Easy integration into existing workflows; Supports multiple languages	Lack of control over code consistency
IntelliJ IDEA Code With Me [**1**]	Desktop (IntelliJ IDEA Plug-in)	Multiple Languages	Debugging collaboration; Pair programming; Code reviews; Cross-OS support	Link-based access with read-only or read-write permissions	Textual based	Strong cross-OS support; Integrated with IntelliJ ecosystem	Lack of control over code consistency
Jimbo [**8**]	Web-based	Web development	Live preview; Audio/text chat; Notifications	Dedicated server	Operational Transformation algorithm for real-time consistency	Focused on improving pair programming; Live preview improves designer-developer collaboration	Limited to web technologies and may not support general-purpose programming.
Collabode [**10**, **11**]	Web-based	Java	Independent error-free working copies; Continuous compilation; Syntax highlighting	Dedicated server	EtherPad integration for real-time editing and error-aware integration algorithm prevents broken builds	Prevents propagation of compilation errors; enables seamless editing without immediate conflict.	Lack of information regarding errors; Confusing interface
Coder [**17**]	Web-based	C, C++, Java	Real-time chat; Terminal execution; File/folder management	Dedicated server	Operational Transformation algorithm for real-time consistency	Real-time execution and immediate communication enhance debugging and efficiency	Language-specific; Single file editing
Javardair	Desktop	Java	Conflict-aware editing; Semantic-level Conflict detection	Dedicated server	Operational Transformation with AST comparison	More precise conflict detection and merging; Immediate conflict awareness; Manual propagation of changes	Scalability issues with large projects or teams; Language-specific

CHAPTER 3

Approach

3.1. Motivating Example

This section presents two illustrative examples to elucidate and substantiate the rationale behind this dissertation approach to conflict detection and change propagation in collaborative coding environments.

These examples are meant to illustrate the practical advantages of the approach methodology over traditional VCS and live-edit collaboration tools in addressing the prevalent challenges encountered in collaborative software development.

Both examples utilize the file `Calculator.java` as a base version (Figure 3.1).

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    int subtract(int a, int b) {
        return a - b;
    }
}
```

FIGURE 3.1. Base version of Calculator.java.

3.1.1. Example 1

Figure 3.2 and Figure 3.3 illustrate a scenario where two developers, Developer A and Developer B , work independently on local versions of the same file (Figure 3.1). Developer A adds a new `main` method that references the existing `add` method, while Developer B renames the method `add` to `sum`. Figure 3.4 illustrates the optimal outcome for this particular type of scenario and reflects the capabilities of our approach.

In a traditional VCS systems such as Git, this situation would typically result in a merge conflict (Figure 3.5). Both Developer A and Developer B have modified the same file, both interacting with the method `add`, but in a incompatible way. A VCS system would try to merge these changes by looking for a common ancestor (the base file present in the Server) and then apply both modifications on top of that.

Developer A's main method contains a reference to method `add`, a method that has been renamed by Developer B, effectively removing the method `add` from the code in Developer B's version. This leads to ambiguity in the merge process: the system cannot automatically determine whether the reference in `main` should point to the newly renamed

```
class Calculator {
  int add(int a, int b) {
    return a + b;
  }

  int subtract(int a, int b) {
    return a - b;
  }

  void main(String args[]) {
    int result = add(1,2);
  }
}
```

FIGURE 3.2. Developer A version.

```
class Calculator {
  int sum(int a, int b) {
    return a + b;
  }

  int subtract(int a, int b) {
    return a - b;
  }
}
```

FIGURE 3.3. Developer B version.

```
class Calculator {
  int sum(int a, int b) {
    return a + b;
  }

  int subtract(int a, int b) {
    return a - b;
  }

  void main(String args[]) {
    int result = sum(1,2);
  }
}
```

FIGURE 3.4. Ideal merged version.

`sum` method or remain as `add`. As a result, Git treats this as a conflict, requiring manual intervention to resolve the ambiguity.

In live-editing systems, such as those provided by integrated development environments like Visual Studio Code or IntelliJ IDEA, a similar issue arises. Live-editing tools propagate

```
<<<<<<< Developer A's change
    int add(int a, int b) {
        return a + b;
    }
    ...
    void main(String args[]) {
        int result = add(1,2);
    }
=======
    int sum(int a, int b) {
        return a + b;
    }
>>>>>>> Developer B's change
```

FIGURE 3.5. Conflicting changes after Developer A and Developer B commit their changes.

changes made by collaborators in real-time without the use of conflict detection or resolution mechanisms.

In this scenario, as soon as Developer B renames the method `add` to `sum`, this change would be immediately reflected in the shared document. Concurrently, Developer A's changes, which reference a method the now-renamed `add` method in the new `main` method, would cause a runtime or compile time error due to the disappearance of the `add` method.

For this type of collaboration to work, both developers must be constantly aware of each other's real-time changes, and coordinate this type of modifications to avoid introducing inconsistencies or errors in the code.

3.1.2. Example 2

Figure 3.6 and Figure 3.7 illustrate a different scenario in which Developer A and Developer B are concurrently modifying the body of the same method, `add`.

```
class Calculator {
  int add(int a, int b) {
    return a + b + 1;
  }

  int subtract(int a, int b) {
    return a - b;
  }
}
```

FIGURE 3.6. Developer A version.

In a traditional VCS, attempting to push these changes would lead to a merge conflict (Figure 3.8). This is because the modifications are applied to the body of the same method, resulting in overlapping edits that Git cannot automatically reconcile. Git would

```
class Calculator {
  int add(int a, int b) {
    return 10 + b;
  }

  int subtract(int a, int b) {
    return a - b;
  }
}
```

FIGURE 3.7. Developer B version.

prompt the collaborators with a merge conflict notification, requiring manual intervention to determine which version of the body should be kept or how the changes should be integrated.

```
<<<<<<< Developer A's change
    int add(int a, int b) {
        return a + b + 1;
    }
=======
    int add(int a, int b) {
        return 10 + b;
    }
>>>>>>> Developer B's change
```

FIGURE 3.8. Conflicting changes after Developer A and Developer B commit their changes.

Our proposed approach was designed to not only streamline the merging process but also to proactively handle potential conflicts. It would detect that the changes made by Developer A and Developer B to the body of the `add` method are incompatible, as both involve distinct modifications to the same section of the method. The system would flag this situation as a conflict as it requires manual resolution.

Rather than allowing the conflicting changes to proceed unchecked or rely solely on post-commit conflict resolution, it ensures that these types of conflicts are recognized at an early stage. It would notify both collaborators of the conflict before either of them attempt to push their changes to the shared code base.

3.2. Protocol Architecture

The objective of this protocol is to facilitate collaborative software development, enabling multiple collaborators to work concurrently in isolated environments while maintaining a shared codebase.

The principal function of the system is to monitor individual modifications made by each collaborator and provide awareness of potential conflicts. This allows the architecture

of the protocol to be divided into four different scenarios, which can be used to manage and synchronise the existing work.

Connect: Synchronizes collaborator workspace with the existing workspace in the Server.

Update: Automatically alerts the server of new changes and verify for potential conflicts.

Push: Submits the collaborator's local changes to the server and propagates them to other connected collaborators, provided they are conflict-free with other collaborators at that time.

Force Push: Similar to Push, but allows collaborators to submit their changes even in the presence of conflicts, overriding any conflicting modifications. Ideally used to unblock development when a conflict prevents progress.

To better understand the protocol, consider a scenario involving two collaborators, Developer A and Developer B, both working on a shared Java file, `Calculator.java`, represented by Figure 3.9.

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }
}
```

FIGURE 3.9. Base version.

3.2.1. Connecting to the server

When a developer attempts to establish a connection with the server, it initiates a synchronization process between their local workspace and the server's latest version of the codebase. The connection is initiated with the transmission of a `Handshake` message, which contains the universally unique identifier (UUID) and a name associated with the Developer. This identifier is used to uniquely identify the client within the system.

Subsequently, a `Fetch Request` is transmitted, prompting the server to return the latest version of the shared codebase. In response, the server returns a `Fetch Response`, containing an encoded version of all files available in its codebase. Figure 3.10 illustrates the scenario in which Developer A attempts to establish a connection with the server.

3.2.2. Making a local change

In this scenario, Developer A renames the `add` method to `sum` (Figure 3.11), while Developer B introduces a new method `subtract` (Figure 3.12).

Upon these modifications, an Update message is sent to the server, containing information about the respective transformations performed by each one of the collaborators.

