

FREJA: A framework for effectively creating programming assignments based on code transformations

Erlend Berntsen

Master's thesis in Software Engineering at

Department of Computer science, Electrical
engineering and Mathematical sciences,
Western Norway University of Applied Sciences

November 2022



**Western Norway
University of
Applied Sciences**



Abstract

Programming assignments have become the prevalent method for evaluating programming skills and knowledge of students in computer science courses. Typically, a programming assignment is composed of several independent artefacts, such as the solution code, start code, and assignment description. These artefacts are often created in isolation from each other, but contain a redundancy of information and code between them. Maintaining consistency between the artefacts is a tedious and error-prone task, due to the extensive manual labor required to reflect a change in all the required locations. The work in this thesis has resulted in a new framework, FREJA, that aims to guarantee consistency and reduce redundancy between assignment artefacts. The goal is to make the process of developing programming assignment more efficient, and FREJA tries to achieve this by centralizing the development to a single source. Code transformations and code generation techniques are used to produce the assignment artefacts semi-automatically. FREJA is exclusive to assignments for the Java programming language, however, a small prototype has been developed to determine the possibility of a language agnostic version of FREJA. Analysis of several assignments from computer science courses at the university-level has set the basis for the design and implementation of both frameworks. FREJA was tested on several real-life assignments, and evaluated based on how well it could replicate the original structure and content of each assignment, while still maintaining consistency and reducing redundancy. The evaluation of the language agnostic prototype was similar, but put more emphasis on its ability to manage assignments in an arbitrary programming language. The results of these experiments showed that FREJA significantly reduces the number of elements in an assignment that need to be manually maintained to preserve consistency. The framework did also replicate the target assignments well, suggesting it would be beneficial and effective to incorporate FREJA into the assignment development process. The language agnostic prototype handled several programming languages, but not in a manner that was deemed effective or practical, due to its lack of consistency guarantees.

Contents

1	Introduction	5
1.1	Research Questions	6
1.2	Methodology	7
1.3	Contribution	8
1.4	Outline	8
2	Background	11
2.1	Code Generation and Transformation	11
2.2	Refactoring	13
2.2.1	Single Source of Truth	14
2.3	Model Driven Software Engineering	15
2.4	Code Documentation	17
2.5	Related Work on Programming Assignments	18
3	Design and Requirements	21
3.1	Requirements	21
3.2	Artefacts of a Programming Assignment	23
3.2.1	Solution Project	23
3.2.2	Start Code Project	23
3.2.3	Assignment Description	25
3.3	Modelling a Programming Assignment Framework	26
3.4	Defining Meta-Information	29
4	Implementation	31
4.1	FREJA	31
4.1.1	Annotations	31
4.1.2	Parsing	35
4.1.3	Execution Phase	36
4.1.4	Exercise Descriptions	38
4.2	Programming Language Agnostic Framework	40
4.2.1	First Prototype	41
4.2.2	Second Prototype	43
5	Architecture	49
5.1	Architecture of FREJA	49
5.1.1	Maven	50
5.1.2	Parsing	51

5.1.3	Transforming	52
5.1.4	Generating	55
5.1.5	Testing	56
5.2	Programming Language Agnostic Prototype Architecture	57
5.2.1	Structure	57
5.2.2	Editor	58
5.2.3	Input and Output	59
6	Installation and Usage	61
6.1	Installing FREJA	61
6.2	Example	63
6.3	Installing the Programming Language Agnostic Prototype	70
6.4	Example	71
7	Evaluation	75
7.1	A First Evaluation of FREJA	75
7.1.1	Details of the Assignment	75
7.1.2	Testing and Results	76
7.2	A Second Evaluation of FREJA	81
7.2.1	Details of the Assignment	82
7.2.2	Testing and Results	84
7.3	Evaluating a Language Agnostic Framework	87
7.3.1	Using the Language Agnostic Framework DSL	87
7.3.2	Using FREJA	88
7.3.3	Using the Conventional Method	88
7.3.4	Comparing the Different Methods	88
7.3.5	Other Programming Languages	91
8	Conclusions and Future Work	95
8.1	Summary	95
8.2	Conclusions	96
8.3	Future Work	99
	List of Figures	101
	List of Tables	103
	Listings	105
	Acronyms	107
	Appendix A: Source code	109

Chapter 1

Introduction

Most courses at informatic departments in universities revolve around programming. Students will generally have programming assignments throughout the semester to assess their knowledge and skills, which are often graded by lecturers and/or teaching assistants with some feedback supplementing the grade. Producing such assignments can be a long and tedious task, and is prone to human errors. Programming assignments are comprised of many closely related but independent parts. Often there is some starting code produced to help the students focus on the specific purpose of the assignment without them doing everything from scratch. The starting code is based directly on the solution for the assignment. The students also receive an assignment description that explains the exercises of the assignment in detail. These different parts, or artefacts, is the foundation of a programming assignment.

A common workflow and approach for creating programming assignments was identified by speaking with assignment creators and analyzing past assignments from courses at Western Norway University of Applied Sciences (HVL) and University of Bergen (UiB), such as DAT100¹, an introductory course to programming, and INF102², a beginner course for algorithms and data structures. The creator of a programming assignment generally writes the solution code before publishing the assignment, often accompanied by tests to make sure that the solution is sound. Developing the solution first may also reduce frustration on the students' end by identifying potential complexities or other problems with the assignment before it is published [51].

A copy of the solution project is made when it is finished. The solution code is then removed, and handed out to students afterwards. This project is referred to as the start code project, and is often accompanied with some unit tests. The assignment text is usually written completely independent from the assignment code. Lastly, the assignment is corrected when the students hand in their solutions. This last step is usually a combination of automated tests and manual inspection of the source code. Some courses, such as INF101³, focus more on

¹<https://www.hvl.no/studier/studieprogram/emne/1/dat100>

²<https://www.uib.no/en/course/INF102>

³<https://www.uib.no/en/course/INF101>

code quality instead of functionality. Correcting assignments for these courses relies much more on manually inspecting the solution. However, in recent years, there has been a rapidly growing development of tools and frameworks that can automatically evaluate code quality [7, 9, 23, 45].

Most of the tasks described above involves manual work by a human. A change in one part may need to be reflected in other parts to maintain consistency. For example, a change in the starting code may imply changes to certain tests and the assignment text. Since these artefacts are mostly independent of each other, manual refactoring is needed. Despite many developers still favoring manual refactoring over automatic tools, most refactoring defects are often due to manual refactoring [18, 24].

1.1 Research Questions

The current issues with programming assignments identified above lead to the following research questions which forms the basis of this thesis:

RQ1: How can we centralize programming assignment development to a single source to ensure consistency and avoid redundancy?

RQ2: How can we automatically, or semi-automatically, generate the necessary programming assignment artefacts from that source?

What we mean by a "single source" here is that the creation of the entire assignment is contained within the same project and development environment. This means that the development environment does not extend beyond the needs of the underlying assignment, which in most cases will be limited to a singular IDE. The term "project" used above and throughout this thesis always refers to what is considered a project within an IDE⁴, which is more or less synonymous with the word *application*. For example, the start code and solution for an assignment can be considered as two separate projects.

This master thesis is focused on developing a new approach and a framework for creating programming assignments. The redundancy of code and information in programming assignments is a problem we are looking to address. For example, a method name can be repeated in the assignment solution, start code, and assignment description, all of which is manually created. This is a direct violation of the Do Not Repeat Yourself (DRY)[20] principle that aims at reducing redundancy. By centralizing information and code, in conjunction with automating steps wherever possible, we intend to reduce both inconsistencies and time spent creating programming assignments. Our framework is aimed at connecting the currently independent parts so that there is always consistency between them. Code generation and code transformation plays a central role in the implementation to make this work.

Ideally, the framework would be compatible with any programming language to be as accessible and useful as possible. However, most programming courses that were analyzed at HVL and UiB focus either on Java or Python, with the occasional shift to C, Haskell, or JavaScript for more specific courses. This is

⁴Here is JetBrains's definition of a project in IntelliJ: <https://www.jetbrains.com/help/idea/creating-and-managing-projects.html>

also very similar for American colleges and universities, where Java and Python clearly dominates, followed by C++ and C [48]. To keep the project focused, and to identify a feasible path, we only worked on creating a framework for programming assignments written in Java at the start of the project. This framework will be referred to as **FR**amework for **EN**gineering **J**ava **A**ssignments (FREJA) throughout the rest of the thesis to keep an ubiquitous language. We later reevaluated to see if a more general framework is feasible, which inspired a third research question:

RQ3: To what degree is making a general language agnostic framework possible?

This is a difficult question to give any meaningful answer to by just reading, discussing and contemplating ideas that might be possible. Instead, an attempt at implementing a proof-of-concept prototype of such a framework was made to determine its feasibility. The overall purpose of this framework is the same as FREJA, with the additional benefit that a new specific framework for each programming language does not need to be created if a generic one works well enough.

1.2 Methodology

The main focus of this thesis is the development of FREJA, but some initial ground work and planning needed to be done before development could begin. At the start, a lot of research and testing went into finding suitable code transformation and code generation technologies that could be the foundation of FREJA's implementation. Afterwards, several programming assignments were studied to identify different artefacts of an assignment. The contents and structure of the artefacts were analyzed to find a common pattern. We primarily used a preexisting programming assignment from an introductory programming course at HVL, DAT100, that focuses on Java, to base the design and implementation of FREJA on. Two other assignments from the same course was then used to test and evaluate FREJA. The purpose of these tests was to evaluate the versatility of the framework, and ensure that it could handle other assignments than just the one it was mainly designed after. The evaluations were based on how well the framework could replicate the target assignment, while also maintaining consistency and reducing redundancy between artefacts. Additionally, testing on other assignments allowed us to identify shortcomings of the framework. This served as a way to change and improve the implementation of FREJA in an iterative process.

As for the third research question, an entirely separate implementation of a prototype was created. This prototype was tested with several programming exercises in different programming languages that differ drastically in style and structure to determine its flexibility. Finally, it was compared to other methods of creating programming assignments, such as the conventional method of manual development, and FREJA, to compare its functionality and effectiveness to those.

1.3 Contribution

The work in this thesis has resulted in the creation of FREJA, a framework aimed at producing Java programming assignments more effectively, and a prototype implementation of a programming language agnostic framework with the same purposes, but targeted at assignments in any programming language. Links to the source code for both of these frameworks can be found in *Appendix A*. Figure 1.1 shows an overview of the workflow for developing a programming assignment, both using FREJA and the conventional method of manual development. A yellow box represents an artefact that is manually developed, while a green box represents an artefact that is semi-automatically generated. The figure highlights the principle idea of FREJA of centralizing assignment information into a single source that needs to be managed. We hope that this framework can have an impactful contribution in the field of programming assignments. Moreover, we hope that an adoption of FREJA in a real-life setting would be beneficial and bring forth a more effective assignment development process.

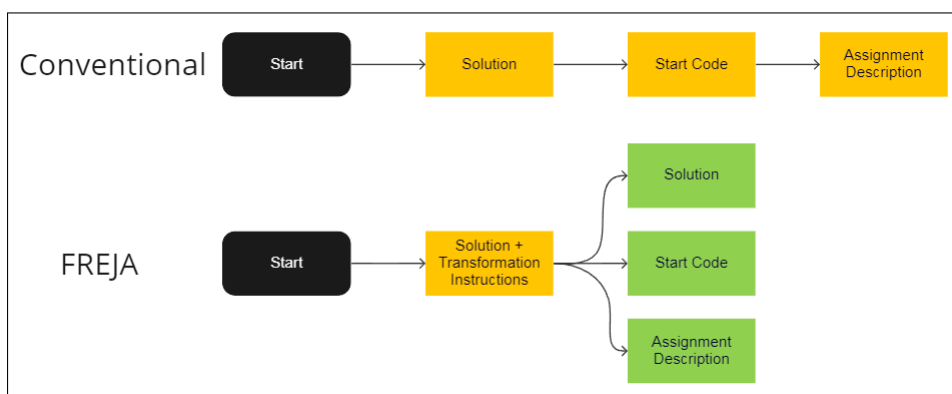


Figure 1.1: Workflow of FREJA compared to the conventional method of creating programming assignments.

1.4 Outline

The rest of this thesis is structured as follows:

Chapter 2: Background Background information and related work that are relevant to the topic of this thesis are presented and discussed.

Chapter 3: Design and Requirements This chapter covers the design and requirements of both FREJA and the programming language agnostic prototype that is implemented, as well as explaining the overall new approach to creating programming assignments.

Chapter 4: Implementation In this chapter, an overview of the implementation of FREJA and of the generic framework is shown and discussed. The focus is mostly put on the implementation details that concern end-user interaction.

Chapter 5: Architecture This chapter puts more emphasis on the source code behind the implementation of FREJA and its generic counterpart. The architecture of the source code of both frameworks is explained, as well as code logic and other necessary background information that is important to understand for anyone that is interested in continuing the work presented in this thesis.

Chapter 6: Installation and Usage This chapter thoroughly explains how to install FREJA and the programming language agnostic prototype, and how to use them. A simple tutorial is presented for each framework that demonstrates how to use them on a small example assignment.

Chapter 7: Evaluation The experiments that were conducted to both of the frameworks are explained in this chapter. Afterwards the results of these are presented.

Chapter 8: Conclusion and Future Work The final chapter summarizes the work done in this thesis, as well as discussing the main conclusions that can be drawn from the results, in the context of the research questions. Finally, there is a small section detailing possible further work that can be done on FREJA or other related projects.

Chapter 2

Background

In this chapter, relevant theory that is important for understanding both design decisions and implementation details of each framework is presented. Related work that this thesis draws inspiration from are presented and discussed. There are also comparisons between the work done in this thesis and the work of others to demonstrate the differences between them.

2.1 Code Generation and Transformation

The term *low-code* has had a steady increase in popularity the last decades [11], which focuses on quickly producing applications with minimal effort. Low-code programming is based upon model-driven software development, rapid application development, automatic code generation and visual programming [54]. The frameworks implemented in this thesis are also inspired by these paradigms, with less emphasis on visual programming.

The implementation of low-code tools relies heavily on code transformations and code generation. These techniques are the foundation of *metaprogramming*, which is based on creating programs that use other programs as data, either to generate new programs or modify existing ones. Such programs are called metaprograms, and are written in a metalanguage. The object language is the language of the generated or transformed program. It is called *homogeneous* metaprogramming if the object language and metalanguage are the same, and *heterogeneous* metaprogramming if they differ. Many programming languages today have official support for metaprogramming, such as C++ and Java, while the inception of metaprogramming can be traced all the way back to macros in Lisp [33]. Metaprogramming tools in general purpose languages are often used for homogeneous metaprogramming, while a dedicated metaprogramming language, such as Rascal [27], is often used for heterogeneous metaprogramming.

Whether its with the help of built-in metaprogramming functionality, such as reflection and annotation processors for Java, or through a dedicated metaprogramming language, the plethora of code generation and transformation tools available today are all rooted in metaprogramming techniques. One such tool is Meta3, a conceptual framework for creating code generators for Domain Specific

Language (DSL)s [29]. This framework identifies several layers of a system for generating code from DSLs. These layers are divided into input languages and meta-models, conceptual meta-models, and output artefacts and related meta-models. Changes between these layers are done by Model-To-Model (M2M) and Model-To-Text (M2T) transformations. Other transformation tools, such as the C/C++ refactoring tool Nobrainer, are built and used in the same language, and argues that avoiding using DSLs increases the ease of use [47].

Xevolver is a framework for code transformations that lets developers specify their own code translation rules [53]. It is mainly focused on transforming code by optimizing it for better performance of applications. This is done by turning the source code into an Abstract Syntax Tree (AST) and converting the tree into a XML document. User-defined code transformations can then be written with XML data transformation tools. Xevtgen [52] builds on similar ideas but generates the transformation rules automatically based on input and output code patterns. Other transformation frameworks stray away from user-defined transformations, and instead make use of machine learning [31].

Common for most of these code transformation tools is the target focus on large established code bases. The cost of manually rewriting enormous amount of code may be too large and not economically beneficial. However, with transformation tools as described above, developers can still focus on new features and functionality, while old code is automatically transformed in the background. The purpose of these transformations may vary. Some are meant for refactoring code to improve code cleanliness and structure, others focus on optimizing the code as much as possible for hardware limited systems. It is usually the case that large code bases have been in development for years, even decades, and migration from older language or API versions to newer ones for such code bases can be done more efficiently through code transformations.

Code transformations are typically done at the structural level, while the semantics remain the same. Unit tests can be created and executed before and after a transformation to ensure such transformations do not change the functionality of the application. Transformation tools recognizes certain structural code components, which are then used as inputs for a transformation rule. Some even use syntactic isomorphism [38], i.e., code statements that differ in their syntactic structure but are identical in their semantics, to find more code components eligible for transformations. A transformation rule specifies precisely what to transform the input into, often referred to as the output of the transformation rule.

The work in this thesis differ from these tools in several aspects. Firstly, FREJA does not aim to rewrite hundreds of thousands lines of code, but rather be a tool to efficiently produce small to medium-sized programming assignments. Secondly, the transformations that take place are not because of the need to transform old, inefficient or bad code into something that is objectively better, but rather as a way to ensure consistency and reduce the amount of repeated information. Finally, most of these tools have a clear objective, with specific code components that the transformation rules look for, and a predefined output of those rules. FREJA does also look for specific components in a programming file, but it is up to the user to mark sections of the source code that should make up a specific component. These components are meaningful in terms of a

programming assignment and are used as input for a transformation rule. For example, marking a section of code as a solution to an exercise would create a *solution* component, and handled by a solution transformation rule, as opposed to a typical transformation tool that uses structural components in the programming language for transformation rules.

The user of FREJA also has full control over what the transformation should do, e.g., they can specify that a code component should be removed, or swapped out with some other code. In short, the framework only provides the tools to perform code transformations on a code base, without any presumptions on what should be transformed, or how to transform it. It is entirely up to the user to specify those aspects.

2.2 Refactoring

Syntactical and structural code transformations as discussed above are often referred to as refactoring, that is, changes to the code that improves it in some manner without changing its functionality. In practice, the definition of refactoring does not only encapsulate semantic-preserving changes [26]. A comparative study of manual refactoring and automatic refactoring shows that automatic refactoring saves time across the board [37]. There are differences in how much time an automatic refactoring saves compared to a manually refactoring, based on what type of refactoring it is. For example, *Extract Method* is a type of refactoring where some code gets extracted out of a preexisting method to a new method, when the initial method becomes too large, while a *Rename Field* refactoring simply changes the name of a field in a record data structure, or a field variable in a class [17]. The first type of refactoring saves the most time being done automatically instead of manually, while the latter has the smallest difference. Several studies show many more benefits to refactoring in addition to time-saves, such as increasing code maintainability, readability and reusability, while other studies highlight possible negative effects such as increasing the amount of bugs [26].