Assuming the system has already processed Developer A's changes (the renaming of `add` to `sum`) without detecting any conflicts (Figure 3.13, Developer B's Update message, including the addition of the method `subtract`, would be compared against these prior changes. In the light of the fact that no conflicts are evident in the transformation

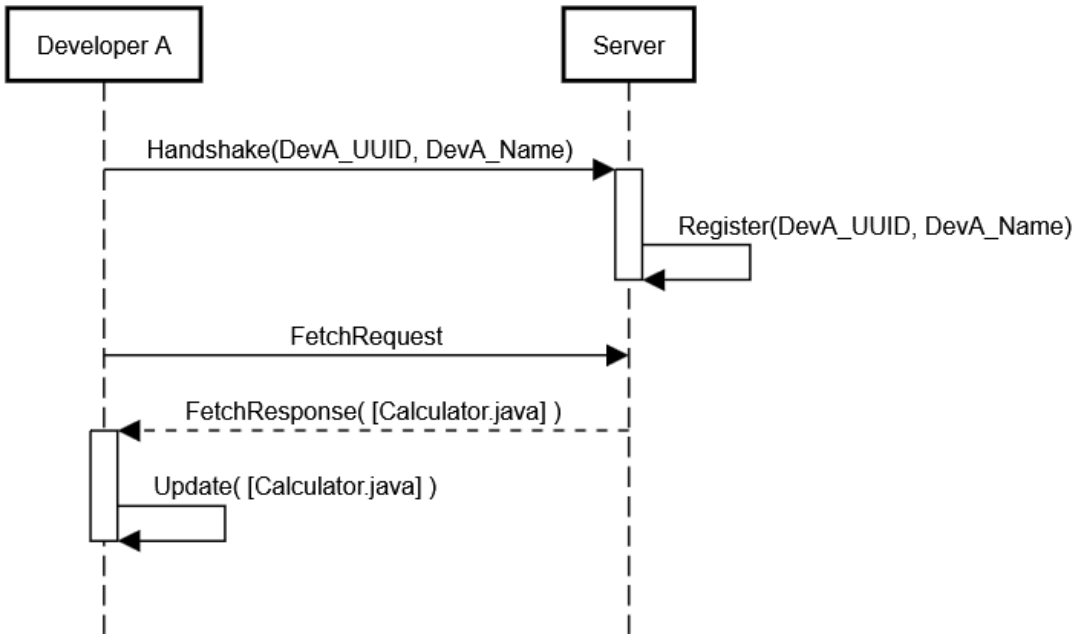

FIGURE 3.10. Developer A establishing connection with the server.

```
class Calculator {
    int sum(int a, int b) {
        return a + b;
    }
}
```

FIGURE 3.11. Developer A version.

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    int subtract(int a, int b) {
        return a - b;
    }
}
```

FIGURE 3.12. Developer B version.

in question, the server concludes that both actions are non-conflicting. As a result of such determination, a `Notify Conflicts` message is then issued by the server to both Developer A and Developer B, as represented in Figure 3.14. The message contains an empty list of conflicts.

However, if Developer B had instead attempted to rename the `add` method to `addition` (Figure 3.15), a conflict would arise since both developers are renaming the same method, but to different names.

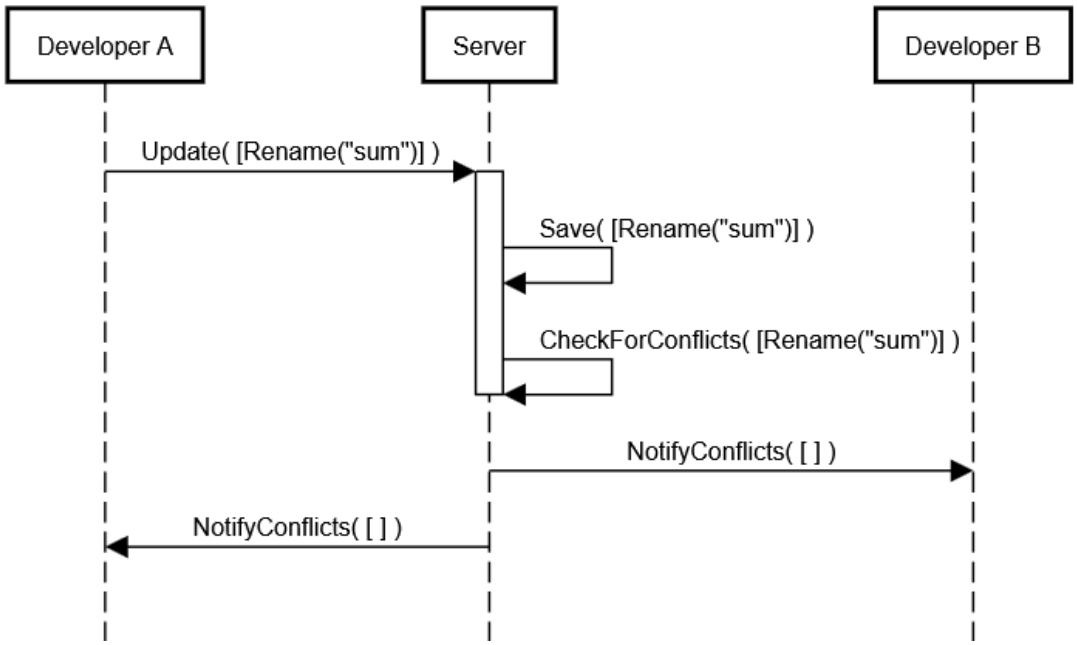

FIGURE 3.13. Update action performed after local changes are made by Developer A.

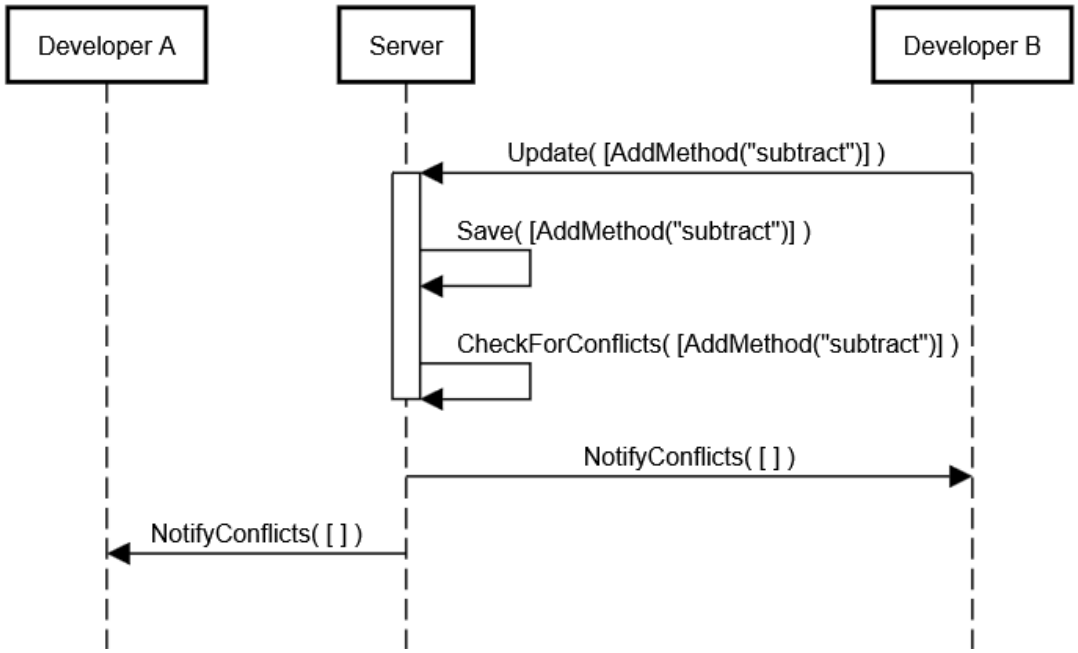

FIGURE 3.14. Update action performed after local changes are made by Developer B.

In this instance, the system would identify a conflict and generate a `Notify Conflicts` message, directed to both Developer A and Developer B, indicating that a conflict has occurred. This message would provide detailed information of the specific conflict regarding the original method `add`, along with information identifying the conflicting collaborator (Figure 3.16).