Popular IDEs today bring many tools and functions for automatically refactoring source code. However, the limitations of these are much of the inspiration for this thesis. Typically, these automatic refactorings only do code changes for a single project, and mostly just targets programming files. Other files in a project that may be connected to the source code in some way, such as an assignment description text file, will not be affected by such refactorings. As mentioned, a programming assignment is usually split into several projects, for example, one project contains the solution to the assignment, while another project contains the start code that the students receive. Both of these projects have code that are related to each other, but an IDE has no way of knowing this relationship. Using an IDE's automatic refactoring tool to rename a function in one project would not also rename a function with the same name in another project.

These limitations of automatic refactoring tools forces the use of manual refactoring in a scenario like this. Even with the current abundance of automatic refactoring tools, one study shows that there are still many developers that pre-

fer manual refactoring in some cases [26]. Although it is not specified, it is most likely that this concern refactorings within the same project. However, there is an argument to be made that manual refactoring across multiple projects would not be favorable. This is because manual refactoring within the same project can use compilation errors or warnings as a crutch. For example, if a developer forgets to manually refactor some part of a project, an IDE would likely give a warning and guide the developer to the location of the issue. The same cannot be said if someone forgets to refactor a semantically related part in another project.

Issues with semantically related code or dependencies across multiple projects and code bases is nothing new. This thesis tackles a small-scale version of problems that developers have encountered for a long time when integrating or changing Application Programming Interface (API)s. Providing functionality through an API have given the opportunity for developers to work in parallel on different parts of the same application. One of the major drawbacks of an architecture like this is the unpredictable nature of software engineering most certainly guarantees performing unplanned changes to an API. This forces clients of the same API to use unwanted time and effort to change their own code to reflect the changes in the API [49]. API migration is often done manually and most of the changes that break preexisting applications are refactorings [15]. Synchronization between the clients and creators of an API is a problem of restoring consistency between different instances of the *same thing*. The field of databases has tackled this problem for several decades, and clever innovations started with the birth of the internet.

2.2.1 Single Source of Truth

As the internet grew bigger and enterprises began to take their businesses online, a way of handling the increasing load on their servers was needed. Much of the time the bottleneck was the slow database queries that could cause data to be unavailable for extended periods of time, depending on how it handled concurrent access. One solution to this was to replicate the data to several database servers, so that several users could interact with the same data at the same time. There are several ways to replicate data, *lazy replicating* focus on achieving higher performance and availability but sacrifices consistency in doing so, while *eager replication* prioritizes consistency in cost of performance and scalability. The former option was the one that got preferred and adopted in the infancy of distributed database systems [22]. Gray et al. [19] pointed out the pitfalls of the early primitive lazy replication solutions, showcasing potential increase in response times and deadlocks.

Because of this, there was a growing need for a better way of handling inconsistencies and synchronization problems when replicating data, similar to the problem of having distributed assignment projects with replication of information and code between them. The *Single Source of Truth (SSOT)* architecture emerged later, putting more emphasis on retaining consistency between replicated data. This architecture is built on the idea of having only one place to edit data or information in a distributed system. Data that is replicated, or in some way linked to the original data, only have a reference to it. In this way, updates to the original data will be propagated to any other system that

reference it, greatly increasing consistency over other methods. Other software architectures are often used to implement some sort of SSOT, such as the Enterprise Service Bus (ESB) architecture, Service Oriented Architecture (SOA), and Data Warehouse (DW) [57]. Some methods handles SSOT even without data replication, shifting the responsibility from the data layer to the service layer [40].

Constructing programming assignments does not require the same level of infrastructure as big enterprise services with distributed systems. Usually, only one or a few people are authoring an assignment at a time. Important features such as availability and scalability for enterprise services are therefore not a big concern for FREJA. The different artefacts of an assignment also include a lot of repetition of some code or information, but are rarely a direct copy in the way replicates of data in database systems are. Although the SSOT architecture is usually targeted at large enterprise systems, this thesis still draws inspiration from the architecture for achieving better consistency between different projects and artefacts. FREJA combines ideas from large database systems with code generation and code transformation techniques, allowing for one single project to be the source of information that can replicate certain code fragments or information to other projects through code transformations and code generation.

2.3 Model Driven Software Engineering

One domain that also utilizes transformations and code generation tools to increase efficiency in software development, is Model Driven Software Engineering (MDSE). Within MDSE, central concepts of a problem domain are abstracted away as models, which are then used for varying purposes, such as being used as sketches, blueprints or programs. There is a distinction between *prescriptive models*, that define how the target system should be implemented, and *descriptive models*, that describes a system in a bottom-up fashion by abstracting away implementation details [10]. Prescriptive models are often used in constructing systems in a top-down process by model transformations and code generation.

Central to the topic of MDSE is the idea of a concept being represented at different abstraction levels. Prescriptive models in software development often use a model at a high level of abstraction as the starting point for code generation, with M2M transformations that reduce the abstraction level until a M2T transformation finally transform a model into code. This too is an example of several concrete artefacts that are semantically related but can quickly become inconsistent. The Eclipse Modeling Framework (EMF) [43] provides the ability to describe object classes and hierarchies through graphical diagrams, which can then be used to generate the corresponding code. However, the prominent difficulty with such tools is figuring out how to deal with inconsistencies between the generated code and the original model. What happens when the generated code is edited and afterwards the code generation tool is executed again from the same source model?

This problem occurs when inconsistent models needs to synchronize so that they are consistent again. *Bidirectional Transformations (BX)* is an attempt at defining a consistency relation between different models that can be used

to figure out how to restore consistency again. The background of BX can be traced back all the way to the 80s, from what is now known as the *view-update problem* in databases [3]. This problem boils down to having a *view* that is composed of data from several tables in a database, while also providing the opportunity for an user to change the data that is presented in the view. This update does not have an unambiguous way of handling the change in data from the view back to the database.

BX describes the notion of *lenses*, which defines functions to restore consistency between models [2]. A *symmetric lens* refers to consistency restoration functions for models that contain some common information, but also some private information that other models do not know about [21]. For *asymmetric lenses* however, there is a clear distinction between the models. One model act as the *source* S and the other as the *view* V . Two functions are then defined

$$get : S \rightarrow V$$

$$put : S \times V \rightarrow S$$

The *get* function creates the view from source, while the *put* function produces a new source from an updated view and the old source. A forward consistency restoration, i.e., restoring consistency to the view from the source, is trivial with an asymmetric lens. This is because the source completely defines the view, and consistency restoration can be done by just applying the *get* function. While backward consistency restoration is just the *put* function, it can be difficult to define a implementation for the function so that it restores consistency in a meaningful way. To demonstrate this difference, imagine a source that consists of two integers, 1 and 2. Furthermore, the *get* function is just the operation of adding the numbers in the source together, thus the view is also an integer, the sum of the source. If a number in the source was changed, e.g., the 1 was changed to 4, restoring consistency between the models is very easy, since the *get* function can be applied and the view would update from 3 to 6. On the other hand, if the view was changed instead, e.g., from 3 to 5, it is not so clear how this change should propagate back to the source so they become consistent again. Remember, they are only consistent if the view is the sum of the source. The models could become consistent again by updating 2 to 4 ($1 + 4 = 5$), which was calculated by subtracting the other number in the source from the sum, but the source could also be updated by changing the 1 instead, using the same method. There is no distinct method of updating the source from the view that could be deemed as "correct". Although this example is very simple, it demonstrates that a *get* function in an asymmetric lens is much easier to define than a *put* function.

At least for FREJA, an assignment will always have one project that acts as the source of information, and can be seen as the source in an asymmetric lens. The *get* function is defined partially by the framework itself, but mostly by the creator of a programming assignment. It produces the view by transforming the code from the source project. More importantly, there is no need to define a *put* function for backward restoration since there is only one source of information. This is because there is never a point where FREJA needs to use changes made in the generated artefacts, i.e., the view, to update the source project, i.e., the source. As will become apparent later, the FREJA *get* function does not follow

the exact BX definition. This is due to difficulty surrounding the assignment description, especially the elements that concern code documentation.

2.4 Code Documentation

The rise in popularity of agile software development methods can largely be credited to Robert C. Martin, who was one of the first to preach the agile methodology. The agile approach to software development follows certain rules and practices. One rule, called *Martin's first rule of documentation*, says to “produce no document unless its need is immediate and significant” [35]. His reasoning for this was that source code is bound to change and the documentation for it will inevitably be outdated, or worse, outright misleading. He argued that the only true documentation is the source code itself. In his book, *Clean Code*, he explains that clean code requires self-documenting, or self-explanatory code [34]. The *literate programming* paradigm also emphasises the use of natural language when developing programs. This is done by incorporating macros inside an explanation of the program in a natural language, resulting in a file that documents the program with the code embedded inside the documentation [28]. The opposite of the literate programming paradigm is the field of code documentation generators. Many can agree with Martin's notion of the truth being in the source, which is why there has been many attempts at creating tools that generate documentation from the source code itself.

Javadoc¹ is one of the more well-known tools that does exactly this. It was created to eliminate the need for dedicated technical writers that wrote standalone documentation for the Java source code. The tool is bundled with the Java language. Special comments, called *Javadoc comments*, are used directly in the source code to write documentation about the code. These are then processed and transformed into HTML documents that uses hyperlinks to link together documentation for related code. This is done through doclets², which are Java programs that determine the format and content of the documentation that gets produced. The *Standard Doclet*³ creates documentation as described above, but it is also possible to create custom doclets that produces documentation differently, even in other file types than HTML, such as XML or PDF. Javadoc works well to document API specifications, but is limited in its ability to create API documentation that is more useful for developers [30]. This other type of documentation that is targeted at developers is much larger, with elaborate explanations and examples of what the code does, as well as documentation of potential bugs.

There are many other tools in the code documentation domain. Swagger⁴ is a framework that can document REST APIs by generating documents that conform to the Open API Specification⁵. These documents are programming language agnostic and can be used for a multitude of things, such as generating

¹<https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html>

²<https://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/doclet/overview.html>

³<https://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/standard-doclet.html>

⁴<https://swagger.io>

⁵<https://spec.openapis.org/oas/v3.1.0>

websites that visualises an API. Other tools, such as VisUML [16], can produce real-time UML diagrams from source code. Doxygen [5] is another document generator that also uses special comments to generate code documentation, but works with many more programming languages than Javadoc.

As mentioned, one component of traditional code documentation that is sometimes missing, is a thorough explanation of the code in a natural language. Such explanations document the code by describing its behavior and intent. There are several ways of explaining code in this manner, such as a line-by-line explanation, or a more abstract summary concerning the purpose of the code [56]. In recent years, there has been a growing interest and success within the realm of AI and natural language processing [25]. This has naturally led to innovative work concerning the natural language of code explanations also. Codex [12] is a language model, trained on public Python code from GitHub. It can write source code using a natural language explanation of the desired code as an input prompt. Codex was also trained to do the same thing in reverse, that is, writing a natural language explanation of some source code given as input. Moreover, one particular version of Codex powers GitHub Copilot⁶. Dubbed as an "AI pair programmer", GitHub Copilot can provide smart code suggestions from short natural language code explanations. However, not all suggestions are correct in terms of the desired behavior, and some suggestions can be simplified further [39].

2.5 Related Work on Programming Assignments

There is a lot of related work done on programming assignments in an educational context when it comes to correcting and giving feedback to solutions automatically or semi-automatically. Florencia Miranda from the University of Chile developed a tool to give more specific feedback to students when their solution was incorrect, as a way of having more positive effect on student engagement, while also giving tutors a better overview of different problems that students encounter [36]. An inquiry-based tool with conceptual feedback is also shown to help students produce better code, compared to normal code coverage feedback [13].

Other studies have shown that real-time feedback on assignments can enhance the learning experience of students, and a scoreboard based on the grading results can motivate students to work harder on the assignments before submitting them [32]. Penalties can be introduced to stop students from relying too much on autograders [4]. This results in students testing their own solutions more before checking it with an autograder. CloudCoder [41] is a web-based open platform for sharing programming exercises. The motivation behind it is to enable free a distribution of exercises to increase better reuse between institutions. Another purpose of the platform is to collect data about students and programming exercises, to better understand how students learn to program.

The work done on the creation side of programming assignments is much more scarce. OpenAI Codex was mentioned earlier as a tool for creating source code from natural language code explanations, and vice versa. Codex has also been

⁶<https://github.com/features/copilot>

used in the context of programming exercises, both as a tool for generating source code for exercises, and as a tool for creating natural language exercise descriptions [46]. CrowdSorcerer [42] is a tool that combines crowdsourcing and programming assignments. Instead of following the normal approach of having an instructor making the assignments, students can instead make their own assignments through an online interface. Other students from the same course can access these assignments to test themselves, and evaluate the quality of the assignment. Students must create all assignment artefacts themselves, such as the solution, start code, test cases, and description text, before uploading them. CodeWrite [14] is another tool that also focuses on student-driven programming exercises. Exposing students to the code of others, and gaining deeper knowledge of the curriculum are some of the benefits for tools like these.

nbgrader [8] is a tool for creating and grading assignment in Jupyter Notebook⁷. Jupyter Notebook is a web-based interactive development environment for creating notebook documents, which are traditionally used to document research and other engineering related work. The core idea and purpose of nbgrader is similar to FREJA when it comes to producing assignments. It centralizes assignment artefacts into a master copy that includes tests and the solution. The master copy is then used to generate the version of the assignment that the students receive, which has the solution removed. The creators of nbgrader have also realized the benefit of avoiding maintaining several copies, or different artefacts, when creating assignments.

However, the use of Jupyter Notebook in an educational context is often aimed at students from other fields than computer science, such as mathematics and mechanical engineering, to teach them basic computational skills [8]. FREJA differs in this aspect, which targets programming assignments used in courses for computer science majors. These courses do not use notebooks to teach software engineering, but rather industry standard software development environments and methods, such as using a traditional IDE instead of a notebook.

There has also been work done on automatically generating programming exercises [50]. These are randomly created programs within a certain domain of programming, e.g. bit-level programming, that should be implemented from scratch by students as an exercise. The exercises that were automatically created were used as a supplement for students, on top of manually produced exercises by the teachers. FREJA attempts to simplify the process of manually creating larger assignments that are tailor-made for the curriculum, not to automate the content of the assignment itself. Creating high quality assignments is not an easy task. Different aspects such as difficulty, relation to current class topic, and relevance to real-world scenarios must all be delicately balanced, while still being interesting for students [51]. Taking all of these aspects into meaningful consideration is difficult for an automatic assignment or exercise generator, which is why this thesis focuses on improving the development of manually designed assignments instead.

⁷<https://jupyter.org>

Chapter 3

Design and Requirements

This chapter presents the high-level design of FREJA and its language agnostic counterpart. Requirements that were established at the start of the thesis are explained. The design part will briefly discuss technologies, theory and methodology used in the design, and explain the reasoning behind certain design decisions.

3.1 Requirements

The requirements discussed here are primarily targeted at FREJA, and not the language agnostic prototype. The reason for this is that the purpose of the prototype is to be more of a proof-of-concept, rather than a complete framework like FREJA that aspires to be incorporated into real-life assignment creation. A baseline for the language agnostic framework is that it has a solution to the problems posed in the first two research questions. That is, it centralizes development to a single source that reduces redundancy and ensures consistency, while also providing a way to generate the necessary artefacts from that source. Of course, the implementation of the framework aims to meet all the requirements described below, however, any requirement may be sacrificed in the favor of realizing the language agnostic aspect of the framework.

Most of the requirements for FREJA are non-functional due to the difficult nature of defining appropriate functional requirements that can accurately measure the impact of the framework. Even with a well-defined measuring unit for a functional requirement, it is still a difficult task to determine the acceptable values of the chosen unit of measure. The non-functional requirements are confined into several explicit points so they can be easily referred to later in the thesis. They are as follows:

1. **Effectiveness:** For the framework to be successful, it must be more effective than the conventional manual way of creating programming assignments. In other words, the framework needs to reduce the effort and time spent writing assignments. How this is achieved is not important, as long as there is a decent beneficial gain of using FREJA.

2. **Consistency:** As stated in the first research question, the goal is not only to reduce the amount of inconsistencies between the different artefacts of an assignment, but to *guarantee* that they remain consistent.
3. **Non-redundant:** The first research question emphasized the need for avoiding redundancy across the artefacts of an assignment. In reality, there will be repetition between the artefacts nonetheless, but the goal is to maintain the repeated information from a single source. More specifically, this point is aimed at reducing the amount of information or code that need to be manually maintained
4. **Accessibility:** This requirement concerns the accessibility of FREJA for those who use it. For it to be easily accessible, it needs to be platform independent, i.e., not specific to Windows, Linux or Mac OS. It should also be easy to retrieve and install. The users of FREJA are naturally programmers, which tend to be stringent on their development environment and which tools and software they use. Because of this, an emphasis was made on making the framework independent of whatever IDE is used for development. This means that one of the early ideas of creating an IDE plugin is not a suitable solution.
5. **Dependencies:** A consequence of the previous requirement is to limit the amount of dependency on other tools or software as much as possible. In this context, a dependency is considered as something that the user need to acquire themselves before being able to use the framework. It is not considered a dependency if there is some underlying software that the framework relies on, as long as it is bundled in the installment of the framework.
6. **Learning ability:** These last requirements are about the design of the program in regards to the user experience. The implementation of FREJA can be thought of as its own small language. This "language" should then be easy to learn, meaning the amount of new syntax and keywords should be kept to a minimum, but sufficient to cover all the needs of the user. Also, the framework should avoid relying on deep knowledge of any third party tools or software.
7. **Ease of use:** In addition to being easy to learn, it should also be easy to use. The syntax should be short but descriptive and meaningful. The framework should also provide means for helping the user when they are using it. Standard IDE functions, such as auto-complete and code suggestions, should also extend to the syntax of the framework. The user should be informed when syntactical errors are made, with an explanation of what caused it and where it happened. Syntax warnings should also be presented as early as possible to reduce run-time errors. Likewise, any semantic wrongdoing should also be caught and presented to the user, along with a description of the issue.

3.2 Artefacts of a Programming Assignment

To get a better understanding of programming assignments and the artefacts that they are composed of, we analyzed assignments from courses where we had access to all artefacts. This was important in terms of getting an overview of where different programming assignments differ from each other, and where they share commonalities. One of the courses was INF102¹ at UiB, an introductory course to algorithms and data structures aimed at second-year students. The other course was DAT100² at HVL, an introductory course to programming for first-year students. Both courses teaches concepts from the syllabus through examples and assignments, exclusively using Java as the programming language of choice. The courses did also have weekly exercises that were completely voluntary, which were sometimes more focused on theory than implementation. These were not analyzed for this thesis, however, the larger compulsory assignments from both courses were studied in detail. After analyzing the structure of these programming assignments, a common pattern and structure was identified.

One of the assignments from DAT100 was extensively studied and primarily used to define implementation details and design decisions for FREJA. The assignment was split into several exercises that concerned calculations and visualizing of GPS sensor data from sport watches or fitness apps. The goal of the assignment was to asses the students skills and knowledge of the syllabus, which included input-output operations with local files, object-oriented programming, simple mathematical methods, and use of external libraries. Examples and code listings from this assignment are used throughout the thesis to explain things in more detail.