```
class Calculator {
    int addition(int a, int b) {
        return a + b;
    }
}
```

FIGURE 3.15. Developer B version.

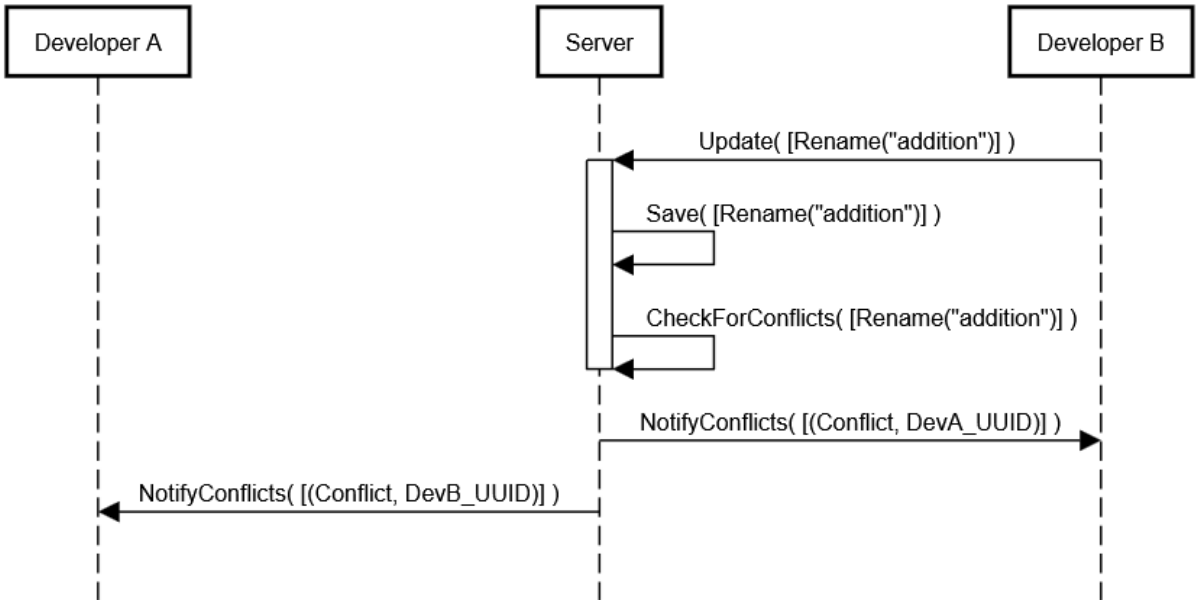

FIGURE 3.16. Update action performed after local changes are made by Developer B.

3.2.3. Pushing changes

When Developer A decides to propagate the changes, a `Push` message is transmitted to the server. This message contains a list of the transformations performed - in this case, renaming `add` to `sum`. Prior to applying the changes globally, the server performs a further check for conflicts. Upon finding none, the server applies the transformations to its own directory and sends a `Propagate` message to all connected collaborators (Figure 3.17), ensuring their workspaces are updated accordingly.

Following the receipt of the `Propagate` message, Developer B's workspace is updated to reflect the following structure, represented by Figure 3.18.

Once Developer A's modifications have been implemented, the server's codebase will assume the structure illustrated on Figure 3.19:

It should be noted that the renaming transformation did not override other non-conflicting changes performed by Developer B, such as addition of the method `subtract`.

In the event that Developer B intends to submit the modifications to the code, the procedure would be analogous to that of Developer A, given that no conflict has been identified (Figure 3.20). Consequently, all versions of the `Calculator.java` file will look like Figure 3.21.

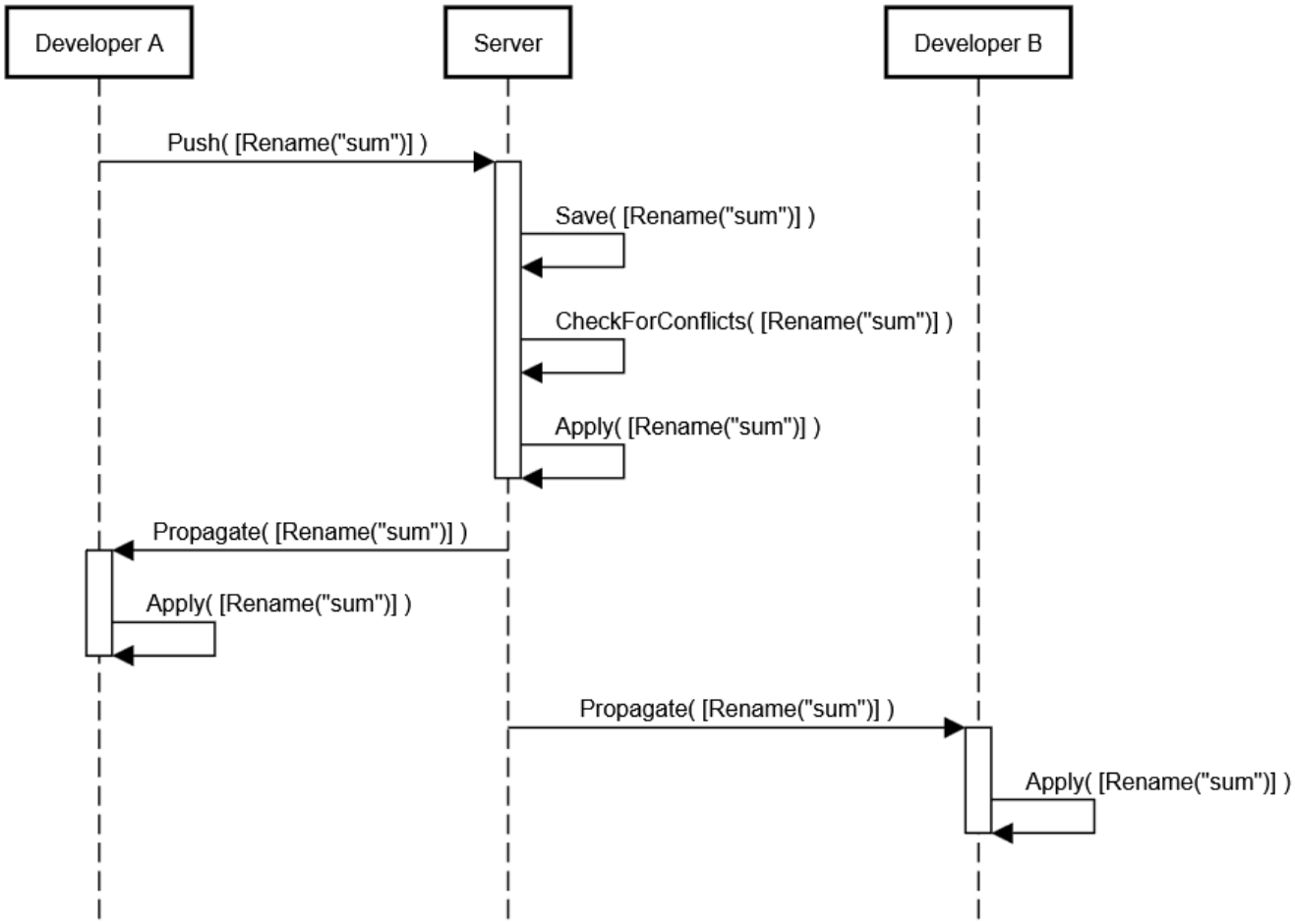

FIGURE 3.17. Push action performed by Developer A.

```
class Calculator {
    int sum(int a, int b) {
        return a + b;
    }

    int subtract(int a, int b) {
        return a - b;
    }
}
```

FIGURE 3.18. Developer B version after receiving the Propagate message.

However, if for some reason a conflict would be detected by the server, the system will instead issue a `Notify Conflict` message, containing the relevant information, rather than a `Propagate` message. Furthermore, the server will not be updated with the modifications that are being shared.

3.2.4. Force-pushing changes

Consider now a scenario in which Developer B also renames the `add` method, this time to `addition` (Figure 3.23). This creates a conflict with the previous renaming of the method

```
class Calculator {
    int sum(int a, int b) {
        return a + b;
    }
}
```

FIGURE 3.19. Server version after accepting Developer A changes.

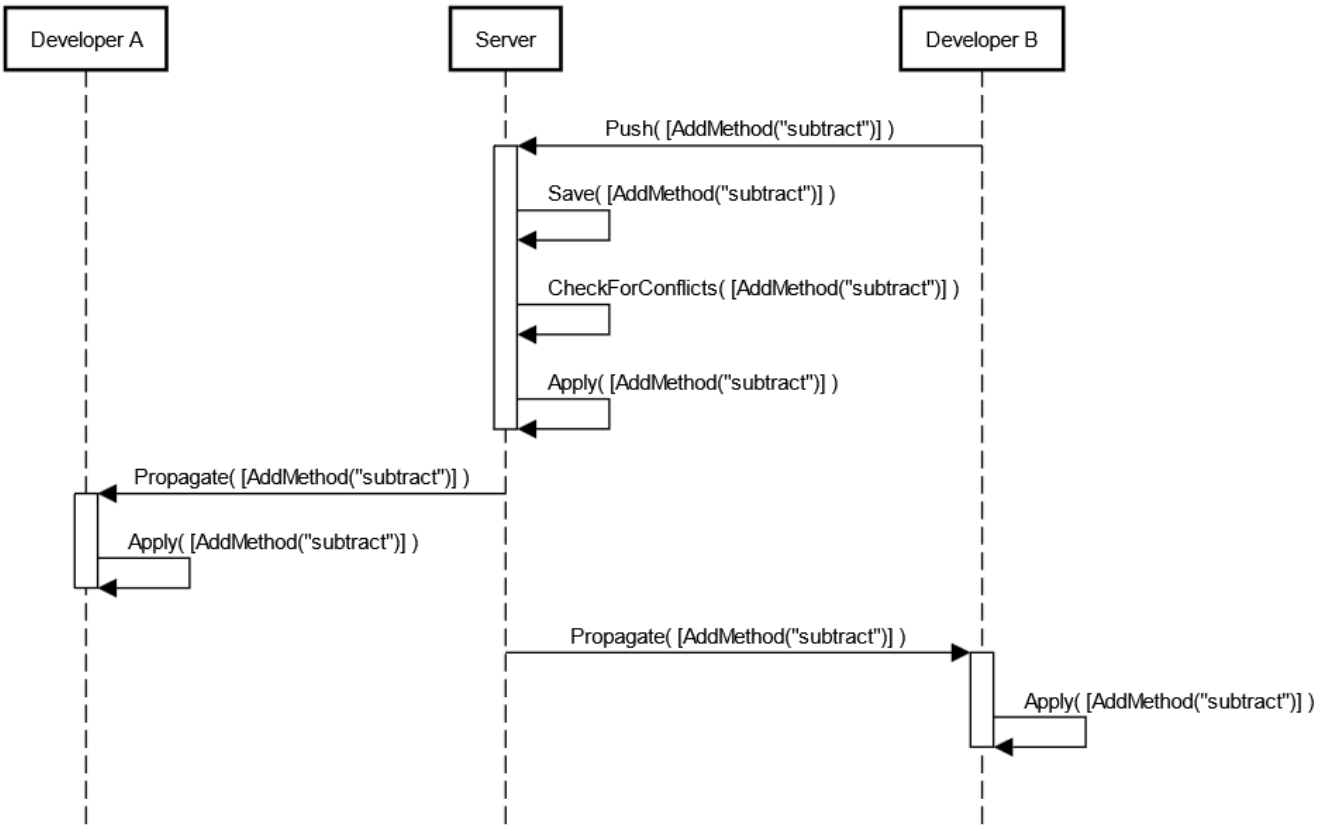

FIGURE 3.20. Push action performed by Developer B.

```
class Calculator {
    int sum(int a, int b) {
        return a + b;
    }

    int subtract(int a, int b) {
        return a - b;
    }
}
```

FIGURE 3.21. Every instance of Calculator.java.

by Developer A to `sum` (Figure 3.22). Both developers are then notified of the conflict through the `Notify Conflicts` message just like the scenario in Figure 3.16.

```
class Calculator {
    int sum(int a, int b) {
        return a + b;
    }
}
```

FIGURE 3.22. Developer A version.

```
class Calculator {
    int addition(int a, int b) {
        return a + b;
    }
}
```

FIGURE 3.23. Developer B version.

However, should Developer B be insistent upon these changes, the `Force Push` action may be used. This action transmits a `Force Push` message to the server, forcing the alterations performed by Developer B to be applied to the server codebase and propagating these same changes via a `Propagate` message to the connected Developers (Figure 3.24).

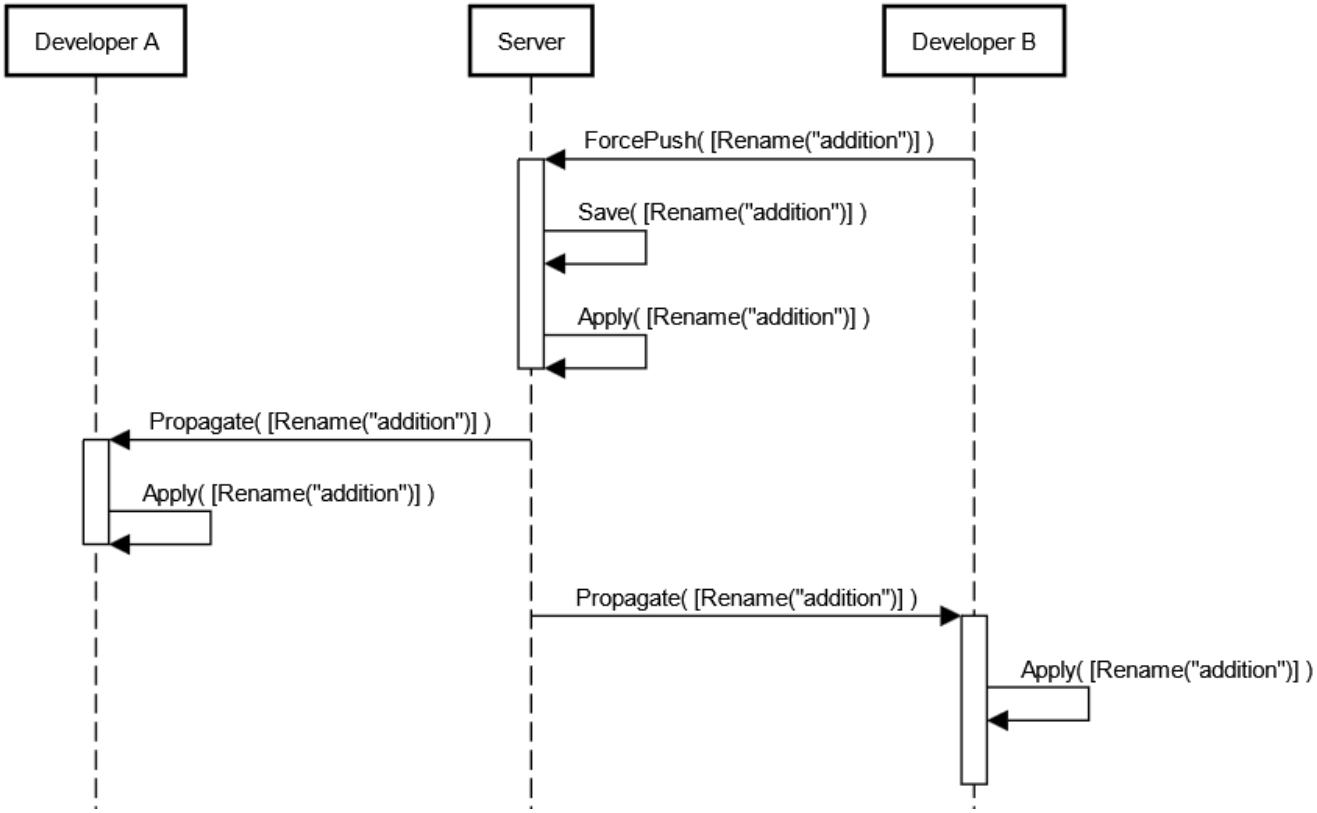

FIGURE 3.24. Force Push action performed by Developer B.

CHAPTER 4

Implementation

As a proof-of-concept for the proposed approach, we have developed Javardair, a collaborative coding environment for Java projects. Javardair uses two key technologies to facilitate the extraction of modifications, conflict detection, and structured editing of Java code. The first of these is Jaid [**26**], a transformation-based conflict detection tool, and the second is Javardise, a structured code editor.

4.1. Enabling Technologies

4.1.1. JavaParser

JavaParser [**27**] is an open source library that allows the transformation of Java source code into an Abstract Syntax Tree (AST). An AST is a hierarchical tree-like data structure that represents the syntactic structure of the source code. Each node in the AST represents an element of the source code, such as classes, methods, or statements. Figure 4.1 illustrates an example of a Java class represented in the format of an AST.

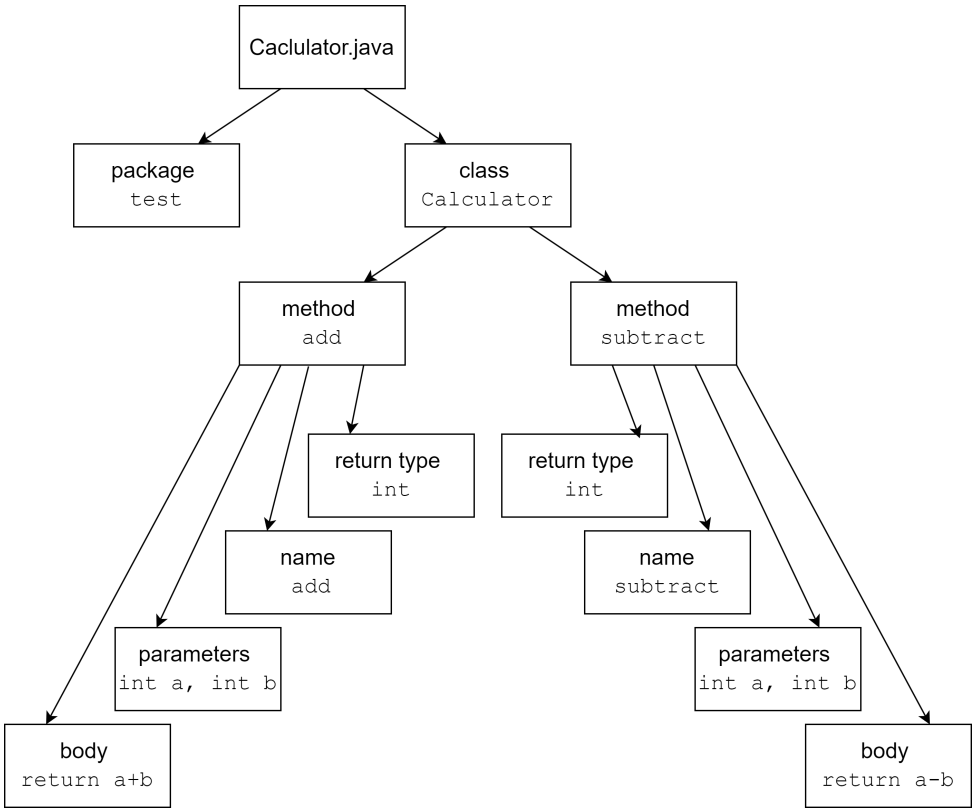

FIGURE 4.1. AST of a Java file (code of Figure 4.2).

4.1.2. Jaid

Jaid is a prototype tool designed to assist and improve the merging process. It uses an approach based on transformations and member identity, rather than the traditional text-based conflict detection used in VCS [**26**]. This approach enables Jaid to address instances where traditional VCS either identifies conflicts that are not present or are not problematic, or fails to identify conflicts that do exist, due to its textual and line-based approach.

The use of AST as a means of representing code enables Jaid to perform more granular and language-aware conflict detection.

Jaid operates on the basis of UUIDs attached to the structural elements of the code (such as methods and fields). This ensures that each element retains a persistent identity across multiple versions of the code. The use of unique identifiers also facilitates the tracking of changes to individual elements, which is a key mechanism for this project. Figure 4.2 illustrates a Java class with UUID comments attached to each member.