3.2.1 Solution Project

An assignment often has two to three corresponding projects, each with different properties and accessibility. One project that was common for all assignments that were analysed was the solution project. This project contains the entire solution to all the exercises for the assignment, and is shared to an online cloud service for sharing Git repositories, such as GitHub or GitLab. Only lecturers and teaching assistants have access to this repository. The solution project is commonly used as a guideline when evaluating the student solutions. It is not uncommon for the repository to be open for everyone to view after every student has received their grading. This is so they can learn and study from the intended solution. The online repository for the aforementioned DAT100 assignment³ shows that the solution project does not contain much information besides the source code for the solution.

3.2.2 Start Code Project

Another separate project is created for the start code that the student receive at the beginning of the assignment. It is not always the case that students receive any starting code at all, but this usually only happens for courses later

¹<https://www.uib.no/en/course/INF102>

²<https://www.hvl.no/studier/studieprogram/emne/dat100>

³<https://github.com/lmkr/dat100-prosjekt-gps-complete>

in the studies when they are more experienced programmers. For the introductory programming courses that were studied, some skeleton starting code is generally given as to not overwhelm students when they are first starting out. Additionally, there is often unit tests accompanied with the start code project. An example of this can be seen in Listing 3.1, which is an excerpt from the start code for the first exercise in the DAT100 assignment⁴. The exercise is to implement the constructor, getters and setters, and field variables of the `GPSPoint` class, which still have their skeleton implementation in the start code.

```

1 public class GPSPoint {
2     // TODO - field variables
3
4     public GPSPoint(int time, double latitude, double longitude,
5         double elevation) {
6         // TODO - constructor
7         // Remove the throw statements as the methods get implemented
8         throw new UnsupportedOperationException("TODO.constructor("
9             GPSPoint");
10    }
11
12    // TODO - get/set methods
13    public int getTime() {
14        throw new UnsupportedOperationException("TODO.method()");
15    }

```

Listing 3.1: An excerpt of the start code for an exercise.

Another reason for giving out starting code is the ability to predefine class and function names. Having a standardized structure and naming for the code makes it much easier to create a test suite for the assignment. The benefits of this are many. First of all, unit tests can be accompanied with the start code for the students to help them check their solution as they go along. This is also very useful for lecturers and teaching assistants to guide students if their program is not functioning correctly. Unit tests can then be used to quickly determine the root cause of the problem. Lastly, hidden tests can be created to determine the correctness of the solutions, speeding up the process of grading assignments. The unit test in Listing 3.2 is from the test project⁵ from the same DAT100 assignment, and is a test for the exercise in Listing 3.1.

```

1 private GPSPoint g;
2 private int T_TIME = 1;
3 private double T_LAT = 2.0;
4 private double T_LONG = 3.0;
5 private double T_ELEV = 5.0;
6
7 @BeforeEach
8 void setUp() throws Exception {
9     g = new GPSPoint(T_TIME, T_LAT, T_LONG, T_ELEV);
10 }
11
12 @Test
13 void testgetTime() {
14     assertEquals(T_TIME, g.getTime());
15 }

```

Listing 3.2: An excerpt of a test to the above exercise.

⁴<https://github.com/dat100hib/dat100-prosjekt-gps-startkode>

⁵<https://github.com/dat100hib/dat100-prosjekt-gps-testing>

Previously, the start code projects were handed out through zipped (compressed) folders with the source code. Nowadays, cloud file-sharing technologies have become increasingly popular and accessible so that even students with no experience in software development can quickly learn to use them. As with the solution project, the start code project is also shared to one of the popular repository cloud-hosting services. The students then get their own local copy by either cloning or forking the original repository.

The content of the start project can vary, but it more or less includes every file from the solution project. Of course, the actual solutions to the exercises are removed. Only the bare-bone structure of the solution code is given, potentially with some helper classes and functions that is out-of-scope or unnecessary for the students to implement themselves. There may also be differences in the tests that are in each project. For example, some tests in the solution project may only serve as verification that helper classes and functions behave as intended. On the other hand, the project that the students receive generally includes tests that only checks that the implementation of the exercises works as intended.

3.2.3 Assignment Description

Lastly, the students also obtain an assignment description that describes each exercise, explaining exactly what the student should implement. If the starting code includes several implemented classes, functions, or libraries that students must use, a thorough explanation of how to use them is often included in the description as well. The assignment text usually have a hierarchical structure since an assignment is often composed of several exercises, and each exercise has the possibility of containing sub-exercises, and so on. These exercises are structured as an ordered list, starting with exercise 1 and counting upwards. Sub-exercises are also ordered lists, often with alphabetical instead of numerical numbering. An example of an exercise description can be seen in Fig. 3.1.

Exercise 1: Class for GPS datapoints

In this exercise you will implement a class `GPSPoint.java` for representing GPS data points.

a) Constructor

Look at the start code for the class `GPSPoint.java` in the package `no.hvl.dat100ptc.exercise1`.

Expand the starting code with a constructor that can give values to all the field variables. :

```
public GPSPoint(int time, double latitude, double longitude, double elevation)
```

Test your implementation by running the tests in the test-class `GPSPointTester.java`

Figure 3.1: An example of an exercise description.

This is the exercise description for the same sub-exercise seen previously in Listing 3.1, but simplified and condensed, in addition to being translated from

Norwegian to English. This example highlights the exercise and sub-exercise hierarchy. It also shows that an exercise description contains direct references to the source code, including the filename, package location, test-class filename, and a constructor definition. These references are hard-coded and must be manually updated if they are changed later in the source code.

How the assignment description is handed out varies between courses, but each course tend to use the same method throughout the semester. Some courses deliver the description separately from the start code project, while others include it in that project. The file structure also differs from course to course, with some preferring centralizing everything into one file. Other courses split up the description, usually with one file for each exercise. This is the case for the assignment description for the DAT100 example⁶ that has been discussed throughout. The file format is generally PDF if the description is handed out separately, or Markdown if it is included as part of the start code project.

3.3 Modelling a Programming Assignment Framework

The last section identified three key artefacts in a programming assignment: the *solution project*, the *start code project*, and the *assignment description*. The assignment description has a close relationship with the start code project by having references to the source code. Actively linking the assignment text to the source code would allow for changes in the source code to be automatically updated in the text. A way to do this is by generating or transforming the assignment description directly from the source code. It is also possible to construct the hierarchical structure of an assignment description this way, by including extra meta-information in the source code, such as specifying what class or filename corresponds to an exercise.

Even with the ability to generate a template-based assignment description just from the source code, it is not reasonable to believe that this is good enough in all scenarios. It is often the case that an exercise description explains the overall goal of the exercise, along with a loose explanation of how the exercise should be solved. Most exercises can be solved in a multitude of ways, but there may be only one way that is correct with respect to the current class topic. For example, an exercise concerning iterating an array may be intended to be solved with a for-loop instead of a while-loop. Such semantics is difficult to automatically infer from the source code alone. Studies also show that the most crucial part of source code descriptions is an explanation of the intended behavior of a method or a program [1]. Therefore it is important for the connection between assignment text and source code to allow manually editing the description after generating it. In this scenario, an exercise description can have a thorough explanation of the code and its intended behavior, and also be specifically facilitated towards the curriculum for a course. At the same time, the description can still automatically update any code references in the text when the referenced source code changes. An example of a detailed semantic code explanation can be seen at the start of the description for exercise 2 in the

⁶<https://github.com/dat100hib/dat100-prosjekt-gps-testing/tree/master/docs>

DAT100 assignment, show in Fig. 3.2.

Exercise 2 - GPS data reading and arrays

This part of the project concerns how GPS data can be read from a file and how the GPS points in the file can be represented using an array of references to `GPSPoint`-objects.

The class `GPSDataFileReader.java` contains finished Java-code for reading in a CVS-datafile with GPS points in the format explained earlier. This Java-code is not necessary to change, and it is only later in the course where we will learn how to read from - and write to - files in Java.

The focus for this exercise is to implement methods in the class `GPSData.java` that should save GPS datapoints that were read in an array of `GPSPoint`-objects, and the helper-class `GPSDataConverter.java` that contains helper methods for converting the data to `GPSPoint`-objects.

Figure 3.2: An exercise description that explains the semantics of the exercise.

Another observation that was made is the subset-like relationship between the start code project and the solution project. This relationship can also make good use of code generation and transformation techniques. With extra meta-information in the source code one could, for example, specify what code fragments constitutes as the solution to an exercise. Additionally, specifying what to do with an exercise solution (or other code fragments) would allow a transformation tool to completely generate the start code project from the solution project. Listing 3.3 shows the solution to the exercise described previously in Fig. 3.1, while Listing 3.4 shows the start code that the students received for the same exercise. This example showcases that an exercise solution is not necessarily just removed, but can also be replaced with comments and different code.

```
1 public GPSPoint(int time, double latitude, double longitude,
2                 double elevation) {
3     this.time = time;
4     this.latitude = latitude;
5     this.longitude = longitude;
6     this.elevation = elevation;
7 }
```

Listing 3.3: Solution to a simple Java exercise.

```
1 public GPSPoint(int time, double latitude, double longitude,
2                 double elevation) {
3     // TODO - Implement Constructor
4     throw new UnsupportedOperationException("TODO.constructor("
5         GPSPoint");");
6 }
```

Listing 3.4: Start code for the above exercise.

After analyzing multiple exercises from DAT100 and INF102 assignments, five main strategies were identified for transforming a solution to start code. Most of these were exercises that only spanned a single method in a Java class. The strategies were:

- Remove everything, both the method definition and body.
- Remove only the body but keep the method definition.
- Replace the body of the method with some other code.
- Remove only the solution while keeping the method definition and other program statements in the method body that was not a part of the solution.
- The same as above but the solution is replaced with some other code instead.

Exercises within the same assignment often used different strategies for the transformations, meaning there is a need for specifying which transformation strategy, or transformation rule, to use on an exercise-by-exercise basis. These options, and the other meta-information discussed above, must somehow be injected or connected to the solution project. Including this meta-information in the same solution project that other teachers, assistants, and possibly students get access to would most likely cause confusion and annoyance. This is because the meta-information is entirely foreign to them, and it also makes the solution source code bloated and harder to understand. Another project is therefore included in the design, which contains the solution project with the additional meta-information needed for FREJA. This is often referred to as the *source project* or *annotated solution project* throughout the thesis. The regular solution project is generated from the source project by simply removing all the meta-information. Figure 3.3 shows a simple model that demonstrates the high-level design of the framework. The arrows shows which artefacts gets generated. This design embodies the SSOT architecture by only needing one place for development. This central source of information acts as a prescriptive model for generating the other artefacts.

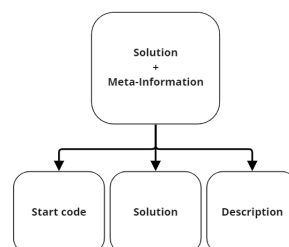


Figure 3.3: A model of the artefacts for a FREJA assignment. The generated artefacts have an incoming arrow.

3.4 Defining Meta-Information

The above design is based on the idea of defining meta-information related to the source code, so that assignment artefacts can be actively connected. However, this design raises two difficult questions. Where is this meta-information defined? And equally important, how is it defined? The meta-information must target several pieces of code, and define their meaning in the context of a programming assignment, as well as specifying how they are transformed. As for the first question, there are two central strategies. Either the meta-information is directly injected in the source code, or it is defined in a separate file or project.

Both of these strategies poses great difficulties. Defining the meta-information outside of the source code presents the task of targeting certain fragments of code. For example, suppose someone wants to specify in the meta-information that a certain method in a class constitutes as an exercise in an assignment. One could target this method by, e.g., specifying the filename in conjunction with the start and end line of the method, or perhaps use a combination of class and method name to target it. The immediate pitfall of these solutions is that they fall apart when the source code is changed, either by renaming things, or by changes that affect line positions. Keeping track of such changes would cause for a complicated implementation. Besides, a DSL would likely need to be created so that the meta-information has a clear syntax that can be interpreted. This inconsistency issue between isolated elements is reminiscent of the problems that inspired this thesis, so basing a design on that is undesirable.

On the other hand, defining the meta-information directly inside the source code makes targeting specific sections of code much easier. Though, an entirely different issue arises with this strategy. How can meta-information be injected in the source code and still conform to the syntax of the programming language? One possible solution is to make use of comments, which can be placed almost anywhere in the code in any programming language. However, the laid-back nature of comments makes it difficult to enforce a clear syntax that can warn and help the user when syntactical errors are made. Another possibility is to define a new DSL with clear definitions of programming assignment meta-information, both syntactically and semantically, while also defining concepts in the DSL for code blocks of the underlying programming language of the assignment. The design for the generic language agnostic prototype is based on this idea. A later section goes into more detail about the benefits and challenges for this design. One other solution is to make use of the tools given by the programming language that the assignment is written in, especially any metaprogramming functionality. The meta-information syntax would then be defined as a regular library of the language, meaning it can easily be inserted into the source code by simply importing the library. This is the core idea behind the design of FREJA. No matter which of these strategies are used, there needs to be an accompanying parser that can process the meta-information, and an interpreter that understands the meaning of it. Once the source code is completely processed, the necessary code transformations can take place. Finally, the generation is initiated to produce the desired artefacts. The next chapter goes into detail on the structure and functionality of the implemented meta-information designs, while the *Architecture* chapter puts more emphasis on the later steps concerning parsing, transformations, and generation.

Chapter 4

Implementation

This chapter covers implementation details of both FREJA and the programming language agnostic prototype. Detailed explanations are given on how both frameworks function, which include descriptions and brief explanations of the underlying technologies that are used. The majority of this chapter revolve around the parts of the frameworks that the user interacts with, while the next chapter focuses more on how the source code is structured internally.

4.1 FREJA

FREJA is implemented in Java and is only available for programming assignments also written in Java. The end-user uses this framework by writing Java code with specific semantics inside the assignment solution project. The assignment solution project is the only artefact that needs to be manually developed, whilst other artefacts such as the start code and assignment description are generated from that.

4.1.1 Annotations

Assignment meta-information is inserted into the code by making use of Java's annotation API¹. The API is used to define custom annotations that can be inserted into to the solution code to mark code fragments with specific semantics related to programming assignments. Java annotations are only metadata and is not part of the program logic itself. There are some predefined annotations in the Java language that does specific things, but creating custom-made annotations must also be accompanied by some processor that can parse and use the metadata in a meaningful way. If not, the annotations will have no effect.

A Java annotation is defined in its own file where the name of the annotation is specified in addition to potential annotation elements and extra metadata about the annotation itself. Annotation elements are essentially variables to store some type of data inside the annotation. They can be required, or optional by having a default value. There is also a restriction on where annotations can be placed in a

¹<https://docs.oracle.com/javase/tutorial/java/annotations/index.html>

program. By default, annotations can be applied on nearly all declarations, i.e., declarations of classes, methods, variables, parameters, and packages. However, the annotation definition can specify a more limited application space for the annotation. Other annotations, called *meta-annotations*, are used to define such metadata about the annotation itself.

Listing 4.1 shows the definition of an annotation that has three elements: `id`, `transformOption` and `replacementId`. The last one is optional, since the empty string is specified as the default value. The `@Target` meta-annotation limits the scope of the annotation to only be applicable to field variables, constructors and methods. The part that precedes the annotation name, `@interface`, is a keyword in the Java language that is used to define that the type for this Java class should be an annotation. This is similar to how other keywords such as `enum`, `class`, `interface` are used to create classes for those respective types. The reason the word *interface* is used in the keyword for an annotation class is that annotations are a form of Java interface, which becomes more apparent by noticing the similarities between their class definitions.

```

1 @Target({ElementType.FIELD, ElementType.CONSTRUCTOR, ElementType.
   METHOD})
2 public @interface Exercise {
3
4     int [] id();
5     TransformOption transformOption();
6     String replacementId() default "";
7 }

```

Listing 4.1: The definition of the `Exercise` annotation.

There are currently five different custom annotations in FREJA that can be used. The `Exercise` annotation shown in Listing 4.1 is used on code structures that should be implemented for an exercise in an assignment, such as on field variables or methods. The annotation includes some options to specify its identifier and how it should be transformed when generating the start code. Listing 4.2 shows an example of using the `Exercise` annotation on the constructor exercise from earlier, where the `id` annotation element specifies that it is the first sub-exercise for exercise 1. Using an integer array for specifying the numbering of an exercise allows for enforcing a more syntactic structure of the hierarchical numbering, compared to using something like a string. Different annotations can have the same number, or identifier rather. For example, one exercise might be to implement several field variables, which requires a separate annotation for each of them. These can be grouped as several *tasks* for one exercise by using the same identifier for each annotation.

```

1 @Exercise(id = {1,1}, transformOption = TransformOption.
   REPLACE_BODY, replacementId = "1")
2 public GPSPoint(int time, double latitude, double longitude, double
   elevation) {
3     this.time = time;
4     this.latitude = latitude;
5     this.longitude = longitude;
6     this.elevation = elevation;
7 }

```

Listing 4.2: An example of the `Exercise` annotation being used.

`TransformOption` is an enum used to pick between the transformation strategies discussed in the previous chapter. As a reminder, these transformation strategies define how the solution should be transformed into the start code. In this scenario the body of the constructor should be replaced with some other code, specifically some replacement code with 1 as its identifier. Replacement code is marked with a `ReplacementCode` annotation, as seen in Listing 4.3. Here, the entire body of the constructor from Listing 4.2 would be swapped out with the body of the `replacement` method, since the `replacementId` in the exercise annotation has the same value as the `id` element in the `ReplacementCode` annotation. The reason these annotation elements are strings and not some other type like integers, is to give the user the option to create their own identification system for replacement code. Multiple exercise annotations can target the same replacement code, allowing for better code reuseability.

```

1  @Remove
2  @ReplacementCode(id = "1")
3  public static void replacement(){
4      // TODO - Implement Constructor
5      throw new UnsupportedOperationException("TODO.constructor("
6      GPSPoint"));
  }

```

Listing 4.3: An example of the `ReplacementCode` annotation being used

However, the actual definitions of these code replacements does not need to be included in the start code after the replacement have taken place. There might be confusion from students if this unused code is included, so it is better if these definitions are removed entirely. The third annotation, `Remove`, can be used on any component in the source code. That component will be removed completely from both the start code and the generated solution project as well. Using the annotation at the class level will remove the entire file instead. The `Remove` annotation in Listing 4.3 would allow the replacement to take place first, before removing the entire `replacement` method, including all of its annotations.

The last two annotations are used a little differently. After studying solutions and start code from many example exercises, a good deal of them had some statements from the solution code in the start code also. These statements were not necessarily a part of the solution, but often left in the start code to either help students, or for the sake of avoiding syntax/compilation problems. Listing 4.4 is the solution code to an exercise from the DAT100 assignment.

```

1  public boolean insert(String time, String latitude, String
2      longitude, String elevation) {
3      GPSPoint gpspoint;
4
5      // TODO - START
6      gpspoint = GPSPDataConverter.convert(time, latitude, longitude,
7      elevation);
8      boolean inserted = insertGPS(gpspoint);
9      // TODO - END
10     return inserted;
  }

```

Listing 4.4: The solution to an exercise that highlights which statements are part of the exercise solution through comments.