```
//9e30e98a-36db-47f4-836c-16c390a1d2d7
package test;

//13c9f311-0d07-46aa-8591-ef22c6ab8e49
class Calculator {

    //73bb1c00-f3ab-41a7-9c56-1e1ba192f751
    int add(int a, int b) {
        return a + b;
    }

    //52f17915-21ad-413a-824d-0c39584037f0
    int subtract(int a, int b) {
        return a - b;
    }
}
```

FIGURE 4.2. Java source code with UUID comments attached.

4.1.3. Javardise

Javardise is a structured editor that supports a subset of Java's syntax [**22**], created with the aim of improving the pedagogical experience of programming. It is based on the MVC architecture and makes use of JavaParser AST parsing mechanism, which means that when using this editor, changes are applied over the AST version of the source code.

Javardise provides an important feature for this project: the ability to extend and customise its functionality through the development and use of plug-ins. Plug-ins allow specific actions and behaviours to be added to the editor interface. This way, it was created four different plug-ins for Javardise to streamline main actions: connecting to the server, tracking the changes, pushing changes and forcing pushing changes.

4.2. Implementation

Figure 4.3 represents a diagram of components that highlights the relationship between the components mentioned above and that will be used in the implementation of the prototype.

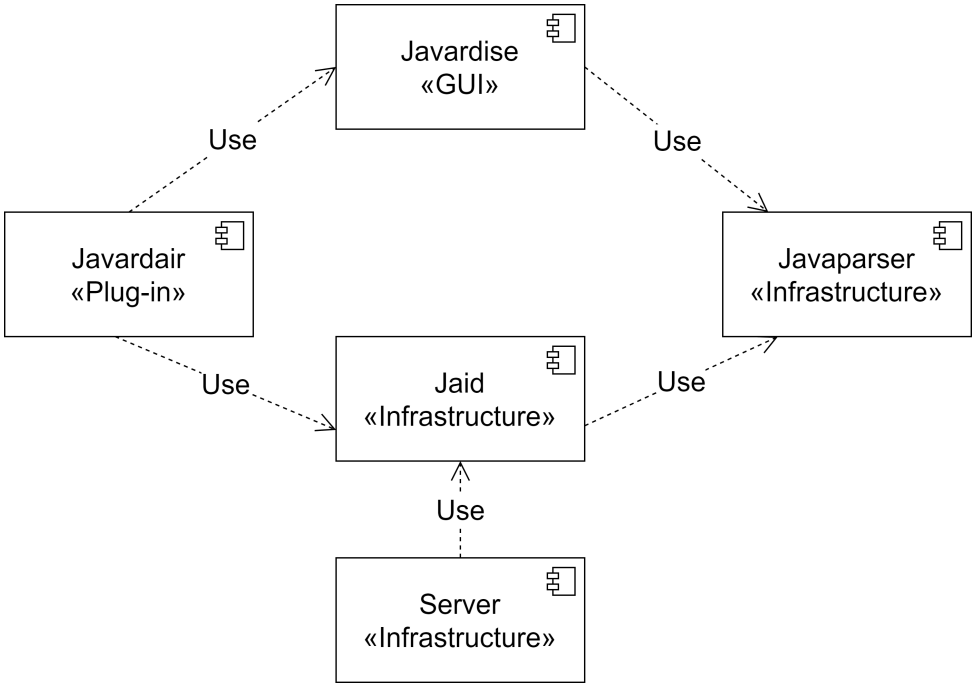

FIGURE 4.3. Javardair component diagram.

4.2.1. User Interface

Figure 4.4 illustrates the Javardair user interface. The interface's toolbar is comprised of four fundamental buttons, which provide access to the Javardair collaborative actions. These include buttons to connect to the server, enable tracking of modifications, push changes, and force push changes, as previously described.

Furthermore, the prototype employs the use of two pop-up windows. One displays the current list of modifications made to the code by the user (Figure 4.5), thus enabling the user to ascertain which modifications diverge from the server and also to identify the changes that will be propagated. The second pop-up window provides a detailed view of any conflicting changes with other connected users (Figure 4.6).

This way, the prototype serves as a comprehensive functional implementation of the proposed approach to collaborative coding with conflict detection, supporting the core features discussed in this chapter.

4.2.2. Messaging Format

Communication between the Javardair client side and server side is facilitated by a socket-based communication system. For the purposes of clarity and organisation, the messages exchanged between the client and the server are divided into two categories: Client Messages and Server Messages. Client Messages originate from the client side of

FIGURE 4.4. Javardair interface.

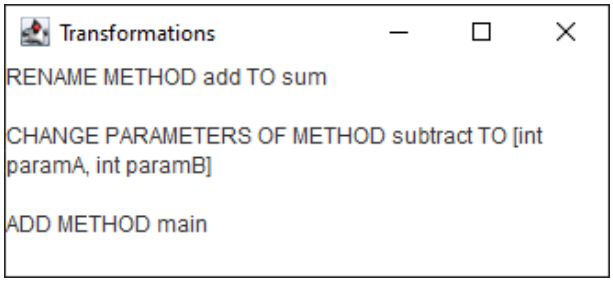

FIGURE 4.5. Transformations window.

Javardair and are employed for the purpose of interaction and updating the server. Server Messages are those transmitted by the server back to the client in response to a Client Message.

Both Client and Server Messages adhere to a consistent structural pattern, as follows:

- **Operation**: Indicates the type of operation being performed, such as updating the server, pushing the changes or requesting synchronisation.
- **Content**: Represents information relevant to the operation, such as lists of transformations, conflicts, or file metadata.
- **Identifier** (only present in Server Messages): Identifies the collaborator who sent the Client Message that lead to the Server Message.

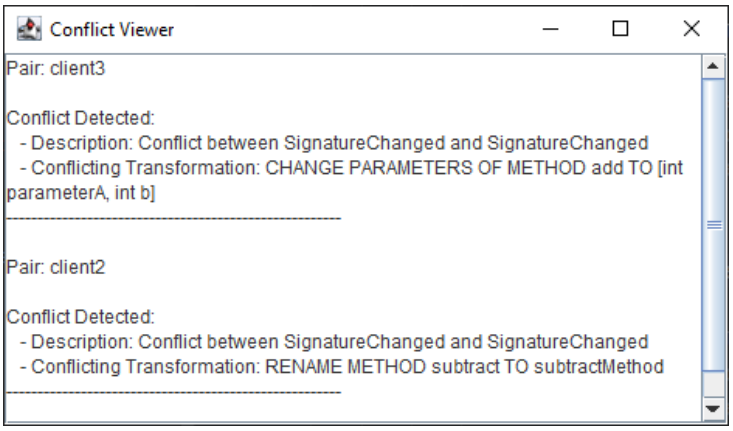

FIGURE 4.6. Conflicts window.

Table 4.1 presents a comprehensive table detailing the different types of messages based on the different operations, outlining the operation, content and structure for each one.

Operation	**Type**	**Content**	**Description**
Handshake	Client	Client Information	Establishes a connection with the server.
Fetch Request	Client	None	Requests the server to update the files to the most current state.
Update	Client	List of Transformations	Notifies the server of recent changes done to the code.
Push	Client	List of Transformations	Requests the server to apply the changes done and propagate it, if no conflicts exist.
Force Push	Client	List of Transformations	Similar to Push, but ignores the existence of conflicts.
Fetch Response	Server	Encoded List of Files Client ID	Response to Fetch Request messages, providing the list of files.
Propagate	Server	List of Transformations Client ID	Sends the transformations pushed to connected collaborators.
Notify Conflicts	Server	Map of Conflicts Client ID	Provides information about existing conflicts, to the collaborators involved.

TABLE 4.1. Existing message types.

These messages are all serialised in JavaScript Object Notation (JSON) format, allowing both sides to easily parse and interpret the data received.

4.2.3. Dealing with multiple clients

The server-side architecture employs a multi-threaded approach to facilitate the management of concurrent connections from multiple users. This model assigns a dedicated

thread to each connected peer. The function of each thread is to handle all communication between the server and its associated worker in an autonomous manner, thereby ensuring that interactions with one worker do not impinge upon those of other clients. Upon establishing a connection, the server assigns each client's unique identifier and name, provided in the handshake message, to the corresponding thread managing that client. This facilitates the tracking and management of each client on an individual basis.

When the server receives an Update or a Push message from a collaborator, it processes the attached list of transformations and stores them in a shared data structure. This structure is designed to map the latest set of transformations provided by the collaborator to the corresponding thread instance, thus maintaining an independent record of the changes submitted by each collaborator.

However, the potential for race conditions or data inconsistencies within the server may arise when handling simultaneous changes from multiple contributors. To address this issue, a locking mechanism is employed to guarantee the integrity of the shared data structure.

The synchronisation mechanism guarantees that when a thread attempts to access the data structure for the purpose of storing newly received data or checking for the existence of conflicts, it first verifies whether the structure is currently in use by another thread. In the event that it is, the thread is blocked until the structure becomes available. Once access is granted, the thread locks the structure, updates it in accordance with the relevant criteria and then releases the lock.

4.2.4. Encoding files

As previously discussed, the server plays a pivotal role in providing collaborators with the most recent version of the project files upon receipt a Fetch Request message. In order to achieve this, the server encodes the content of each file in Base64 format and encapsulates it into a custom object. This object not only stores the encoded data but also includes metadata such as the file name, thus enabling the server to transmit a structured list of files.

On the client side of Javardair, the system oversees the reception and integration of the files sent by the server.

Upon receiving the encoded files, the system decodes the Base64 content back to its original form and writes it to the corresponding workspace directory. In the event that a file does not already exist, a new file is created in both the root and local directories. Conversely, in cases where the file already exists, the system overwrites the file in the root

4.2.5. Transformations