The start and end comments were left in the solution code, and showcases a scenario where some statements in the method body are not part of the exercise solution. Listing 4.5 shows the start code for the very same exercise, which has only the solution replaced, while other statements are kept. Removing or replacing the entire method body is not a suitable solution for this scenario.

```

1  public boolean insert(String time, String latitude, String
2      longitude, String elevation) {
3      GPSPoint gpspoint;
4
5      // TODO - START
6      throw new UnsupportedOperationException(TODO.method());
7      // TODO - END
    }

```

Listing 4.5: The start code to the above exercise with only the solution removed.

To make transformations like these a possibility, there needs to be a way to mark only some statements as being part of the solution to an exercise. Unfortunately, the Java language does not provide any obvious functionality for solving this. Again, using comments as seen in the examples above is a possibility, but this limits the ability to enforce a clear syntax, as previously discussed. Another way is to mark the solution statements within curly brackets and labeling that code block as the solution. However, there is no possibility of enforcing a certain name on a labeled code block, and this also encounters problems with scoped local variables. The implementation of FREJA uses annotations to mark the start and end of a solution. Due to the previously mentioned restrictions that Java's annotation API enforces on the placement of annotations, they must be applied somewhat differently to be flexible. An object variable of the `SolutionStart` annotation is declared before the first statement of a solution, while a `SolutionEnd` variable is declared after the last statement to mark specific statements as a solution within a method or constructor body. An example of this can be seen in Listing 4.6, at line 5 and 8.

```

1  @Exercise(id = {2,2,3}, transformOption = TransformOption.
2      REPLACE_SOLUTION, replacementId = "1")
3  public boolean insert(String time, String latitude, String
4      longitude, String elevation) {
5      GPSPoint gpspoint;
6
7      SolutionStart start;
8      gpspoint = GPSPDataConverter.convert(time, latitude, longitude,
9          elevation);
10     boolean inserted = insertGPS(gpspoint);
11     SolutionEnd end;
12
13     return inserted;
14 }
15
16 @Remove
17 @ReplacementCode(id = "1")
18 public void replacement(){
19     throw new UnsupportedOperationException(TODO.method());
20 }

```

Listing 4.6: The solution wrapped in annotation variables

This is the solution code from Listing 4.4, where the start/end comments are swapped out for annotations variable declarations. At the bottom is the replacement code that the solution should be swapped out with. The marked solution, combined with the details of the `Exercise` annotation, will produce the start code shown in Listing 4.7.

```
1  public boolean insert(String time, String latitude, String
2     longitude, String elevation) {
3     GPSPoint gpspoint;
4
5     // TODO - START
6     throw new UnsupportedOperationException(TODO.method());
7     // TODO - END
8
9     return inserted;
}
```

Listing 4.7: The generated start code with a syntax error.

This is not identical to the original start code since it includes the return statement at the end. The reason this statement is removed from the original start code is because it will create a compilation warning. The return statement follows immediately after a throw statement and will create an *Unreachable Statement*² error. This can be easily fixed by putting the `SolutionEnd` variable after the return statement, however, this will also encounter the same unreachable statement problem, only this time it will be in the annotated solution code instead. To account for this, the `SolutionEnd` variable declaration can be omitted if the last statement in the solution is a return statement or a throw statement. The curly bracket that encloses the method body will be used instead to mark the end of the solution. Doing this for the above exercise would result in the correct start code, shown earlier in Listing 4.5.

The naming of the annotation variables does not matter, only that objects of these specific annotations are declared. Of course, the solution markers could be defined as regular classes instead, but defining them as annotations makes the declarations stand out more, since most IDEs will highlight them with a different color. This makes it clearer that these statements have a different function than the rest of the code. Currently, it is only possible to mark one block of statements as a solution per `Exercise` annotation.

4.1.2 Parsing

The very first prototype of FREJA was based on making use of the Java annotation processor API³. As mentioned earlier, this is an officially supported API for performing metaprogramming in Java. A processor can easily keep track of what constructs, such as methods, field variables, and classes, are annotated with the annotations defined above. The source code can then be transformed during compilation. The difficulty with this approach is the bottom-up way of processing the annotations. Other parts of the source code that is not annotated still need to be processed somehow, since this code should also be included

²<https://www.geeksforgeeks.org/unreachable-code-error-in-java>

³<https://docs.oracle.com/javase/8/docs/api/javax/annotation/processing/Processor.html>

when generating the other projects. The processor API only allows navigating to non-annotated code from some other code that is annotated. Classes that do not contain any annotations cannot be processed because of this. In addition, the API for navigating the processed source code is also limited and difficult to use. The processor API is very useful in scenarios where it is important that the processing happens during compilation, for example before initializing a long-running application. However, this is not the case for FREJA, since the underlying assignment program that is processed does not get executed or launched after processing and transforming it. In reality, the end-product of FREJA are static text files, so it does not matter if the transformations are done under pre-processing, compilation, or during run-time.

Because of these problems and details, a pivot was made to use a dedicated Java parsing and generator library. Behind the scenes, FREJA relies on `JavaParser`⁴ for parsing, transforming and generating Java source code. The program starts by looking for the source folder (i.e., a folder literally named `src` or `source`, which is standard in Java projects) and starts parsing all the Java files in that folder. An AST is created for every Java file. The ASTs are stored as Java objects that also includes additional meta-information, such as filename, storage information and more.

All the ASTs are then traversed to create a new object hierarchy, that more closely resembles the structure of a programming assignment. The annotations defined in this framework are searched for in the AST traversal, and are used to create objects in the assignment structure. For example, a method annotated with `@Exercise` is used to create an `Exercise` object. A simplified UML class diagram can be seen in Fig. 4.1 on the next page, showcasing the structure of the assignment class hierarchy. The `Assignment` object is simultaneously a *descriptive model*, in the sense that it describes the structure of an assignment in a more abstract way, and a *prescriptive model*, since it is also used as a blueprint for generating the rest of the assignment artefacts.

The `Node` class is part of the `JavaParser` library, and represents a node in an AST. There exists a node class for every terminal and non-terminal in the Java grammar, but the diagram is kept simple for the sake of brevity. The `Task` class is what corresponds to an `Exercise` annotation, since multiple nodes can be annotated with the same exercise identifier. These task objects have two important methods, `createStartCode` for creating a new node for the start code, and `createSolutionCode` that creates a new node for the solution code without annotations. By following the instructions specified in the `transformOption` element of the `Exercise` annotation, two new lists of ASTs are created in the `Assignment` object, containing the newly created nodes. One list is for the start code project, and one list for the solution code project.

4.1.3 Execution Phase

The popular Java project management and build tool, Maven⁵, is required to execute the program since the framework is implemented as a Maven plugin. There are several reasons for this, but most important is that it makes the

⁴<https://javaparser.org>

⁵<https://maven.apache.org>

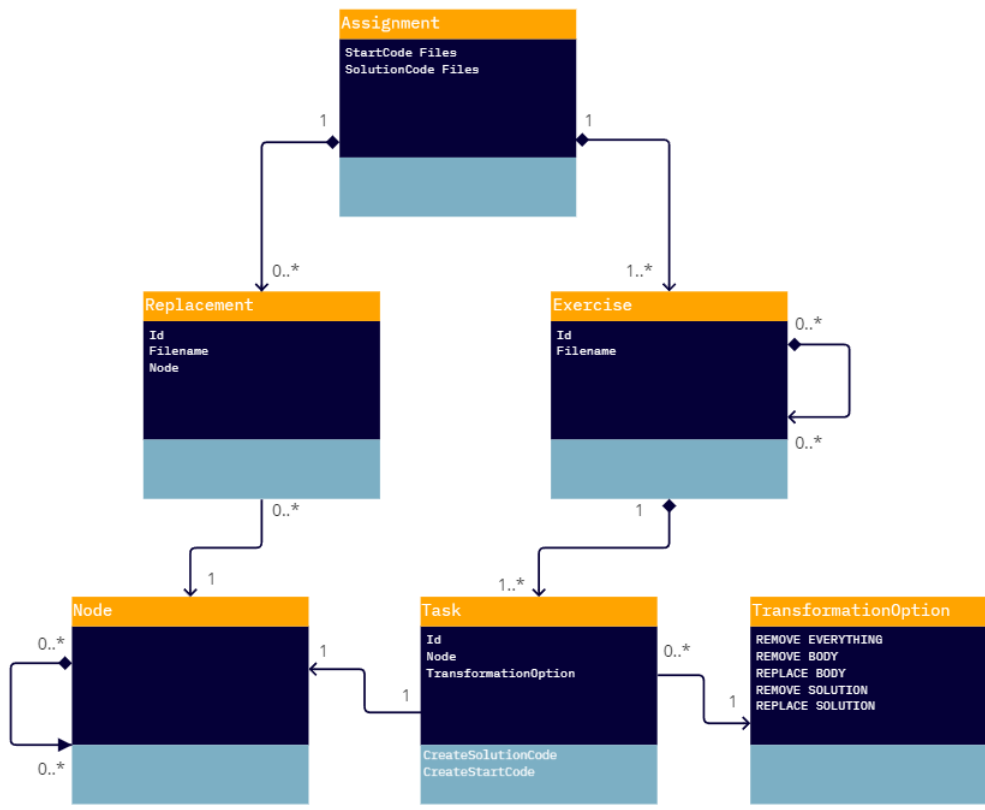


Figure 4.1: A simplified view of the assignment object hierarchy.

framework easy to distribute and install while still being IDE independent. Another benefit is that the framework can be executed within the same project as the one the assignment is developed in.

After the initial ASTs are constructed and used for creating the assignment objects, the program goes through every folder and file in the source project and copies them over into two new projects. The target folder is cleared before generating the projects due to the likelihood of executing the program several times, eliminating the need for manually deleting files to avoid duplication. When copying a Java file, the program will instead look for the associated AST that was created in the `Assignment` object, and then use `JavaParser` to create a Java file from an AST object.

The only information that needs to be provided to be able to execute the program is the location of the target folder to generate the projects in. There are options that can be configured in the Maven Project Object Model (POM) file, which includes the required target path folder. Executing the program without any additional configuration and annotations will simply copy all the files in the source project and paste them over in the generated projects. However, the POM file also includes the ability to specify files or folders that should be ignored, meaning that they will not be copied over into the generated projects. This option uses glob patterns⁶ to specify filenames to make it more standardized and user friendly, since most users will likely already be familiar with such patterns from other types of configuration files, such as `gitignore`.

4.1.4 Exercise Descriptions

During the generation of the start code project, an additional folder is created for exercise descriptions. A separate file is created for each exercise, but sub-exercises are grouped in the same file. The descriptions are structured based on the numbering (identifier) in the exercise annotations. The information is generated entirely from the source code by using the meta-information that `JavaParser` provides for each annotated node. Descriptions are written in the markup language *AsciiDoc*⁷ because of the rich functionality it provides. The language requires an `Asciidoctor` processor⁸ to process `AsciiDoc` files. Most importantly, the language provides the ability to define *attributes*⁹ that can be referenced later in a document. An attribute is a key-value pair that is defined in a document header. Referencing the key later in a document will substitute the key with the value in the definition, making it possible to reference the source code, and also automatically update the reference if the source code changes. To do this, the attribute key must be linked to some piece of the source code while the value stores the actual information that should be referenced. The attribute key must remain unchanged, but the attribute value updates automatically whenever the source code updates. There is no need to manually update the substitutions since the attribute key name does not get changed. This is useful in many scenarios, especially in the case of exercise descriptions.

⁶[https://en.wikipedia.org/wiki/Glob_\(programming\)](https://en.wikipedia.org/wiki/Glob_(programming))

⁷<https://asciidoc.org>

⁸<https://asciidoctor.org>

⁹[https://docs.asciidoctor.org/asciidoc/latest/syntax-quick-reference/
#attributes-and-substitutions](https://docs.asciidoctor.org/asciidoc/latest/syntax-quick-reference/#attributes-and-substitutions)

For example, many exercise descriptions reference the filename of the start code for the exercise. In Listing 4.8, the exercise description would generate the attribute, `:Exercise1_1_FileName: pass:normal[+GPSPoint.java+]`, for the filename. The name of the key for the attribute is `Exercise1_1_FileName`, and the value is `GPSPoint.java`¹⁰. A reference to that attribute in the description must wrap the key in curly brackets, as seen in Listing 4.9. This would result in the substitution shown in Listing 4.10. If the filename then changes afterwards, simply executing FREJA again will update the AsciiDoc description file by changing the value of the attribute. There is no need to manually update the description since both the key and the reference to the key remains unchanged, but the substitution will use the new value instead.

```

1 public class GPSPoint {
2
3     @Exercise(id = {1,1}, transformOption = TransformOption.
4         REPLACE_BODY, replacementId = "1")
5     public GPSPoint(int time, double latitude, double longitude,
6         double elevation) {
7         this.time = time;
8         this.latitude = latitude;
9         this.longitude = longitude;
10        this.elevation = elevation;
11    }

```

Listing 4.8: A simple exercise.

```

1 The start code to exercise 1 can be found in {Exercise1_1_FileName}

```

Listing 4.9: An exercise description with an attribute reference.

```

1 The start code to exercise 1 can be found in GPSPoint.java

```

Listing 4.10: An exercise description with a substitution for the attribute reference.

A template description is generated for each exercise and is structured as a bare-bone explanation of what to implement in the exercise. The purpose of this template is to be a starting point for the full description, and to give the user an example of how to use attributes and other common syntax of AsciiDoc. Executing FREJA several times will regenerate the template description and overwrite any manual changes that may have been done to the description, but this behavior can be changed through an option in the POM configuration. Either way, all attribute values will always be updated to reflect any changes in the source code. AsciiDoc provides the opportunity to transform the text to either PDF or HTML after the writing is finished. Publishing an AsciiDoc file directly to GitHub is also possible, since the website has an embedded AsciiDoc processor.

Going back to the subject of bidirectional transformations and asymmetrical lenses, the implementation details surrounding assignment descriptions does somewhat break the definition of what a normal *get* function is. Instead of the view, i.e., the generated description and other generated artefacts, being defined

¹⁰The other parts of the value definition is just to make the substitution use a monospaced font.

exclusively by the source, it is also defined by the previous view, kind of like a combination of *get* and *put*. The reason for this is to make it possible to preserve any manual changes made to the description. An earlier implementation of FREJA attempted to avoid this by having a pure *get* function instead. This implementation was inspired by Javadoc comments, and revolved around writing entire exercise descriptions in a string variable annotation element. The string variables were used to generate dedicated AsciiDoc description files, the same way it does now. The important detail is that any manual changes to the description would be done in the source project instead, ensuring the source completely defines the view. This idea was eventually scrapped due to its impracticality. Several exercise descriptions that were analyzed contained images, but writing the entire description through string variables did not provide the ability to do this. Since the end product was still AsciiDoc files, the description must still be written in the AsciiDoc language. However, this had to be done without the help of a plugin or editor due to them being written as regular string variables. Moreover, including the entire description in the source code also cluttered the source project immensely.

4.2 Programming Language Agnostic Framework

For a framework that attempts to target every programming language, there obviously cannot be a reliance on implementation details of a specific language, such as relying on annotations in Java. Instead, commonalities must be extracted from all programming languages to be able to develop a framework that can target arbitrary source code. One area that most programming languages have in common is their textual representation. This representation is used for developers to easily write and read their source code as they create it. Often, this is then compiled into a binary representation that computers can understand the meaning of. One step of the compilation is to create an AST of the source code, which is another representation that more clearly shows the syntactic structure of the code, in terms of the grammar of the programming language. After seeing the usefulness of the AST representation for FREJA, the initial prototype of a language agnostic framework was founded on the idea of working directly with ASTs.

As mentioned earlier, the design of the language agnostic framework is centered around a DSL that provides the ability to both write source code of any language, and define meta-information for programming assignments. The idea is for the DSL to allow injecting this meta-information inside the AST. The user would develop the solution for an assignment as normal, and then use the DSL to include meta-information in the syntax tree. This tree would then be processed and used to generate the other assignment artefacts. JetBrains MPS¹¹ is a language workbench that is used to develop the DSL. MPS does not use regular text files to store or edit code. Instead, the code is also represented and stored as an AST. MPS provides a projectional editor where the editor(s) of the language can be defined in terms of how they should work and look.

¹¹<https://www.jetbrains.com/mps>

4.2.1 First Prototype

The first prototype was based on a long transformation pipeline which can be seen in Fig. 4.2. The starting point on the left contains the source code of the complete solution of an assignment. Each arrow is a transformation from one representation of the source code to another, and is numbered to make it easier to discuss each transformation. The outer boxes show which language or format the representation and transformation are written in. *Source language* is the language that the programming assignment is written in.

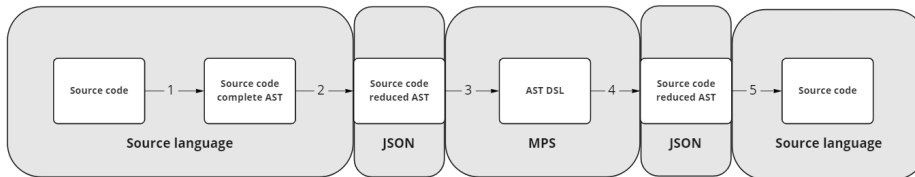


Figure 4.2: The transformation pipeline from source code to the DSL, and back.

Transformation 1 is the most complex, but most languages today have a dedicated library or third party tool to handle this transformation step without writing it from scratch, such as the aforementioned JavaParser, or Python’s official ast module¹². The result of this transformation is an object in the respective source language, that represents an AST for a programming file in the same language. However, this object is dependent on specific details for each programming language. Specific details like these must be abstracted away so that the AST can be transformed into more generic components of the DSL. The AST object only exist in memory in the source language for a brief moment, and obviously cannot be imported directly into MPS. Thus, a second transformation is needed to convert the AST object into a representation that can be used irrespective of the source language, such as JSON. Transformation 3 converts the JSON file to corresponding DSL components in MPS. It is after this transformation that the AST gets edited and injected with meta-information. Once that is done, a fourth transformation generates a new JSON file. The AST in this file is generated based on transformation rules and meta-information included from the previous step, similar to how annotations and their information are used to specify transformation details in FREJA. A fifth and final transformation converts the new JSON file back to code in the original source language. Creating a JSON schema for the JSON files ensures that transformation 3 and 4 only need to be written once. However, transformation 2 and 5 still need a specific implementation for each programming language.

As mentioned, MPS provides the ability to define how the editor of the language should look like, called the projectional editor. Although it may have initially sounded good on paper, it became immediately clear that displaying and working with the AST directly would be futile. Firstly, the people that create programming assignments may not be familiar with ASTs, either in general, or for a specific language. In this scenario, they would need to have knowledge about ASTs for the source language, while also learning the generic ASTs

¹²<https://docs.python.org/3/library/ast.html>

within the DSL. Secondly, an AST is verbose and becomes quite large, even for one simple line of code, as seen in Listing 4.11, making it very hard to get an overview of the actual code. Others have also encountered similar difficulties trying to transform ASTs directly, and have instead opted for a more user-friendly graphical interface for creating transformations [44].

```

1 <statement type="ExpressionStmt">
2   <expression type="AssignExpr" operator="ASSIGN">
3     <target type="NameExpr">
4       <name type="SimpleName" identifier="a" />
5     </target>
6     <value type="BinaryExpr" operator="PLUS">
7       <left type="NameExpr">
8         <name type="SimpleName" identifier="x" />
9       </left>
10      <right type="IntegerLiteralExpr" value="1" />
11    </value>
12  </expression>
13 </statement>

```

Listing 4.11: An XML AST representation of the Java statement `a = x + 1;`

The better option seemed to be displaying the source code instead, since it is more concise, and the author of a programming assignment is already very familiar with this representation. Then they could select different parts of the source code and mark them as specific assignment components, such as the solution to an exercise. Remember that the code in MPS is always structured as an AST, even though the representation in the editor might be different. Instead of editing the AST directly, a selection could be done on the textual representation of the code. Along with some clever implementation, it is also possible to infer what nodes in the AST the selection corresponds to. MPS also makes it possible to have custom code or programs that can be executed through a button click, keyboard combination, or popup menu. This program code could, for example, target the current selection from the editor, and modify the underlying nodes in the AST with assignment meta-information. Figure 4.3 shows an example of how this could look like.

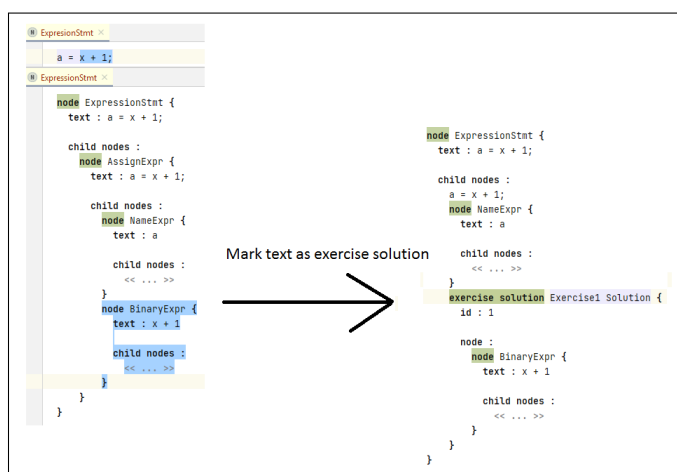


Figure 4.3: An example of how the first prototype could look like.

The top left part shows the editor that user interacts with, which is just the textual representation of the code. Below is the underlying representation of that code in the DSL, i.e., the AST of the code as concepts within the DSL. The figure shows that selecting the code in editor also selects the corresponding nodes in the AST. The idea is that this selection could be marked as an exercise solution for example, either through a pop-up menu or some keyboard combination, and the underlying AST would be transformed into the tree seen at the right side of the figure.

The big hurdle with this approach though, is that displaying the textual representation of the source code is easier said than done at this point in the transformation pipeline, even with all the information from the AST. For example, the semicolon from the program statement is not explicitly represented in the AST in Listing 4.11, and the plus sign is turned into a specific attribute instead of a character. Unparsing an AST would be trivial with the tools that were used to create the original AST, but they would not work at this point since the AST object is already transformed into the DSL.

The mentioned tools for creating ASTs for Java and Python programs includes information about each node in the tree concerning their start and end positions in the source file, i.e., the line number and column offset that each node starts and ends on. This information can also be included for each node in the JSON file after the second transformation in the pipeline. If the entire source code was added as a string in the JSON file as well, then it would be easy to display the source code as regular text in MPS. It would also be possible to figure out which nodes in the AST were selected based on the start and end position of the selection in the textual representation of the source code. Including the concrete syntax in the JSON file does also mean that the fifth transformation would essentially be just writing text to a file, instead of a complicated process of reconstructing an AST object in the source language that could eventually be unparsed.

At this point though, the original AST is not used to display the source code, nor is it used in the final transformation to generate new source code. The textual representation of the source code is what is presented, used for editing, and for generating the final artefacts. Updating the AST based on the edited textual source code would be an unneeded complexity that do not really matter in the end. This led to the initial prototype being abandoned, while focus shifted towards a revised implementation with a much simpler approach.

4.2.2 Second Prototype

Displaying the textual representation of the source code still felt like the natural choice going forward. Not needing an AST of the source code simplifies the process tremendously. All that is needed is some sort of reader that can parse text files and construct concepts in the DSL. This is all possible to do within MPS, no matter the programming language of the text file that is parsed. When selecting parts of the source code in MPS, as described earlier, there still need to be a concept in the DSL that has been selected also. This was supposed to be nodes in the AST previously. A line of code felt like an appropriate substitute for representing a structural component of a programming file, but

there is an argument to be made for having each individual character to be a separate component for more fine-grained control over the selection process. This seemed unnecessary for the purpose of this project, since selecting code related to a programming assignment would rarely need the ability to select only some part of a line. The updated transformation pipeline can be seen in Fig. 4.4

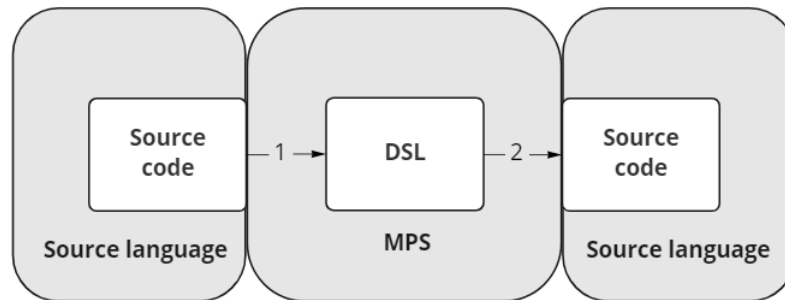


Figure 4.4: The transformation pipeline for the second prototype.

A language in MPS consists of structural components called *concepts*. These concepts can have properties, inherit other concepts, have child concepts or reference other concepts. Only root concepts can be created as a file in the language, but other concepts can be a part of a root concept. In the language for the prototype, one root concept is `File`, a concept that represents a programming file. The only properties of `File` are a string to represent the name of a file, and another string that represents the extension of the file (e.g., `.java` or `.py`). A `File` has a list of `IFileComponent` as its child concepts. `IFileComponent` is an interface concept that is an abstract representation of a component in a programming (assignment) file. Inheriting concepts of this interface can be as simple as a line of code, or more complex concepts such as an exercise solution.

Programming files containing exercise solutions are imported to the DSL with a button click after specifying the file path. This button click runs a small program within the DSL that is written in MPS's counterpart to Java, called `BaseLanguage`¹³. The program reads the imported file line by line and generates a new `File` instance with only lines of text as its file components. A new `LineOfText` concept instance is created for each line of code in the original file.

There are also different aspects of a language in MPS other than the structure, such as editors, constraints, type rules, generators and more, which are all based on the concepts of a language. The `File` editor simply shows the file name and extension at the top, and then lists all of its child concepts, displaying them using their own editor definitions. A `LineOfText` just displays the text it is holding in the editor, meaning that a newly generated `File` is almost displayed identically as in the IDE it was created in.

As previously mentioned, there may be one or several lines of code that have a conceptual meaning within an assignment, such as a solution to an exercise.

¹³<https://www.jetbrains.com/help/mps/base-language.html>

Annotation variables were used in FREJA to mark where a solution started and where it ended. Similarly, concepts in the source code can be marked by inserting temporary file components at the start and end lines of the concept. Keyboard shortcuts are used to easily insert these marking components. For example, a `SolutionStart` concept can be inserted using `Alt + Enter` at the first line of solution, and the end can be marked by using `Ctrl + Alt + Enter` at the last line of the solution. Alternatively, these markings can be inserted with the mouse using a right-click popup menu. Figure 4.5 and 4.6 shows the steps to mark a code fragment as an exercise solution with the DSL.

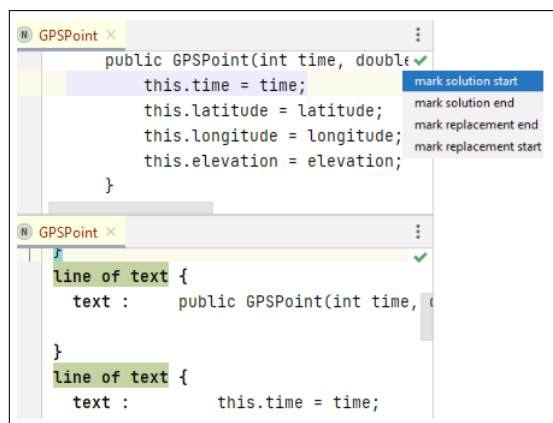


Figure 4.5: The starting line of a solution being marked.

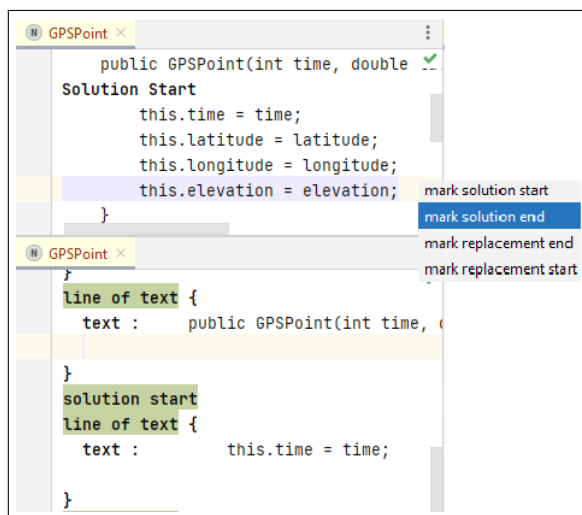


Figure 4.6: The end line of a solution being marked.

Each figure shows the editor view at the top, which uses the editor definition for each concept to determine how to display them. Underneath is the reflective editor¹⁴, which ignores the editor definitions and uses a default tree-like view of the

¹⁴<https://www.jetbrains.com/help/mps/finding-your-way-out.html#reflectiveeditor>

file. This view explicitly shows the structure of the file, and highlights the concept type of a node in green, such as `LineOfText` and `SolutionStart`. Figure 4.5 shows the custom-made popup menu option that inserts a `SolutionStart` marker at the selected line, which is present in the code in Fig. 4.6. After the end is marked, a `Solution` concept is automatically created and inserted into the file, replacing the marking components and the contained lines, as seen in Fig. 4.7

```

public GPSPoint(int time, double latitude, double longitude, double elevation) {
    Solution for task: <no taskNumber>
    Solution should be: <no transformOption>
    this.time = time;
    this.latitude = latitude;
    this.longitude = longitude;
    this.elevation = elevation;
}

}

line of text {
    text :      public GPSPoint(int time, double latitude, double longitude, double elevation) {
}

solution replacement : <no replacement> {
    task number : <no taskNumber>
    transform option : <no transformOption>

    lines of text :
        line of text {
            text :      this.time = time;
        }
        line of text {
            text :      this.latitude = latitude;
        }
        line of text {
            text :      this.longitude = longitude;
        }
        line of text {
            text :      this.elevation = elevation;
        }
}

```

Figure 4.7: The resulting `Solution` concept that was created.

This concept provides additional options that can be specified, such as what to do with the solution when generating the start code for the assignment. The `TransformOption` for the `Exercise` annotation in `FREJA` is analogous, which provides the ability to remove or replace the solution. Both options are

also possible in the DSL, although replacing the solution requires creating and referring to a `SolutionReplacement` concept. `SolutionReplacement` is another root concept, and it has an identifier and a list of text lines. It can be created in the same way a `Solution` concept is created, or by manually writing the code that should be used a replacement.

MPS has also many tools for creating custom generators for a language, including both M2M and M2T transformations. This prototype uses MPS's *TextGen*¹⁵ for creating M2T transformations. TextGen is its own language that comes bundled with MPS, allowing users to define M2T transformations for their own custom-made DSL. For the prototype language, the transformations convert all file components to lines of text according to the specified transformation options. The end-product is a start code file that can be compiled in the source language of the assignment. MPS has a built-in function that allows a user to select an instance of a root concept and preview what the generated file will look like, but only if a TextGen transformation is defined for that root concept. An example of this can be seen in Fig. 4.8, using the created `Solution` concept from earlier examples.

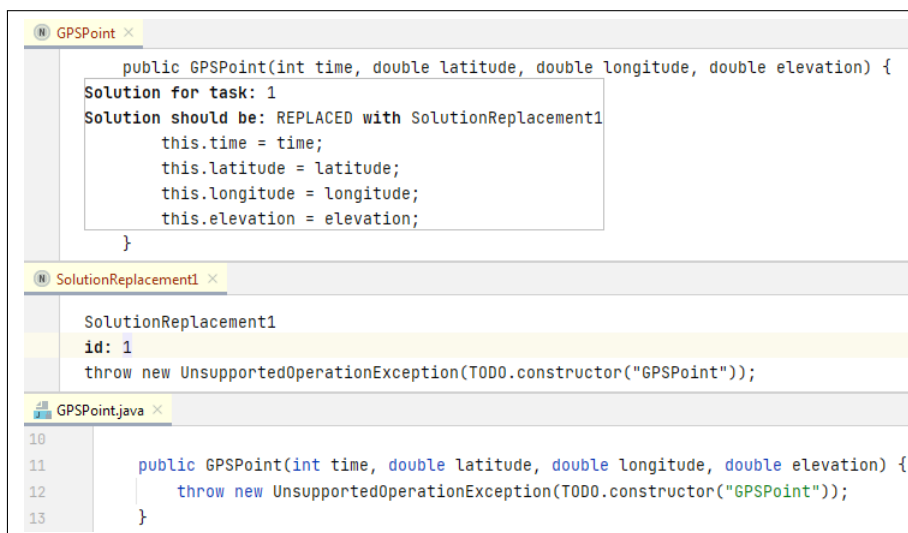


Figure 4.8: The top part shows a File editor with a solution to a Java exercise. Below that is the code that the solution should be replaced with. At the bottom is a preview of the generated output.

¹⁵<https://www.jetbrains.com/help/mps/textgen.html>

Chapter 5

Architecture

This chapter goes into detail about the implementation of FREJA and the programming language agnostic prototype. Compared to the last chapter, this chapter focuses more on what goes on behind the scenes, i.e., the architecture, structure, logic, and the design of the source code. In essence, these are details that an ordinary user does not need to know about to use the frameworks. However, this chapter is important to read and understand for any developer that is interested in continuing the work that is done in this thesis.

5.1 Architecture of FREJA

FREJA is built as a Maven project that is split into three separate modules. The `freja-test-example` module does not contain any implementation that affect the logic of the framework, but as its name suggests, is used as an assignment example for testing. Testing will be covered in more detail later in this chapter, but for now it is only important to know that the reason this module exists is to have a somewhat big example that asserts the correctness of the implementation of FREJA as a whole, from parsing files to generating files, through unit tests, and not just manual testing.

The content of the `freja-annotations` module was mostly covered in the previous chapter. Again, this module does not contain any logic, only the definitions of all the custom annotations for FREJA, and any enums that are used as a type for an annotation element. This module is on its own to maintain a modular design of the implementation, separating the parts of FREJA that the user interacts with, i.e., the annotations, from the program logic of the framework, i.e., the parsing, transformation, and generation of files. This was also beneficial during the development of FREJA when the `main` method was often used to do quick manual testing. The `main` method allows for executing the framework in the same way it can be executed as a Maven plugin by running the `mvn freja:generate` command. However, configuration details need to be specified programmatically within the `main` method, instead of through a POM file. Additionally, the source path of the assignment project that should be parsed must be specified since the framework is executed from another location, as opposed

to the Maven plugin that is always executed within the same project as the source path. Creating an example assignment project for manual testing is very quick because only the `freja-annotations` dependency is required, and not the FREJA Maven plugin. The dependency does not require the use of Maven or a POM file since the module can be imported as a JAR instead.

The last module, `freja-maven-plugin`, is by far the biggest in size, and contains all the logic and tests for FREJA. There are multiple packages in the module that groups related or similar classes within the same package, e.g., the `exceptions` package contains all the custom FREJA exceptions. The classes in this module will be explained mostly in the order of when they are introduced into the program flow when executing FREJA. The program starts by initializing an object of the `Configuration` class, filling it values that hold different FREJA configuration settings for a programming assignment. This includes information such as the source path of the annotated assignment project, target path of the output folder, and any files in the source project that should be ignored.

5.1.1 Maven

As mentioned earlier, these configuration values can be set directly in the `main` method, but much more work is needed to extract those values from the `pom.xml` file into the program memory. In short, this forces the creation of a custom Maven plugin¹. Official Maven dependencies, such as the `maven-plugin-api`, are added to the `pom.xml` file in the `freja-maven-plugin` module to be able to create a custom plugin. This POM file also has the `maven-plugin-plugin`² Maven plugin included in the build life cycle³, as seen in Listing 5.1. The prefix of the plugin goal (i.e., the `freja` part of the `mvn freja:generate` command) for the FREJA plugin is defined in the configuration settings at line 8.

```
1 <build>
2   <plugins>
3     <plugin>
4       <groupId>org.apache.maven.plugins</groupId>
5       <artifactId>maven-plugin-plugin</artifactId>
6       <version>3.6.4</version>
7       <configuration>
8         <goalPrefix>freja</goalPrefix>
9       </configuration>
10      <executions>
11        <execution>
12          <id>help-goal</id>
13          <goals>
14            <goal>helpmojo</goal>
15          </goals>
16        </execution>
17      </executions>
18    </plugin>
19  </plugins>
20 </build>
```

Listing 5.1: The build lifecycle in the FREJA Maven plugin `pom.xml` file.

¹This web-page goes into great detail about Maven plugin development: <https://maven.apache.org/plugin-developers/index.html>

²<https://maven.apache.org/plugin-tools/maven-plugin-plugin/>

³<https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>

`maven-plugin-plugin` is used to create a plugin descriptor for custom plugins by looking through the source code for Maven plain Old Java Object (MOJO)s. A MOJO is an executable goal in Maven, which is distributed through a plugin and can be defined in terms of a Java class. In the `maven` package, there is a class called `FrejaMojo`, which is the MOJO for this custom Maven plugin. It extends the `AbstractMojo` class and implements the `execute` method, which determines the behavior of the plugin goal. In this case, the method instantiates a `Configuration` object with configuration options from a POM file. Each option that can be set in the POM file for the FREJA Maven plugin has a corresponding field variable in the `FrejaMojo` class, as seen in Listing 5.2. These field variables are annotated with `@Parameter`, both to specify that they are configuration options for the MOJO, and to also define other properties of the option, such as its name. Listing 5.2 also shows that the class has a `Mojo` annotation, which defines the name of the goal (i.e., "generate"), and binds the goal to the compile phase of the build life cycle.

```
1 @Mojo(name = "generate", defaultPhase = LifecyclePhase.COMPILE)
2 public class FrejaMojo extends AbstractMojo {
3
4     @Parameter(property = "targetPath", required = true)
5     private String targetPath;
6
7     @Parameter(property = "ignore")
8     private List<String> ignore;
9
10    @Parameter(property = "keepOldDescriptions")
11    private boolean keepOldDescriptions;
12
13    ...
14 }
```

Listing 5.2: A snippet of the `FrejaMojo` class and its field variables.