Transformations represent a fundamental concept within the Javardair framework. Transformations encapsulate the atomic operations that modify the source code in a structured and traceable manner. Each transformation represents a structured alteration to the source code, reflecting any modification made by each collaborator.

A transformation is not merely a discrepancy between two iterations of a code file. Rather, it is a formal, structured operation that can be applied to an AST. This methodology permits alterations to be depicted in a more comprehensive manner than is feasible with conventional VCS. Transformations are classified into four categories:

- **Addition**: Introduction of new elements, such as a method, class, or field, into the source code.
- **Removal**: Deletion of an existing element from the source code.
- **Modification**: Changing the properties or content of an existing element (e.g., altering a method's signature or modifying a field's type).
- **Move**: Relocation of an element within the source code (e.g., changing the position of a method within the same class).

Each collaborator's workspace in Javardair is divided into two main directories: the root directory, which mirrors the most recent version of the codebase stored on the server, and the local directory, where collaborators independently modify their own version of the source code. This structure allows collaborators to work autonomously without immediately affecting the global state of the project, while ensuring that all changes are tracked by the system.

In order to extract the transformations from the code, the AST of the files in the root directory are compared with the AST of the files in the local directory, which reflects the changes made by the collaborators. This comparison enables Javardair to detect structural changes at a granular level, identifying additions, deletions or modifications that have been introduced into the code. The system initially identifies file-level alterations, including the addition or removal of an entire file, through a process of comparison involving file identifiers. It then proceeds to examine specific elements within each matching file. Each item is uniquely identified by a unique identifier (UUID), which enables the system to track changes even in instances where the item's properties, such as its name or content, have undergone modification [**26**]. This comparison yields a set of transformations, each of which describes a specific change.

Transformations in Javardair are implemented using the Command pattern, whereby each transformation is treated as a command that can be applied to the AST. This design pattern facilitates the application and reversal of transformations. Each transformation command contains the necessary information to be applied to the AST.

4.2.6. Serialising Transformations

In order for the transformations to be shared between collaborators and the server, they must be serialised into a format that is easily transmittable and interpretable by all parties. Therefore, a format that ensures reliable serialisation, transmission and deserialisation of the transformations was required. JSON was chosen for this purpose, as it ensures that transformations can be reliably serialised, transmitted and deserialised in a format accessible to any potential client implementation. The conversion to JSON ensures the

preservation of all essential information pertaining to the transformation, including the UUID of the modified element, the type of transformation, the associated content, and other relevant details.

Each transformation is serialised by first creating a JSON object, which serves as a container for key-value pairs. The key is a String that identifies specific attributes, such as the UUID of the element, name, parameters, and so forth. The value is a JSON element that contains the corresponding data. Jaid comprises 35 different transformations types, but for the current implementation Javardair, only 10 transformation types were taken into account (AddCallable, SignatureChanged, ReturnTypeChangedMethod, BodyChanged-Callable, RemoveCallable, AddField, RemoveField, RenameField, TypeChangedField, InitializerChangedField). Each of these transformation types has its own distinct structure. However, they all include a key-value pair for a field that serves as an universal identifier for the type of transformation being represented.

In Figure 4.7 a new method `division` is added - the system would define this transformation as a `AddCallable`.

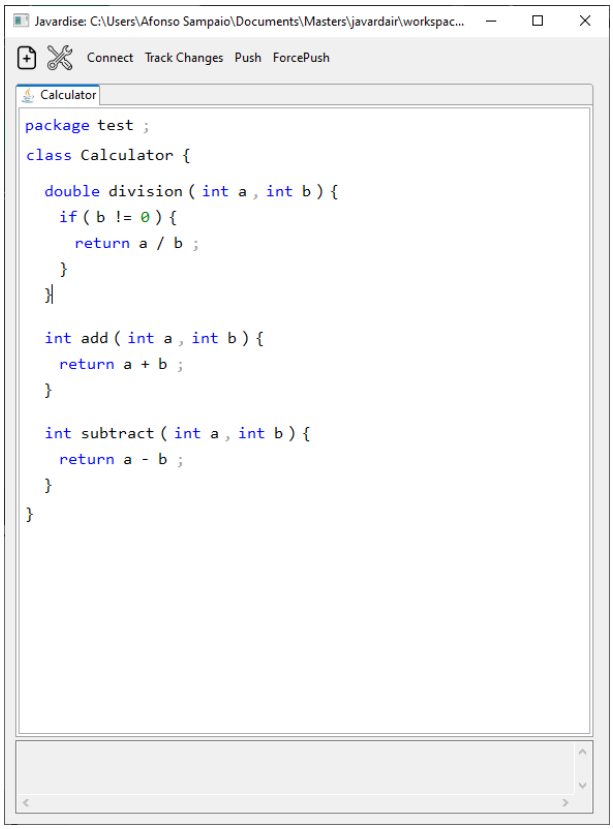

FIGURE 4.7. Adding a new method to a Java file.

When serialising this transformation, the content of the JSON regarding the Add-Callable transformation would look like the following:

- **Code**: "AddCallable"
- **Owner UUID**: "13c9f311-0d07-46aa-8591-ef22c6ab8e49"
- **Constructor**: false

- **Body**: `"//fa661ee2-54c9-40bb-af9c-c8093de1bb3c \r \n double division(int a, int b) { \r \n if (b != 0) { \r \n return a / b; \r \n } \r \n}"`

4.2.7. Applying Transformations

In order to prevent inconsistencies or errors, it is important to correctly apply the transformations, particularly when elements may no longer exist due to specific modifications. Javardair uses the algorithm employed by Jaid to apply transformations to the code, which enforces a specific sequence [**26**], as outlined below:

(1) Apply all file additions.
(2) Filter out all inter-type move transformations and apply only their corresponding removal transformations.
(3) Apply all other removal transformations (including files, methods, fields, etc).
(4) Apply all local move transformations in the order they were extracted.
(5) Filter all inter-type move transformations and apply only their insertion transformations.
(6) Apply all other insertion (adding method, fields, etc).
(7) Apply all remaining transformations in any order.

In scenarios where a transformation introduces new references, such as the one illustrated in Figure 3.2 and Figure 3.3 , where Developer A renames the method add() while Developer B makes a call to the same method using the "old" name, the system updates any outdated references to maintain code consistency.

This is made possible by Jaid's utilisation of a translation mechanism that updates references to renamed or moved elements, employing the use of UUIDs to track these elements [**26**]. During the application of transformations, such as renames or moves, the system records the changes and uses this information to update references in subsequent transformations. In the case of other types of transformations, the system performs a verification process to ascertain whether the referenced element has undergone a renaming or relocation. In the event of such a change, the system translates the reference to the new identifier.

Upon receipt of a Propagate message from the server, the system initiates a sequence of operations prior to attempting to implement the modifications. Algorithm 1 and 2 describe this sequence of operations.

Javardair initiates the process by extracting any existing transformations at the moment the message is received. Subsequently, the list is compared to the transformation list intended for application to ascertain whether there is a match. In the event of match, the received transformation list is applied to both directories, unless the receiving client is identical to the client that sent the original Push message (which initiated the Propagate), in which case it is only applied to the root directory.

Algorithm 1 Function to evaluate transformations by comparing the current set of transformations with the received set of transformations. The parameter *receivedTransformations* is a set of transformations and the parameter *sender* is the UUID of the collaborator responsible for the transformations.