5.1.2 Parsing

Whether its from the `main` method or from the `execute` method in the MOJO, the `Configuration` object is used to create a `Generator` object. Afterwards, the `generate` method of the `Generator` object is promptly executed. This method orchestrates the high-level flow of the program by first parsing the source files, transforming them, and finally generating the desired artefacts. The parsing is done by creating a `Parser` object and initiating it with the path of the source project. Except for finding the source folder for the Java files in the source project, this class hands over the responsibility of parsing to the `JavaParser` library. The result of a successful parsing is a list of `CompilationUnit` objects, one for each Java file. The `CompilationUnit` class extends the abstract `Node` class, and can be thought of as the root node in an AST. Every node in an AST is an instance of a class that is subclass of the `Node` class, either directly or indirectly. These objects have references to their parent and child nodes, and also have additional information about themselves, such as their range and concrete syntax. For anyone looking to continue development on FREJA, it is vital to get a good understanding of the node system from the `JavaParser` library first. The book *JavaParser: Visited*⁴ is free and a good place to start

⁴<https://leanpub.com/javaparservisited>

learning the JavaParser API, as well as the official documentation of the source code⁵.

5.1.3 Transforming

After the parsing is done, the `generate` method starts transforming the syntax trees. This is done by creating an `Assignment` object that represents a programming assignment in the context of FREJA. This object store information about the parts that composes a FREJA assignment, such as the exercises, replacement code, and the syntax trees from the source project. Figure 4.1 from the previous chapter shows the structure of the `Assignment` hierarchy. The reason for creating this assignment representation through Java objects is to get a better overview of the different parts of an assignment, while also providing a way to navigate between them easily. It also makes things easier in the generation phase afterwards, as well as making testing and debugging more straightforward.

The `Assignment` class and other assignment concepts can be found in the `concepts` package. These classes are only vessels of data, but the creation of them are complex tasks. The architecture of FREJA draws inspiration from the *separation of concerns*⁶ design principle and *single-responsibility principle*⁷ to alleviate some of the responsibility of the concept classes. The *builder pattern*⁸ is used for the creation of these classes, which dedicates a separate builder class for each concept class. The sole purpose of a builder class is just to create an object of another class. For example, the `Assignment` class has an `AssignmentBuilder` class that instantiates an `Assignment` object, as seen in Listing 5.3. The `build` method starts off by initiating other assignment concepts, such as replacements and exercises, by calling the methods `findReplacements` and `findExercises`, respectively. These methods will find nodes in the ASTs from the parsed files that are annotated with the corresponding FREJA annotation, and invoke the `build` method of those classes again. The creation of objects in the assignment hierarchy will be initiated recursively top-down from assignment to either tasks, replacements or solutions, but finished in the order of bottom-up.

```
1     public Assignment build() {
2         Assignment assignment = new Assignment();
3         parsedFiles = parser.getCompilationUnitCopies();
4         assignment.setParsedFiles(parsedFiles);
5         assignment.setReplacements(findReplacements());
6         assignment.setExercises(findExercises());
7         assignment.setStartCodeFiles(createStartCode());
8         assignment.setSolutionCodeFiles(createSolutionCode());
9         assignment.setFileNamesToRemove(fileNamesToRemove);
10        return assignment;
11    }
```

Listing 5.3: The `build` method of the `AssignmentBuilder` class.

⁵<https://www.javadoc.io/doc/com.github.javaparser/javaparser-core/latest/index.html>

⁶https://en.wikipedia.org/wiki/Separation_of_concerns

⁷https://en.wikipedia.org/wiki/Single-responsibility_principle

⁸https://en.wikipedia.org/wiki/Builder_pattern

It is important to understand the difference between a `Task` object and an `Exercise` object. Although a little confusing, a `Task` object is created for each `Exercise` annotation. This is due to the ability of having several tasks for an exercise, which can be done by annotating different code structures with `Exercise` annotations using the same value for the `id` element. Such tasks will be grouped together within the same `Exercise` object, but an `Exercise` object does not necessarily need to have a corresponding `Task` object. The reason for this is that `Exercise` objects are created to represent the full hierarchy of exercises and sub-exercises. A common occurrence is that an `Exercise` object is created to represent a root exercise with only a collection of sub-exercises, and no `Task` objects linked directly to the root exercise. However, the sub-exercises lowest in the hierarchy must always have at least one corresponding `Task` object.

After all the assignment parts are created, the `build` method from Listing 5.3 above, invokes two other methods: `createStartCode` and `createSolutionCode`, which will create two new lists of ASTs, one for each project that should be generated. Since there are several projects that are generated, the ASTs are cloned before transforming them to not mutate the original syntax trees, since that can cause unintended changes and hard-to-spot bugs. Both of these transformations start off the same way by removing all meta-information related to FREJA. This includes entire nodes of an AST that are annotated with `Remove`, and all FREJA annotations and import statements. This information is no longer needed since it is stored in the assignment concept objects. Afterwards, the transformations differ when the nodes in the AST that are encapsulated by an `Exercise` annotation, i.e., the `Task` objects, should be transformed. For the solution project, all nodes in the tree will be kept to maintain the exercise solution, except for any potential `SolutionStart` or `SolutionEnd` variable declarations. The start code project transforms the nodes according to the value of the `transformOption` in the `Exercise` annotation. Transformations for both the start code project and solution project are done using methods from the `TaskOperations` interface shown in Listing 5.4.

```
1 public interface TaskOperations {
2
3     BodyDeclaration<?> createSolutionCode(BodyDeclaration<?>
4         nodeToUpdate);
5     BodyDeclaration<?> createStartCode(BodyDeclaration<?>
6         nodeToUpdate);
7 }
```

Listing 5.4: The `TaskOperations` interface.

The `BodyDeclaration` type is from the `JavaParser` API, which is a common parent class for any other node types that can be annotated with the `Exercise` annotation, such as `MethodDeclaration` or `FieldDeclaration`. The methods take an old node as an argument and returns a new one with the appropriate transformations, which then replaces the old node in the AST. This interface is implemented by the abstract `Task` class, but it only implements the `createSolutionCode` method and not the `createStartCode` method. Instead, there is a different subclass of the `Task` class for each type of `TransformOption` that implements the method differently. This allows the `AssignmentBuilder`

class to conform to the *open-closed principle*⁹, since it does not need to modify the `createNewTaskNode` method in Listing 5.5 whenever a new `TransformOption` is introduced. A new subclass of the `Task` class is created instead to handle such an extension.

```

1     private BodyDeclaration<?> createNewTaskNode(TargetProject
2         targetProject, Task task, BodyDeclaration<?> newTaskNode) {
3         if(targetProject.equals(TargetProject.SOLUTION)){
4             newTaskNode = task.createSolutionCode(newTaskNode);
5         }else if (targetProject.equals(TargetProject.START_CODE)){
6             newTaskNode = task.createStartCode(newTaskNode);
7         }
8         removeAnnotationTypeFromNode(newTaskNode, EXERCISE_NAME);
9         return newTaskNode;
    }

```

Listing 5.5: A method for transforming a `Task` node.

A significant amount of work has gone into keeping the source code of FREJA as clean as possible. This is comprised of several practices, such as short method bodies and classes, explicit and meaningful naming, ordering of code in terms of program flow, and reducing the amount of responsibility a method or class has. This has led to the creation of several utility classes with static methods that removes responsibilities from other classes. For example, the `TaskBuilder` class needs to know the value of the `transformOption` in the `Exercise` annotation to know which type of task class it should create. However, the sole purpose of this class is to build `Task` objects, so the responsibility of finding this value is shifted to the `AnnotationUtils` class, which can find this value through the `getAnnotationMemberValue` method in Listing 5.6.

```

1     public static Expression getAnnotationMemberValue(
2         NodeWithAnnotations<?> node, String annotationName, String
3         memberName){
4         Optional<AnnotationExpr> annotation = node.
5             getAnnotationByName(annotationName);
6         if(annotation.isPresent()){
7             return getAnnotationMemberValueFromAnnotationExpr(
8                 annotation.get(), memberName);
9         }else{
10            throw new NodeException((Node) node,
11                String.format("Could not find annotation \"%s\"
12                    on the node:%n%s", annotationName, node));
13        }
14    }

```

Listing 5.6: A method for finding the value of a annotation element/member.

Listing 5.6 also serves as an example of the error-handling of FREJA. Normal Java error logs that show where in the call stack an error was thrown from is not very helpful in most cases for FREJA. It is often the case that the problem lies with a node from one of the ASTs from the parsed Java files. There can be two reasons for this: a developer misuses a method or some other code, or the end-user of FREJA have done a semantic mistake while using an annotation, e.g., using a `replacementId` in an `Exercise` annotation that does not exist in any `ReplacementCode` annotation. In both cases it is helpful to print out the

⁹https://en.wikipedia.org/wiki/Open-closed_principle

node and the location and file it was parsed from, which is exactly what the custom `NodeException` class does. An example of an error message for the aforementioned scenario can be seen in Fig. 5.1. Most other custom exceptions in the FREJA implementation is a subclass of this exception, enabling more specific exceptions and error messages.

```
no.hvl.exceptions.NodeException: There was an error with a node @
File name: HelloWorld.java
Line start: 7
Line end: 13

Cause: The replacementId "2" does not match any id of the ReplacementCode annotations

Node:
@Exercise(id = { 1 }, transformOption = TransformOption.REPLACE_SOLUTION, replacementId = "2")
public static void main(String[] args) {
    String helloWorldMessage = "Hello World!";
    SolutionStart start;
    System.out.print(helloWorldMessage);
    SolutionEnd end;
}
```

Figure 5.1: A console showing a `NodeException` error message.

5.1.4 Generating

Going back to the `generate` method of the `Generator` class, the next step is to generate the solution project and the start code project. The relevant classes for this are in the `writers` package. A `ProjectWriter` object is created next in the `generate` method by taking the `Configuration` object and `Assignment` object as arguments in the constructor. The `createAllProjects` method in this class is then called to begin generating the projects. The first step in this process is to delete all files in the target folder to not create duplicate files. Modification of files follows the *visitor pattern*¹⁰ by having classes that implement the `FileVisitor` interface from the official Java file library. This interface provides methods that can be implemented to perform some operation at certain points when traversing depth-first down a file tree. These points are either before visiting a directory, while visiting a file, after visiting a directory, or if a file visit failed. The `DeleteFileVisitor` class implements the `visitFile` method to delete a file from the file system, while the `postVisitDirectory` method is implemented to delete the folders after they have been visited. This visitor is used to traverse the file tree that is rooted in the target path folder. Before deleting a file, it checks if it is an `AsciiDoc` exercise description file. In that case, the content of the file needs to be saved before deleting, to retain any manual changes of the description. The `preVisitDirectory` method also skips visiting any `Git` folders to both avoid run-time errors when deleting them (due to restrictive access by the operating system), and to not destroy any potential `Git` repositories in the target folder.

After the target folder is cleared, the files in the source project (i.e., the annotated solution project) are copied over using the `CopyFileVisitor` class. This

¹⁰https://en.wikipedia.org/wiki/Visitor_pattern

works in the same way as `DeleteFileVisitor` by implementing the same interface, only that files are copied instead of deleted. All file paths are checked if they match any of the glob patterns to ignore that are listed the in the `pom.xml` file, in which case they are not copied over. Copying regular files is easy due to the Java `file` library providing a method to do exactly this. However, copying over Java files must include all the transformations that were done previously. These changes are stored in the list of ASTs in the `Assignment` object. Luckily, the `JavaParser` API offers the ability to print an AST, or a `CompilationUnit` rather, to a string. This string is then simply written to a file, maintaining the transformations in the process.

The next task is to generate the exercise descriptions. This is done by the `DescriptionWriter` class, which contains a small home-brewed API for creating AsciiDoc files and attributes. The hierarchical structure of an exercise description is created by going through the `Exercise` and `Task` objects in a recursive manner. Depending on configuration settings, any previous descriptions that were saved before deleting them with the `DeleteFileVisitor` may also play a part in generating the new descriptions. Finally, the last part of the `generate` method is to clear the POM file for the source project of any FREJA information, similar to how the syntax trees removed this information previously. This is done by the `MavenWriter` class, which navigates the XML tree-like structure of the original `pom.xml` file, and creates two new POM files without the FREJA information. Any other information that were included in the original `pom.xml` file will be kept.

5.1.5 Testing

Test-driven development has been incorporated extensively during the development of FREJA. The result of this is a large test-suite and high confidence that the behavior of the framework is working as intended. The tests are mostly grouped together based on what class they are testing. For example, the `MavenWriterTest` class has multiple tests that check the behavior of the `MavenWriter` class. The nature of how FREJA functions has made it a difficult to test it. First of all, there needs to be test data, which in this case is Java code represented as syntax trees. Creating this programmatically is tedious and too cumbersome [55]. Therefore, the test folder has a package with example Java files that are used for testing. These are parsed using the `JavaParser` library to easily create syntax trees. However, sometimes there is only a specific part of an AST that is needed for a test. The `TestId` annotation is created to make it much easier to target specific code fragments. This annotation takes an integer as a value that can be thought of as an identification for a code fragment that should be used for testing. With the help of methods from the `TestUtils` class, this system provides an easy way of creating test data for tests. An example of marking code with the `TestId` annotation can be seen in Listing 5.7, which is then used as test data for a test in Listing 5.8.

```
1     @TestId(1)
2     @Exercise(id = {1,2}, transformOption = REMOVE_EVERYTHING)
3     public int fieldVariable;
```

Listing 5.7: An example of test data that is identified by the `TestId` annotation.


```

1     @Test
2     void testGettingIdValueFromNormalExerciseAnnotation() {
3         NodeWithAnnotations<?> node = TestUtils.getNodeWithId(
4             parser.getCompilationUnitCopies(), 1);
5         assertEquals(new int[]{1, 2},
6             getIdValueInExerciseAnnotation(node));
7     }

```

Listing 5.8: The test data from Listing 5.7 is retrieved using the `getNodeWithId` method.

5.2 Programming Language Agnostic Prototype Architecture

This section covers the architecture of the DSL that is created with MPS, while also explaining the language thoroughly. However, for anyone interested in developing the language further, we recommend acquiring a deep understanding of MPS before starting development. The scope of this thesis does not provide the opportunity to explain all the intrinsics of MPS. Instead, we refer to the official resource page for learning MPS¹¹ to help in this regard. In either case, one important concept to understand is that a language in MPS consists of several models, each defining a certain *aspect* of the language. The most important aspect is the *structure aspect*.

5.2.1 Structure

The structure aspect is the only required aspect for a language, and any potential other aspects are based on the definitions from the structure aspect. It describes the nodes and the structure of the language AST through *concepts*. This is similar to how terminals and non-terminals from a grammar defines the structure of a formal language. Since MPS store files as ASTs and not text files, the nodes of the AST are defined directly through concepts. A concept defines the properties of a node, along with additional information such as child concepts, parent concepts, and references. The `File` concept is the basis of the whole language. Its definition can be seen in Fig. 5.2 on the following page, which shows that it extends the `BaseConcept`. This is by default, and is similar to how Java classes automatically extend the `Object` class. The interface `INamedConcept` is implemented, and allows `File` nodes to have a name. The name value for this concept is just the filename of the imported file. The string-type `extension` property has the value of the file extension, e.g., `.java`. The definition also shows that instances of this concept can be root nodes. Lastly, the child nodes of a `File` node can be any number of concept nodes that implement the `IFileComponent` interface.

The concepts `LineOfText`, `SolutionStart`, `SolutionReplacementStart`, and `Solution` all implement this interface. The `LineOfText` concept has only a string property that represents a single line of text. Both `Solution` and `SolutionReplacement` can have multiple `LineOfText` child nodes to represent

¹¹<https://www.jetbrains.com/mps/learn/>

```

concept File extends BaseConcept
    implements INamedConcept

instance can be root: true
alias: file
short description: <no short description>

properties:
extension : string

children:
fileComponents : IFileComponent[0..n]

references:
<< ... >>

```

Figure 5.2: The definition of the File concept.

a bigger code fragment. A `Solution` node has two properties, `taskNumber` and `transformOption`. The former is just an integer, while the latter uses a value from the `TransformOption` enumeration, i.e., `REMOVED` or `REPLACED`. Depending on this value, a `Solution` node can have a reference to a `SolutionReplacement` node. Both the `SolutionStart` concept and the `SolutionReplacementStart` concept are meant to be temporary "marker" nodes, used to ease the process of creating `Solution` and `SolutionReplacement` nodes, respectively.

5.2.2 Editor

Building ASTs of a language in MPS takes place inside an editor. The *editor aspect* defines how this editor should look like, and some of its functionality. Each concept in a language can have its own editor definition. Editor definitions use a dedicated editor language to describe *cells* that make up the editor. Figure 5.3 shows the editor definition for the `File` concept, which have multiple rectangular boxes that each represent a cell.

```

<default> editor for concept File
node cell layout:
  [/]
  [- File { name } extension { extension } -]
  [/]
  ^(/ % fileComponents % /)
  /empty cell: <default>
  /]
  /]

```

Figure 5.3: The editor definition for the File concept.

The different symbols refer to different cell types. In this case, there is an outermost vertical collection cell, which has a horizontal collection at the beginning that displays constants ("File" and "extension"), and also shows the value of the `name` and `extension` properties. Below it is another vertical collection that displays all of the `fileComponents` child nodes, which will refer to the editor definition of each child node concept. There is a lot more customization that can be done for each cell by right-clicking it and selecting `Inspect Node`. The vertical collection cell of the `Solution` editor uses this to add a stylesheet to that cell, which draws a border around the node in the editor.

However, the editor aspect has the ability to define much more than just the visuals of the editor. Each concept can also have several defined keybindings that does something different each. For example, the `LineOfText` concept has multiple such definitions in the `markText_KeyMap` file. One of those definitions can be seen in Fig. 5.4, and is used to mark the end of a solution. The keybinding is `Ctrl + Alt + Enter`, and is applicable whenever the caret (i.e., the "text cursor") is placed inside a `LineOfText` node in the editor. The figure also shows the code that will get executed whenever the key combination is pressed. The code is written in `BaseLanguage`, and is essentially just the MPS version of Java. In this case, the code will navigate the parent `File` node and find the lines that make up the solution, remove them from the `File` node, and add them to a new `Solution` node that will get inserted in the same place.

```

item description      : mark solution end
keystrokes           : <ctrl+alt> + <VK_ENTER>
                    : Enter a character to handle KeyTyped events.
                    : Enter a VK_xxx constant to handle KeyPressed events.

caret policy        : ANY_POSITION
show in popup       : true
menu always shown   : false
is applicable        : <always>
execute             : (node, selectedNodes, editorContext)->void {
                    :   node<Solution> solution = new node<Solution>();
                    :   node<File> file = ((node<File>) selectedNodes.get(0).parent);
                    :   foreach sibling in selectedNodes.get(0).prev-siblings {
                    :     if (sibling.isInstanceOf(SolutionStart)) {
                    :       foreach nextSibling in sibling.next-siblings {
                    :         file.fileComponents.remove(((node<LineOfText>) nextSibling));
                    :         solution.linesOfText.add((node<LineOfText>) nextSibling);
                    :         if (nextSibling == selectedNodes.get(0)) { break; }
                    :       }
                    :       int index = file.fileComponents.indexOf((node<LineOfText>) (sibling));
                    :       file.fileComponents.insert(index, solution);
                    :       file.fileComponents.remove(sibling);
                    :     }
                    :   }
                    : }

```

Figure 5.4: A keybinding definition and the code that gets executed by it.

5.2.3 Input and Output

There was a brief mention earlier that programming files could be imported into the DSL through a button click. The concept and editor for this import button is defined in another language called `file.importer`. The editor definition has

a cell with a Java Swing¹² component for the button. Inspecting this cell shows the code that will get executed on a button click. This code attempts to call the `importData` method of the `ImportLogic` Java class. The class is located under the `util models` aspect of the first language. The purpose of this method is to create a new `File` root node for the imported file. It iterates through every line in the file and creates a `LineOfText` node for each, which are then added as child nodes for the `File` node.

The `TextGen` aspect can have generators for each concept, that determine how to transform a node in an AST into text. This aspect contains the logic for generating a text file from a `File` root node. In practice, this will generate a programming file that has the start code for one or more exercises. The implementation for this is very simple, mostly due to the simple nature of the language itself. There are only two generators, one for the `LineOfText` concept, and one for the `File` concept. The generation builds a text string by going through the AST and appending text by invoking the generator for each node in the tree. The `LineOfText` generator simply just appends the `text` property and a newline. The `Solution` concept and `SolutionReplacement` concept does not need its own generators, since they can refer to the `LineOfText` generator to append the necessary parts. The `File` generator can be seen in Fig. 5.5, which handles the logic of making other concepts use the `LineOfText` generator. The `append${text}` line at the end will implicitly use the `LineOfText` generator since the concept type for the `text` variable is `LineOfText`.

```

text gen component for concept File {
  file name : <Node.name>
  file path : <node\qualified/name>
  extension : (node)->string {
    node.extension;
  }
  encoding : utf-8
  text layout : <no layout>
  context objects : << ... >>

  (node)->void {
    foreach fileComponent in node.fileComponents {
      if (fileComponent.isInstanceOf(Solution)) {
        node<Solution> solution = ((node<Solution>) fileComponent);
        if (solution.transformOption.is(REMOVED)) { continue; }
        if (solution.transformOption.is(REPLACED)) {
          node<LineOfText> text = ((node<LineOfText>) solution.linesOfText.get(0));
          int leadingSpaces = text.text.length() - text.text.stripLeading().length();
          string leadingWhiteSpace = " ".repeat(leadingSpaces);
          foreach lineOfText in solution.replacement.linesOfText {
            append ${leadingWhiteSpace};
            append ${lineOfText};
          }
        }
      }
      if (fileComponent.isInstanceOf(LineOfText)) {
        node<LineOfText> text = ((node<LineOfText>) fileComponent);
        append ${text};
      }
    }
  }
}

```

Figure 5.5: The `TextGen` definition for generating a `File` node into text.

¹²[https://en.wikipedia.org/wiki/Swing_\(Java\)](https://en.wikipedia.org/wiki/Swing_(Java))

Chapter 6

Installation and Usage

This chapter explains how to install FREJA in a step-by-step guide. All annotations, annotation elements, and other options are presented with thorough explanations about what they do and how they should be used. A short tutorial is included, which demonstrates how to use FREJA on a small example assignment. The second half of the chapter covers the same points for the programming language agnostic prototype.

6.1 Installing FREJA

To install FREJA, head over to the GitHub repository for it: <https://github.com/ErlendBerntsen/freja>. Get a local copy of the framework by either cloning, forking or downloading the repository. The project relies on Maven¹ so make sure to have it installed before continuing. Open a terminal and navigate to the root folder where FREJA was installed. This is easiest done by opening the project in an IDE and using the built-in terminal interface, which will most likely already be in the correct directory. Once inside the right directory, run the following Maven command:

```
mvn clean install
```

to install the framework into the local Maven repository. This will install two JARs, one Maven dependency for the annotations, and one Maven plugin that handles the parsing and artefact generation. The JDK source and target version properties in the `pom.xml` file may need to be configured to be compatible with a different development environment. This will be set to Java 16 when first cloning FREJA, as seen in Listing 6.1. If Java 8 is used for example, then the value 16 must be changed to 1.8.

```
1 <properties>
2   <maven.compiler.source>16</maven.compiler.source>
3   <maven.compiler.target>16</maven.compiler.target>
4 </properties>
```

Listing 6.1: The default JDK version in the `pom.xml` file.

¹<https://maven.apache.org/install.html>

To start using the framework, create a new Maven project in a Java IDE. Open up the `pom.xml` file in the new project and include the `freja-annotations` dependency from Listing 6.2, which will enable the use of FREJA annotations in the project:

```
1 <dependency>
2   <groupId>no.hvl</groupId>
3   <artifactId>freja-annotations</artifactId>
4   <version>1.1</version>
5 </dependency>
```

Listing 6.2: FREJA annotations Maven dependency.

Next, reload the POM file or the entire project. If for some reason the IDE cannot find the dependency at this point, add the JAR to the project as an external library instead. The JAR is found in the target folder in the `freja-annotations` module in the local FREJA project.

The `freja-maven-plugin` must be added to the Maven build life cycle in the `pom.xml` file, to be able to execute FREJA. This plugin has a `configuration` attribute called `targetPath`, which defines the path of the target folder to generate the artefacts in. An example of this is shown in Listing 6.3 below:

```
1 <build>
2   <plugins>
3     <plugin>
4       <groupId>no.hvl</groupId>
5       <artifactId>freja-maven-plugin</artifactId>
6       <version>1.1</version>
7       <configuration>
8         <targetPath>C:\Users\Acer\IntelliJProjects\
9           HelloWorldOutput</targetPath>
10      </configuration>
11    </plugin>
12  </plugins>
</build>
```

Listing 6.3: The FREJA Maven plugin specifying the target folder.

The `targetPath` configuration option is the only one that is required. There are more options that are optional:

- `keepOldDescriptions` is a boolean attribute that when set to true, prevents the old exercise descriptions (if any) from being overwritten by the default description templates when regenerating them. If there are not any old description files in the target folder, then the default template is used instead. This option is set to false by default.
- `ignore` is an attribute that takes a list of strings as arguments. This is used to specify files and folders that should be ignored when generating the artefacts. The path of these files and folders should be specified relative to the root folder of the source project, using glob patterns². Use the `<ignore>` element tag for both the list and for each list element. An example is shown in Listing 6.4 on the next page with some common patterns:

²[https://en.wikipedia.org/wiki/Glob_\(programming\)](https://en.wikipedia.org/wiki/Glob_(programming))

```

1 <ignore>
2   <ignore>.idea</ignore>
3   <ignore>HelloWorld.iml</ignore>
4   <ignore>src/test</ignore>
5   <ignore>src/main/java/no/hvl/IgnoreThisFile.java</ignore>
6   <ignore>**IgnoreThisFileToo.java</ignore>
7   <ignore>**.txt</ignore>
8 </ignore>

```

Listing 6.4: Different types of glob patterns for the `ignore` configuration.

- The pattern at line 2 ignores a folder called `.idea` (and all of its content) in the project root folder.
- Line 3 ignores a file called `HelloWorld.iml` in the project root folder.
- Line 4 ignores a folder called `test` that is in the `src` folder (the `src` folder is located in the project root folder).
- Line 5 ignores a file called `IgnoreThisFile.java` by specifying its complete path from the project root.
- Line 6 ignores every file called `IgnoreThisFileToo.java` no matter its location relative to the project root.
- Line 7 ignores every file in the project ending with `.txt`

Make sure that the root folder for Java files is either called `src` or `source`. To generate artefacts from the source project, run the following command:

```
mvn freja:generate
```

from the terminal. Make sure to be in the same directory as the POM file. This will not transform the original project in any way if there has not been used any FREJA annotations in the project. At this point, the development of the assignment solution can start, along with annotations that specify the transformation details. Whenever there is a need to regenerate the artefacts, simply run the above command again. To learn more about how to use the annotations, or attributes in AsciiDoc, check out the *Implementation* chapter, or the README file in the GitHub repository. There is also a tutorial in the next section that demonstrates how to use the framework on a simple assignment.

6.2 Example

To give a glimpse into how FREJA is used, a short tutorial example is given here. The assignment has only one exercise, and it is the popular *HelloWorld* exercise. The only thing that should be implemented is a method that prints out "Hello World!" to the console. Different annotations, transformation options, and configuration settings, are used to explain the functionality of FREJA and its usefulness.

Start by creating a new Maven project in a Java IDE. The name of the project is not that important, but "FrejaTutorial" is the project name that is used throughout the example in this tutorial. Create another folder for storing the

artefacts that will be generated later. The tutorial uses "FrejaTutorialOutput" as the name for this folder. Open the `pom.xml` file in the "FrejaTutorial" project and add the dependency and plugin from Listing 6.5 at the end of the file:

```
1 <dependencies>
2   <dependency>
3     <groupId>no.hvl</groupId>
4     <artifactId>freja-annotations</artifactId>
5     <version>1.1</version>
6   </dependency>
7 </dependencies>
8
9 <build>
10  <plugins>
11    <plugin>
12      <groupId>no.hvl</groupId>
13      <artifactId>freja-maven-plugin</artifactId>
14      <version>1.1</version>
15      <configuration>
16        <targetPath>C:\Users\Acer\IntelliJProjects\
17          FrejaTutorialOutput</targetPath>
18      </configuration>
19    </plugin>
20  </plugins>
</build>
```

Listing 6.5: FREJA tutorial POM file.

Remember to change the `targetPath` to the absolute path of the folder created to store the generated projects. Go back to the beginning of this chapter if the FREJA Maven plugin and dependency are not already installed, and follow the installation steps explained there. After changing the `pom.xml` file, the project might need to be reloaded so that the IDE has knowledge of the imported Maven plugins and dependencies.

If there is not already a folder called `src` at the root level in the project, create one now. Right-click it to add a package with the name `no.hvl.freja`. Right-click the package and add new Java class called `HelloWorld`. Add a method with the same name as the class that simply prints "Hello World!" to the console. The class should look like the one in Listing 6.6 at this point. This is the complete solution to the exercise.

```
1 package no.hvl.freja;
2
3 public class HelloWorld {
4
5     public void helloWorld(){
6         System.out.print("Hello World!");
7     }
8 }
```

Listing 6.6: The simple HelloWorld class so far.

The exercise is targeted at beginner programmers so the implementation task should not be too difficult. The start code can remove the print statement, but still have the skeleton of the method to keep it simple. To do this, add an `Exercise` annotation on top of the method declaration. Since this is the only exercise for the assignment, the `id` of the exercise is 1. Set the `transformOption` to

`TransformOption.REMOVE_BODY` so that only the method body will be removed, as seen in Listing 6.7.

```
1     @Exercise(id = {1}, transformOption = TransformOption.  
2         REMOVE_BODY)  
3     public void helloWorld(){  
4         System.out.print("Hello World!");  
    }
```

Listing 6.7: The `helloWorld` method with the `Exercise` annotation.

Open the terminal in the root folder of the project. This path might vary depending on storage location and naming, but this tutorial uses the following path for the folder:

```
C:\Users\Acer\IntelliJProjects\FrejaTutorial>
```

Run the command below to generate the solution project and start code project in the target folder:

```
mvn freja:generate
```

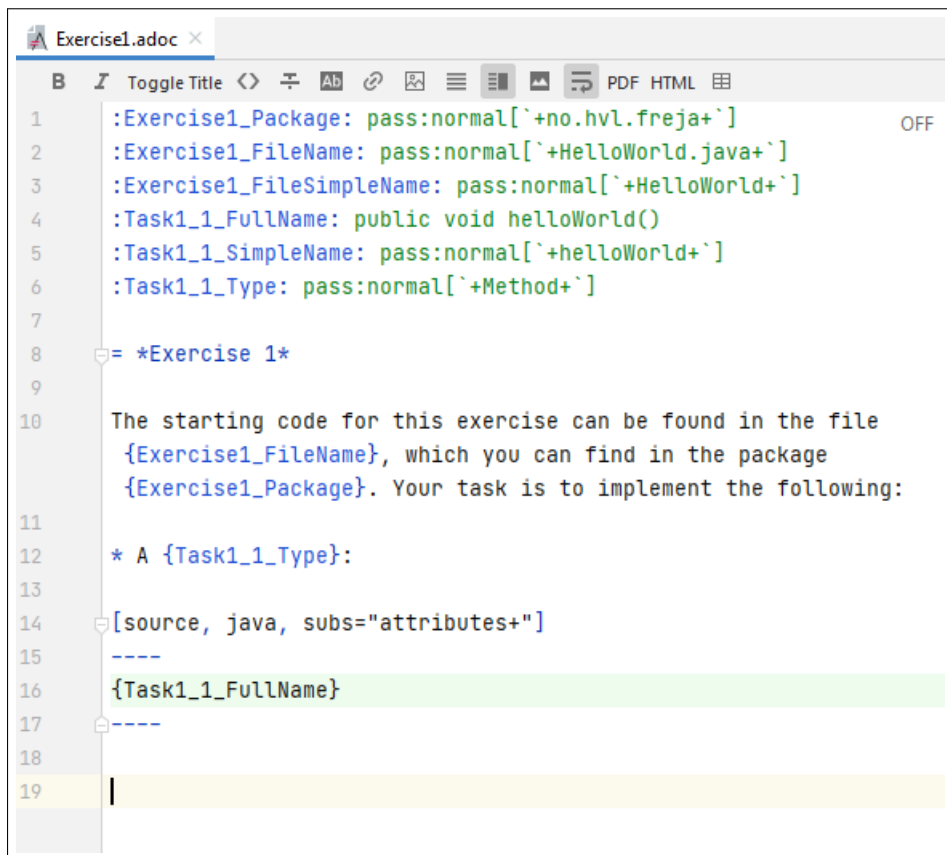
Open the target folder and there should be two new folders, one called `solution` and one called `start code`. Open the solution project in a Java IDE³, and it should still be very similar to the source project. Opening the `HelloWorld.java` file reveals that the `Exercise` annotation is removed, along with the import statements for the annotation. The only other difference should be in the `pom.xml` file, which has removed all information related to `FREJA`. The solution project will always be very similar to the original project since all of the solutions to the exercises are kept. All `FREJA` related information is removed to create a clean and uncluttered solution project.

Close the solution project and open the start code project instead. The same changes are made to the `pom.xml` file in this project as well, since those Maven dependencies and plugins are not needed for the start code either. Open the `HelloWorld.java` file and see that the print statement is removed from the `helloWorld` method, but the rest of the code remains, just as intended. Again, the `Exercise` annotation and import statements are also removed.

There is a new folder created in the start code project called `descriptions`. Open it and there should be an AsciiDoc file there called `Exercise1`. Make sure to have an AsciiDoc IDE plugin installed so that both the editor window and the output window of the AsciiDoc file are shown. Open the file and the editor should look like the one in Fig. 6.1 at the following page. This is a screenshot of the IntelliJ AsciiDoc plugin editor window, so it may look a little different if something else is used, but the text in the editor should remain the same. Figure 6.1 shows that there are six attributes created at the top. Three of them concern the location of the exercise (file and package information), and the other three is about the implementation task in the exercise, i.e., the `helloWorld` method.

Below the attributes is an automatically generated template description of the exercise. This is a good starting point for the exercise description, and demonstrates how attributes are used to create substitutions. However, the template

³Read the following if Eclipse is used: <https://github.com/ErlendBerntsen/freja/blob/master/README.md#eclipse>



The screenshot shows the IntelliJ AsciiDoc plugin editor window for a file named 'Exercise1.adoc'. The editor displays the following AsciiDoc content:

```
1 :Exercise1_Package: pass:normal[`+no.hvl.freja+`] OFF
2 :Exercise1_FileName: pass:normal[`+HelloWorld.java+`]
3 :Exercise1_FileSimpleName: pass:normal[`+HelloWorld+`]
4 :Task1_1_FullName: public void helloWorld()
5 :Task1_1_SimpleName: pass:normal[`+helloWorld+`]
6 :Task1_1_Type: pass:normal[`+Method+`]
7
8 = *Exercise 1*
9
10 The starting code for this exercise can be found in the file
11 {Exercise1_FileName}, which you can find in the package
12 {Exercise1_Package}. Your task is to implement the following:
13
14 * A {Task1_1_Type}:
15 [source, java, subs="attributes+"]
16 ----
17 {Task1_1_FullName}
18 ----
19 |
```

Figure 6.1: A screenshot of the IntelliJ AsciiDoc plugin editor window.

description does not describe the purpose of the code, so often there is a need to expand the description to further explain what exactly the method should do. Go to the bottom of the file in the editor view and add an extra sentence:

The `{Task1_1.SimpleName}` method should print out "Hello World!" to the console.

Now the exercise description has information about what method to implement, where it is located, and what it should do. The AsciiDoc output window at this point should be the same as the one in Fig. 6.2.



Figure 6.2: A screenshot of the AsciiDoc output window.

When an exercise description is manually changed, the configuration in the `pom.xml` in the `FrejaTutorial` project must also be updated, as to not overwrite the changes when executing FREJA again. Add the configuration option

```
<keepOldDescriptions>true</keepOldDescriptions>
```

below `targetPath` inside the configuration element of the Maven plugin. Furthermore, the POM file is not needed in the generated projects since it does not import any libraries, so it might as well be removed. To do this, add an `ignore` element in the configuration element. Add another `ignore` element within the first one, with "pom.xml" as its value. The configuration element at this point should look like Listing 6.8 (the `targetPath` will of course be different).

```
1 <configuration>
2   <targetPath>C:\Users\Acer\IntelliJProjects\FrejaTutorialOutput</
   targetPath>
3   <keepOldDescriptions>true</keepOldDescriptions>
4   <ignore>
5     <ignore>pom.xml</ignore>
6   </ignore>
7 </configuration>
```

Listing 6.8: Configuration settings.

The program for the exercise is not very exciting since it cannot be executed to see the message being printed to the console. To fix this, make the method static and change its name to `main` that has an array of String objects called `args` as a parameter, as seen in Listing 6.9.

```

1  @Exercise(id = {1}, transformOption = TransformOption.
2      REMOVE_BODY)
3  public static void main(String[] args){
4      System.out.print("Hello World!");
5  }

```

Listing 6.9: A main method.

There has been a fair bit of changes to the assignment at this point, so this is a good time to regenerate the projects to update them with the new changes. Execute the `mvn freja:generate` command again, and then open the start code project. Notice that the `pom.xml` file is not copied over this time. Navigate to the exercise description file and open it. The manual change to the description has been kept, but the text in the output window, shown in Fig. 6.3, has been updated automatically to reflect the change in the method declaration.