```
function evaluateTransformations(receivedTransformations, sender)
    currentTransformations ← getTransformations()
    applyChanges(receivedTransformations, sender)
    if setsAreNotEqual(currentTransformations, receivedTransformations) then
        conflicts ← getConflicts(receivedTransformations, currentTransformations)
        if conflicts is not empty then
            applyTransformationsTo(projectLocal, currentTransformations)
        end if
    end if
end function
```

Algorithm 2 Function to verify where to apply the transformations. The parameter *receivedTransformations* is a set of transformations and the parameter *sender* is the UUID of the collaborator responsible for the transformations.

```
function applyChanges(receivedTransformations, sender)
    if sender ≠ clientID then
        applyTransformationsTo(projectLocal, receivedTransformations)
    end if
    applyTransformationsTo(projectRoot, receivedTransformations)
end function
```

In the event of inconsistencies between the transformation lists, the received transformations are nevertheless applied to both directories, unless the receiving client is the same as the one that sent the original Push message (that lead to the Propagate), in which case it is only applied to the root directory. Furthermore, any inconsistencies between the two lists will be identified and addressed concurrently.

In the absence of conflicts, no additional measures are undertaken, apart from updating the server (via an Update message) to ensure uniformity. However, in instances where conflicts are identified, the list of transformations extracted at the time of the message, will take precedence over the recent modifications in the local directory.

4.2.8. Conflict detection

In the context of this dissertation, the term "conflict" is defined as a situation that requires the direct involvement of humans in order to achieve resolution. Such conflicts arise when the application of transformations simultaneously results in semantic errors in the source code.

In the event that two contributors attempt to rename the same method to disparate values, an automated system, such as VCS or a Live Editing tool, is not able to decide which transformation should prevail during the merge process.

Jaid's conflict detection algorithm is based on a predefined set of conflict types, which outlines pairs of transformation types that are inherently incompatible.

Upon receipt of an Update or a Push message comprising a list of transformations, the server is responsible for ascertaining whether any conflicts exist between this newly received list and the recent transformations made by other connected collaborators. In order to achieve this objective, it is necessary for the server to compare the incoming transformation list with the transformation lists from other collaborators, which are stored in the shared data structures, previously mentioned. Algorithm 3 describes this procedure.

Upon completion of the conflict detection process, a set of conflicts associated with the conflicting collaborators is returned. Each item in the set of conflicts provides detailed information regarding the nature of the conflict. This encompasses the following:

- A message describing the conflict.
- A detailed account of the transformations that caused the conflict.
- Additional context explaining the conflicting transformation.

Algorithm 3 Function to compare the received set of transformations with all the remaining sets of transformations from other collaborators in order to obtain a set of conflicts associated with the conflicting collaborators. The parameter *receivedTransformations* is a set of transformations and the parameter *sender* is the UUID of the collaborator responsible for the transformations.

function CHECKFORCONFLICTS($receivedTransformations, sender$)
  $conflicts \leftarrow \emptyset$
  for each $(otherClient, otherTransformations) \in clientsInfo$ **do**
    if $otherClient \neq senderClient$ **then**
      $conflictSet \leftarrow$ getConflicts($receivedTransformations, otherTransformations$)
      $conflicts[otherClient] \leftarrow conflictSet$
    end if
  end for
  return conflicts
end function

In order to identify conflicts between two sets of transformations, the algorithm generates the Cartesian product of the two sets, which results in all possible pairs of transformations between the two sets. Subsequently, each pair is subjected to a comparison with a shared set, which is derived from the server project and represents the common version accessible to all connected collaborators. This comparison is conducted to ascertain the potential for conflict. If a pair of transformation is identified as conflicting, it is recorded in a dedicated set that tracks all detected conflicts. Conversely, if the pair is deemed non-conflicting, it is disregarded and no conflict is raised. For instance, if the same transformation appears in both sets (e.g., both collaborators rename the same method to the same value), the transformation is recognised as redundant and classified as a shared transformation, rather than as a conflict.

Furthermore, the conflict detection algorithm employs a semantic-aware approach for the handling of modifiers, distinguishing between two subsets: the access modifiers (`public`, `private`, `protected`) and non-access modifiers (`final`, `static`, etc). This distinction is crucial, as each subset of modifiers is subject to different conflict rules. In the case of

access modifiers, a conflict arises when two transformations result in incompatible access levels for the same element.

To illustrate, if one collaborator were to transform a method to public while another were to transform it to private, the application of both transformations would result in an element with multiple (and contradictory) access levels (`public` and `private`), which is an impossible scenario. Such cases demand manual intervention for resolution, and thus are treated as a conflict. In the case of non-access modifiers, conflicts are identified when transformations result in the introduction of incompatible modifiers. In the event that one transformation introduces an abstract modifier to an element while another adds a modifier that is in conflict with the previous one, such as static or final, the system will raise a conflict, given that these modifiers cannot coexist within the same element.

A data structure is created that associates the unique identifier of each collaborator with the list of conflicts for that particular individual. The structure is then conveyed to the pertinent collaborators via a Notify Conflicts message. Upon receipt of this message, the system stores the conflict data within a comparable data structure on the recipient's local machine. This configuration allows the collaborator to access the specifics of any conflicts. The conflict information is subsequently displayed to the collaborator in the respective pop-up window.

4.3. Experiments

In order to test the application of the protocol in use with the prototype, it was necessary to create a series of scenarios and experiments designed to assess the system's capacity to detect and handle multiple conflicts, as well as its ability to ensure consistency during concurrent modifications.

To simulate the existence of multiple collaborators in a controlled environment, we developed an additional plug-in for Javardise that functionated as a automated bot. Each bot simulated an individual collaborator, performing a series of pre-defined coding tasks, such as addition, renaming or deletion of elements, as well as systems actions such as connecting to the server and pushing/force pushing the changes. This setup enabled us to reproduce real-world scenarios in a consistent manner and observe how Javardair handled collaborative coding tasks.

In the initial phases, the prototype was evaluated with a smaller Java project consisting of a single file (Figure 4.8). Three bots, each representing a distinct collaborator, were configured to execute modifications including the addition, renaming and deletion of methods. These modifications were programmed to occur concurrently, thereby simulating a realistic scenario of multiple collaborators working simultaneously.

Table 4.2 illustrates a selection of tasks designed to simulate the presence of multiple concurrent users. The results of these experiments demonstrated that Javairdair is capable of propagating and implement modifications. Moreover, the system's capacity to discern and address potential conflicts as each collaborator works locally was also validated.

```
package test;

class Contact{
    private String name;
    private String address;
    private String phone;
    private String email;

    public Contact(String name, String address, String phone,
        String email) {
        this.name = name;
        this.address = address;
        this.phone = phone;
        this.email = email;
    }

    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }

    public String getPhone() {
        return phone;
    }

    public String getEmail() {
        return email;
    }

    public String toString() {
        return "Name: " + name + "\nAddress: " + address + "\
           nPhone: " + phone + "\nEmail: " + email;
    }

    public static void main(String[] args) {
        Contact c = new Contact("Afonso", "Random Lisbon
           Street", "912345678", "afonso@mail.pt");
        c.toString();
    }
}
```

FIGURE 4.8. Java Contact class.

However, the presence of conflicts impeded the progression of predefined tasks, as the option to propagate the modifications was no longer possible due to the existence of conflicts. This resulted in the necessity for human intervention to resolve the issues.

Collaborator One	**Collaborator Two**	**Collaborator Three**
Connect to server	Connect to server	Connect to server
Rename field `name` to `fullName`	Rename field `address` to `addressInfo`	Change field `phone` type to `int`
Add method `updateName`	Rename method `getEmail` to `getEmailInfo`	Change body of `getEmail` method
Push changes	Delete method `getPhone`	Rename `toString` method to `getContactInfo`
Delete method `getContactInfo`	Push changes	Push changes
...	Rename method `getName` to `getNameInfo`	Change body of `main` method
	Push changes	Push changes
	Change body of `updateName`	Change body of `updateName`
	Force push changes	Change body of `getContactInfo` method
	Change body of `main`	...
	Push changes	
	...	

TABLE 4.2. Tasks designed for each collaborator during experimentation.

Nevertheless, the system demonstrated its capacity to promptly identify instances where human intervention was required to resolve conflicts, thereby preventing the accumulation of conflicts before their dissemination to other users, which could otherwise result in merge conflicts.

To illustrate, in the previously mentioned example of tasks, when Collaborator Three attempts to modify the body of the method `getContactInfo` after Collaborator One has deleted it, the system detects a conflict and prohibits these two collaborators from propagating any modifications. However, they are capable of making more local modifications and can also receive modifications from collaborators that are not conflicting. For example, Collaborator Two propagated the changes to the method `main` after Collaborator One and Collaborator Three's changes were deemed conflicting.

CHAPTER 5

Conclusion

The increasing complexity of software development, particularly in collaborative settings, necessitates the availability of effective tools to address the complexities of real-time synchronisation, the merging of changes and continuous integration.

It is crucial that the system is capable of accurately reflecting each user's modification across all local environments for all collaborators, while maintaining code consistency. The primary challenge arises from the detection and resolution of conflicts, particularly when multiple users are simultaneously editing the same segment of code, answering the first part of the initial research question.

In response second part of the first research question, the protocol for propagation of code changes designed to integrate CI practices into a collaborative coding environment was found to address the previously mentioned issues.

As a proof of concept for the proposed idea, we have developed Javardair, a collaborative structured code editor that makes use of a transformation-based protocol for propagation of code changes to facilitate the interaction between collaborators.

Despite their widespread use, traditional version control systems often encounter difficulties in managing concurrent modifications introduced by multiple contributors. This can result in frequent merge conflicts and their accumulation. In these systems, conflicts are typically identified at the textual level, whereby changes are compared line by line. This approach may result in ambiguity and confusion when different developers modify the same lines of code in different ways. This is problematic in scenarios where changes to the structure may not be easily reconciled through simple textual comparison (such as renaming a method).

In contrast, Javardair's transformation-based approach compares the changes at the semantic level. By representing modifications as changes in an abstract syntax tree, the system is able to more accurately identify instances of incompatibility, even when the changes affect different parts of the code. This addresses the second research question.

5.1. Drawbacks

One significant limitation is the lack of scalability. As the number of users engaged in a collaborative project increases, the process of identifying conflicts may become more time-consuming due to the heightened frequency of changes. Similarly, an expansion in the scale of the project, in terms of the number of files and the complexity of the operations involved, may also have an impact on performance, as the system is required to handle a larger set of operations and dependencies.

Additionally, the language-specific implementation represents a further limitation, albeit one that is more closely associated with the development proof of concept for the protocol. Although the protocol is designed to be generalisable, the current implementation is constrained by its reliance on a limited subset of Java syntax and a correspondingly limited set of possible transformations.

Finally, the protocol is contingent upon the capacity to extract transformations from the codebase. In order for this to function effectively, it is necessary to have a reliable mechanism for the generation and processing of abstract syntax trees or equivalent representations for different programming languages.

5.2. Benefits

Despite the drawbacks, the posed approach also offers significant benefits.

Firstly, the protocol does not necessitate the continuous comparison of the local system's state with that of the server. A local replica of the codebase situated on the server is leveraged for the purpose of tracking modifications. This results in a reduction of overhead on both the client and the server, thereby improving performance by minimising the frequency of network operations and central server checks.

Furthermore, the protocol facilitates immediate conflict detection. Upon the introduction of a change by any collaborator, the system promptly analyses the modification and assesses for inconsistencies with other collaborators' alterations. This proactive strategy guarantees that conflicts are identified at an early stage, preventing the emergence of more significant issues at a later stage in the development process.

Additionally, this concept is highly adaptable to diverse software development environments. As long as structural modifications can be extracted, the protocol can be applied to any software platform.

5.3. Future Work

One possible improvement to this work would be the incorporation of a conflict resolution mechanism. Rather than relying on conflict detection alone, the system could facilitate collaboration by enabling users to either accept or override modifications made by other collaborators, thereby providing greater control over the integration process and streamlining collaborative decision-making.

Another area for potential improvement is the visualisation and representation of conflicts. At the moment, conflicts are displayed in a separate window. An alternative, more integrated solution could involve presenting the conflicts directly within the IDE interface, with the conflicting nodes highlighted. This would enhance the user experience by providing a clear and more intuitive conflict awareness, thus enabling users to address the conflicts with greater ease.

References

[1] Getting started with code with me | intellij idea. https://www.jetbrains.com/help/idea/code-with-me.html.

[2] C. Brindescu, I. Ahmed, C. Jensen, and A. Sarma. An empirical investigation into merge conflicts and their effect on software quality. *Empirical Software Engineering*, 25:562–590, 2020.

[3] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 168–178, 2011.

[4] C. R. De Souza, D. Redmiles, and P. Dourish. "breaking the code": Moving between private and public work in collaborative software development. In *Proceedings of the 2003 ACM International Conference on Supporting Group Work*, pages 105–114, 2003.

[5] V. Enes, P. S. Almeida, C. Baquero, and J. Leitão. Efficient synchronization of state-based crdts. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 148–159. IEEE, 2019.

[6] M. Fowler. Continuous integration. https://martinfowler.com/articles/continuousIntegration.html, 2011.

[7] fubaduba. What is live share? - live share. https://learn.microsoft.com/en-us/visualstudio/liveshare/.

[8] S. Ghorashi and C. Jensen. Jimbo: a collaborative ide with live preview. In *Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 104–107, 2016.

[9] Git. Git. https://git-scm.com/, 2019.

[10] M. Goldman. *Software development with real-time collaborative editing*. PhD thesis, Massachusetts Institute of Technology, 2012.

[11] M. Goldman, G. Little, and R. C. Miller. Real-time collaborative coding in a web ide. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 155–164, 2011.

[12] R. E. Grinter. Using a configuration management tool to coordinate software development. In *Proceedings of conference on Organizational computing systems*, pages 168–177, 1995.

[13] M. L. Guimarães and A. R. Silva. Improving early detection of software merge conflicts. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 342–352. IEEE, 2012.

[14] A. Hornsby. Xmpp message-based mvc architecture for event-driven real-time interactive applications. In *2011 IEEE International Conference on Consumer Electronics (ICCE)*, pages 617–618. IEEE, 2011.

[15] B. K. Kasi and A. Sarma. Cassandra: Proactive conflict minimization through optimized task scheduling. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 732–741. IEEE, 2013.

[16] S. Kumawat and A. Khunteta. A survey on operational transformation algorithms: Challenges, issues and achievements. *International Journal of Computer Applications*, 3(12):30–38, 2010.

[17] Aditya Kurniawan, Aditya Kurniawan, Christine Soesanto, and Joe Erik Carla Wijaya. Coder: Real-time code editor application for collaborative programming. *Procedia Computer Science*, 59:510–519, 2015. International Conference on Computer Science and Computational Intelligence (ICCSCI 2015).

[18] G. Litt, S. Lim, M. Kleppmann, and P. Van Hardenberg. Peritext: A crdt for collaborative rich text editing. *Proceedings of the ACM on Human-Computer Interaction*, 6(CSCW2):1–36, 2022.

[19] H. Mcheick and Y. Qi. Dependency of components in mvc distributed architecture. In *2011 24th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 691–694. IEEE, 2011.

[20] S. McKee, N. Nelson, A. Sarma, and D. Dig. Software practitioner perspectives on merge conflicts and resolutions. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 467–478. IEEE, 2017.

[21] M. Qureshi and F. Sabir. A comparison of model view controller and model view presenter. *arXiv preprint arXiv:1408.5786*, 2014.

[22] A. L. Santos. Javardise: a structured code editor for programming pedagogy in java. In *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming*, pages 120–125, 2020.

[23] M. Shahin, M. A. Babar, and L. Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

[24] D. Sun, S. Xia, C. Sun, and D. Chen. Operational transformation for collaborative word processing. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 437–446, 2004.

[25] Z. Syahputra. Website based sales information system with the concept of mvc (model view controller). *Jurnal Mantik*, 4(2):1133–1137, 2020.

[26] André R Teles and André L Santos. Code merging using transformations and member identity. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 71–88, 2023.

[27] Federico Tomassetti, N Smith, C Maximilien, and S Kirsch. Javaparser, 2021.

[28] X. Zhang and D. Gracanin. Service-oriented-architecture based framework for multi-user virtual environments. In *2008 Winter Simulation Conference*, pages 1139–1147. IEEE, 2008.