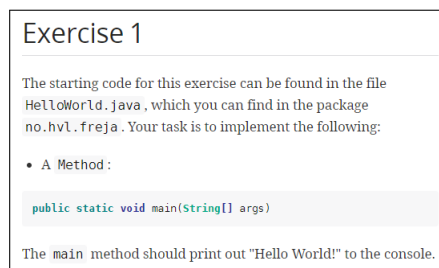


Figure 6.3: A screenshot of the updated AsciiDoc output window.

Taking a closer look at the editor window seen in Fig. 6.4 reveals that the exercise description has not changed at all! It is only the value of the AsciiDoc attributes at line 4 and 5 that has been updated.

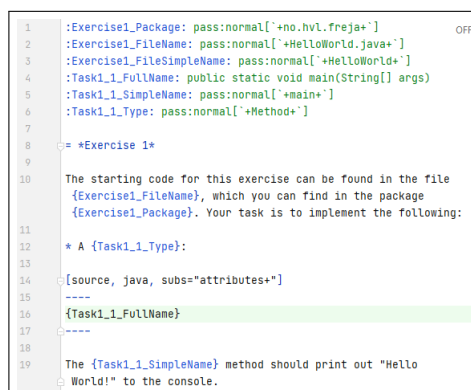


Figure 6.4: Only the attribute values at line 4 and 5 has changed, while the attribute references at line 16 and 19 remain unchanged.

Hopefully this showcases the power and potential by connecting the assignment description to the source code through attributes. Lastly, an example of marking an exercise solution, and replacing it, is shown to finish the tutorial. Since the *HelloWorld* example is famously a beginner exercise for newcomers, it might be wise to make the implementation task of the method as simple as possible. Instead of removing the entire body of the method, the output message will be kept in a string variable that is present in the start code. The exercise is then to simply call the `print` method with the string variable. To make it even more straightforward, the solution is not removed entirely, but replaced with an empty call to the `print` method.

Implement the described changes to the solution. Add a `SolutionStart` variable declaration after the string variable and before the `print` statement, and a `SolutionEnd` variable declaration after the print statement. In the `Exercise` annotation, change the `transformOption` to `TransformOption.REPLACE_SOLUTION`. Add the `replacementId` annotation element, and set its value to "1". The replacement code needs to be written somewhere, so create a new void method below the `main` method that will store the replacement code. It does not matter what the method is called, as long as it only contains the empty print statement:

```
System.out.print("");
```

Annotate the replacement method with `@ReplacementCode`, and then add the `id` annotation element and set its value to "1". Now it will be linked to the `replacementId` in the `Exercise` annotation, since they have the same value. The replacement code does not need to be present in the start code, so add the `@Remove` annotation on top of the `@ReplacementCode` annotation to make sure the replacement code will be removed entirely. The code should now look like the source code in Listing 6.10. Execute the Maven command as before to regenerate the projects with the new changes. The generated start code for the `main` method should now be the same as in Listing 6.11.

```
1      @Exercise(id = {1}, transformOption = TransformOption.  
2          REPLACE_SOLUTION, replacementId = "1")  
3      public static void main(String[] args){  
4          String helloWorldMessage = "Hello World!";  
5          SolutionStart start;  
6          System.out.print(helloWorldMessage);  
7          SolutionEnd end;  
8      }  
9  
10     @Remove  
11     @ReplacementCode(id = "1")  
12     public void replacement(){  
13         System.out.print("");  
14     }
```

Listing 6.10: The final source code.

```
1      public static void main(String[] args) {  
2          String helloWorldMessage = "Hello World!";  
3          // TODO - START  
4          System.out.print("");  
5          // TODO - END  
6      }
```

Listing 6.11: The start code after the solution has been replaced.

6.3 Installing the Programming Language Agnostic Prototype

MPS must be installed to be able to use the prototype. Following the official installation guide⁴ for MPS is the easiest way to install it. After MPS is installed, open it and select the *Get from VCS* option in the welcome dialog box. If the IDE is already opened with another project, select the *VCS* option in the toolbar at the top, and then *Get from Version Control*. Clone the GitHub repository for the prototype: <https://github.com/ErlendBerntsen/Paastel> by pasting the URI into the dialog box. Enable the *Logical View* in the project window, which might be a little slow to update when the IDE first clones and indexes the project, so give it some time to do its job. Afterwards, right-click the root folder of the project and select **Rebuild Project**.

There should be two languages that are imported, `Paastel`⁵ and `file.importer`. The latter is just a helper language to import programming files into MPS, and convert them into the `Paastel` DSL. The languages contain definitions of several language aspects, such as concepts, editors, and generators. It is not necessary to worry about all the implementation details to use the languages. A `Solution` in MPS has instances, or `models`, of a language. A `model` is a collection of files that are instances of a root node for a language. Find the `assignments` solution in the logical view and open it, and then open the `files` model. This model contains a file, called `Import`, which is an instance of the `ImportConfiguration` root concept of the `file.importer` language. The editor for this concept can be seen in Fig. 6.5.



Figure 6.5: The `Import` editor.

This editor imports a programming file and converts it into an instance of the `File` root concept of the `Paastel` language. By pressing the `Import` button, a `File` instance will be created and automatically added to the `files` model. The programming files that are imported should be part of an assignment, and should contain the implemented exercise solutions before importing them. Any file that should be imported must be within the same location as the MPS project. The relative path is then specified in the `Import` editor window, along with the number of spaces in tabs for the source file. The last point is important for any programming language that rely on indentation, such as Python.

⁴<https://www.jetbrains.com/help/mps/installation-guide.html>

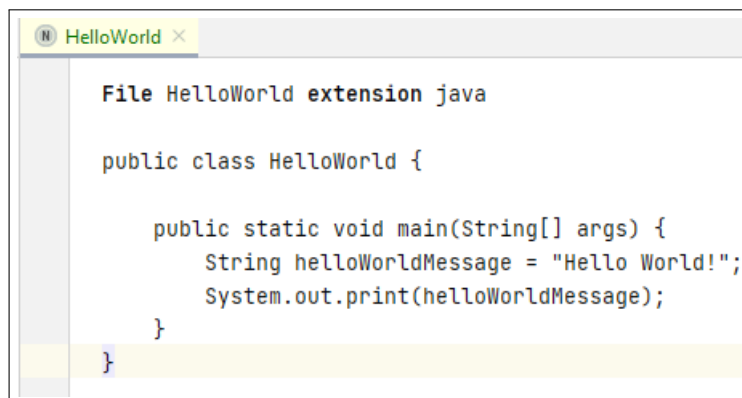
⁵This name stuck around after the initial prototype. It is short for **P**rogramming **A**ssignment **A**ST **E**ditor **L**angage.

After importing a file, the corresponding `File` concept can be opened. This will bring up the editor for it, which gives the ability mark assignment concepts and specify transformation rules. The next section goes into more detail about how to do this. Once the editing of a file is completed, the start code can be previewed by right-clicking the editor window and selecting the `Preview Generated Text` option in the popup menu. If the output is not satisfactory, repeat the process of editing the `File` instance and previewing it until it is. To transform the preview to an actual file, right-click the `assignments` solution in the logical view, and select the `Rebuild Solution 'assignments'` option. This will generate the output file in a place that can be found by switching from the logical view to `Project Files`, and navigating to

```
solutions\assignments\source_gen\assignments\files
```

6.4 Example

This section explains how to use the DSL and its editor, and showcases what options and functionality it can provide through a small tutorial. The same *Hello World* assignment example that was used in the tutorial for FREJA is also used in this tutorial. The solution to the assignment is implemented in the `HelloWorld.java` file, which is already included in the MPS project. It will not show up in the logical view, but switching to the `Project Files` view will reveal it in the `input` folder. Open up the `Import` editor, and the source file should already be set to the correct path if it has not been changed after cloning the project. If it has been changed, set the source file path back to `input/HelloWorld.java` now. Press the import button to create a `File` root node for the file, and then open it to bring up its editor, which should look like the editor in Fig. 6.6.



```
File HelloWorld extension java

public class HelloWorld {

    public static void main(String[] args) {
        String helloWorldMessage = "Hello World!";
        System.out.print(helloWorldMessage);
    }
}
```

Figure 6.6: The editor for the `HelloWorld` file.

The goal is the same as before, i.e., creating the start code for the exercise where the solution is removed, but the method skeleton is kept. The first step is to mark the start of the solution, i.e., the first line of code in the method body. Select the first line by placing the caret (the "text cursor") anywhere on that line, and then press `Alt + Enter`. Alternatively, use the mouse to right-click

the line, and select the `mark solution start` option in the popup menu. This will insert a `SolutionStart` concept before that line. Select the last line in the method body the same way, but use the keyboard combination `Ctrl + Alt + Enter`, or the `mark solution end` option instead.

Once the end is selected, the editor then searches for the closest `SolutionStart` concept that is above the end line, and replaces the contained lines with a `Solution` concept. The lines that were selected as part of the solution can be seen by the border of the solution box. Whenever a `Solution` is created, the options for it are not set to any values. Specify these values for the `Solution` concept that was just created by setting the `taskNumber` to 1, and the `transformOption` to `REMOVED`. Pressing `Ctrl + Space` anywhere in MPS will bring up a suggestion menu for things that can be inserted wherever the caret is placed. For example, placing the caret where it says `<no transformOption>` and pressing `Ctrl + Space` will bring up the menu of possible `TransformOptions` that can be used. An option from this menu can be selected by hitting `Enter`. After the values are set, right-click anywhere in the editor and select `Preview Generated Text` to see the preview of the start code. The editor for the `HelloWorld` file should look like the top part of Fig. 6.7 at this point, while the preview can be seen underneath the editor.

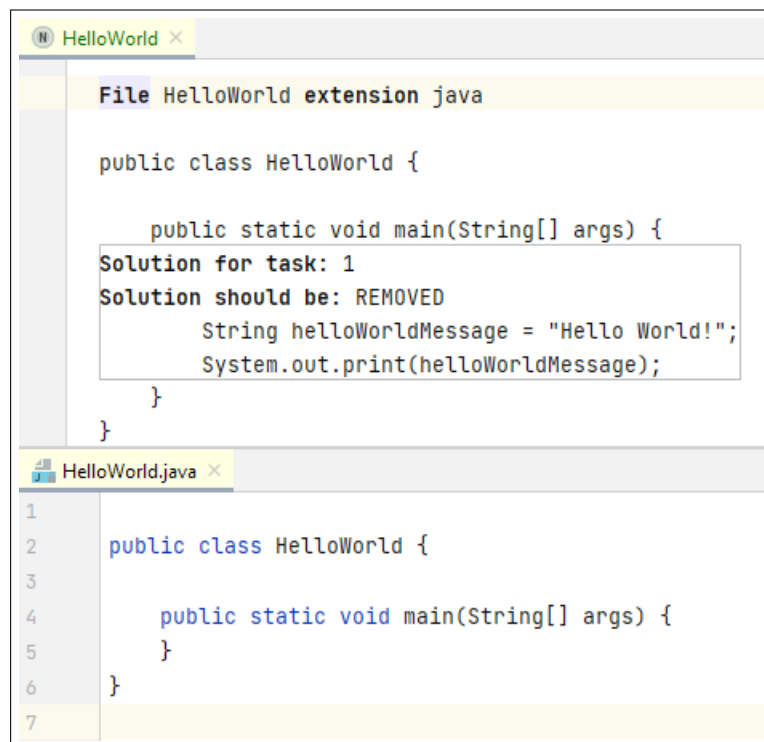


Figure 6.7: The marked solution and a preview of the start code.

The FREJA tutorial showcased that the solution could not only be removed, but also replaced. This is also possible with the MPS prototype framework, and the next step is to do exactly that. Instead of removing the entire method body,

the string variable will be preserved, and the print statement will be replaced. To preserve the string variable for the start code, it should not be marked as a part of the solution. Fortunately, the power of having a customized editor makes this easy to fix. Select the last line of the method body, i.e., the print statement, as the start of the solution the same way as before. The solution should be updated to only have the print statement, while the string variable is moved above the solution.

An instance of a `SolutionReplacement` needs to be created to be able to have code replacements. There are two ways to do this. One is to right-click the `files` model folder in the `assignments` solution, and selecting `new -> Paastel -> solutionReplacement` from the popup menu. The replacement code must be created from scratch doing it this way, as well as specifying all of the replacement settings. However, since the solution should be replaced with an empty print statement, the solution itself can be used as a starting point for the replacement code. Move the caret to the line containing the print statement for the solution, and press `Alt + Comma`. This works exactly the same way as marking a solution by inserting a marker concept before the selected line. Select the same line again and press `Ctrl + Alt + Comma` to create a `SolutionReplacement`. This will show up in the `files` model folder as `SolutionReplacement0`. Open it, and change the print statement to be empty. Remove the leading indentation also, as seen in Fig. 6.8.

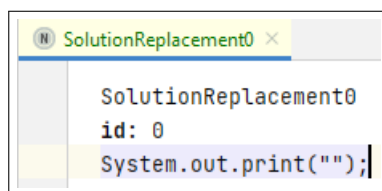


Figure 6.8: The editor for the `SolutionReplacement`.

Go back to the `HelloWorld` editor and change the `transformOption` for the solution to `REPLACED`. Notice that the editor programmatically changed based on the value of `transformOption`. Now the `Solution` editor provides the ability to specify a `SolutionReplacement` to use, by referring to its name. Trying to write a name that does not exist will immediately be pointed out by the editor with bright red text. Place the caret where the replacement code name should be inserted. `Ctrl + Space` will again bring up all the possibilities, which in this case is only `SolutionReplacement0`. Open up the preview again once the correct replacement name has been selected. The editor for the `HelloWorld` file, and the corresponding preview at this point should look like Fig. 6.9 at the next page. The preview shows the desired start code, so generate the file by right-clicking the `assignments` solution in the logical view, and select the `Rebuild Solution 'assignments'` option. Switch from the logical view to `Project Files`, and the generated start code file should be located here:

```
solutions\assignments\source_gen\assignments\files\HelloWorld.java
```

```
File HelloWorld extension java

public class HelloWorld {

    public static void main(String[] args) {
        String helloWorldMessage = "Hello World!";
        Solution for task: 1
        Solution should be: REPLACED with SolutionReplacement0
        System.out.print(helloWorldMessage);
    }
}

HelloWorld.java x
1
2 public class HelloWorld {
3
4     public static void main(String[] args) {
5         String helloWorldMessage = "Hello World!";
6         System.out.print("");
7     }
8 }
```

Figure 6.9: The updated solution that should now be replaced, and its corresponding output below.

Chapter 7

Evaluation

This chapter goes into detail on how we tested both frameworks, with most of the focus being on FREJA. The results from these experiments are presented and discussed, including both shortcomings and benefits of each framework. Resulting changes to the implementations from these tests are also discussed.

7.1 A First Evaluation of FREJA

An assignment from DAT100 2021 fall semester was used to evaluate the first version of FREJA. This assignment was meant to be solved in groups with up to four members. With this first version we were confident that most of the functionality was working as intended, and that it could be efficiently used to create an assignment identical to the one it is primarily designed after. Evaluating the framework on another assignment was insightful for determining if it is too tailor-made for one assignment, or if it is useful in general for Java assignments. Testing FREJA this way would reveal any missing functionality, and also highlight things that work well. The results of the first evaluation are uploaded to GitHub, with one repository created for the FREJA-annotated source project¹, and another repository containing the generated output².

7.1.1 Details of the Assignment

The DAT100 assignment is split into four main artefacts. The first artefact is the solution project which was shared between lecturers and teacher assistants through GitHub³, with no access to students. This project contains the solutions for every exercise in the assignment. A second project, located in a different GitHub repository⁴, was created for the start code, which students were instructed to fork and then clone locally. Another separate GitHub repository⁵ was created for unit tests that students could clone to test their solutions.

¹<https://github.com/ErlendBerntsen/dat100-jplab11-solution-annotated>

²<https://github.com/ErlendBerntsen/dat100-jplab11-output>

³<https://github.com/dat100hib/dat100-jplab11-complete>

⁴<https://github.com/dat100hib/dat100-jplab11-startkode>

⁵<https://github.com/dat100hib/dat100-jplab11-testing>

Finally, the assignment description was uploaded to the main repository for DAT100⁶, separate from all the other repositories.

The overall theme of the assignment is to develop a basic blog system for the web, however, none of the exercises concern networking or web services. The point of the assignment is to learn how a blog system can be split into different concepts that can be represented as Java classes. The goal of the exercises vary, ranging from learning how to implement simple constructors, getters and setters, understanding inheritance through abstract classes and sub classes, to more complex implementation of input/output operations using local files.

The assignment is split into seven main exercises, with three of them being optional. There are 19 sub-exercises, with one exercise having as many as 13, and others having none at all. The source code is split into seven folders, one for each root exercise. The files for an exercise and its sub-exercises are mostly grouped together in the same folder, but there are some exceptions to this. There are a total of 11 Java files between the exercises, with one file that is unrelated to all exercises. Most of the exercises are contained within one file, but a couple exercises span across two or three files. The structure of the project is identical in the solution project and the start code project, with both containing exactly the same files and folders. Obviously, the implementations of the exercise solutions are removed from the start code.

The assignment description is written in markdown (more specifically GitHub flavored markdown⁷) and delivered as one large file. This is different from the assignment the implementation of FREJA is based on, where each exercise had its own separate markdown file. The description file of the testing assignment is structured into several parts. First, there is an introduction explaining the theme and purpose of the assignment. Then there is a section about how to hand in the solution, followed by another section that explains how the students can get the start code locally on their computer. There is also a section detailing the necessary steps to test their own solution using the test project.

Finally, there is an explanation for each exercise and sub-exercise concerning what the student should implement. The text uses different markdown header levels⁸ to clearly display the hierarchical structure of exercises and sub-exercises. The individual exercise descriptions includes an explanation of what to implement, and information about which class or classes the start code is in. There are also many additional references to the source code, such as method names or declarations. In total there are 85 non-unique references to the source code in the assignment description, such as package name, file name, and method names. All of these are hard-coded, and there are also eight additional references to the test source code and test files as well.

7.1.2 Testing and Results

By having access to all the different artefacts from the start, we had a clear blueprint of what the generated artefacts should look like. Testing FREJA on

⁶<https://github.com/dat100hib/dat100public/blob/master/programming/jplab11/JP11.md>

⁷<https://github.github.com/gfm>

⁸<https://www.markdownguide.org/basic-syntax/#headings>

the assignment described above started with cloning the complete solution locally, then adding both the Maven dependency for the annotations, and the Maven plugin for FREJA. The original project was an Eclipse project, but a POM file was created to be able to use the necessary Maven tools. Additionally, IntelliJ IDEA was used to test the framework, which did not pose any extra problems, even though the original project was developed in Eclipse. The solution project was annotated with the relevant annotations in the order of the exercises. The necessary transformation settings was specified to ensure the generated artefacts resembled the original solution, start code, and description as much as possible. After each exercise and sub-exercise, the plugin was executed to ensure the generated output was correct.

The very first sub-exercise is to implement four field variables in a class. Due to the rules of the annotation API, each field variable must be annotated individually. All of the field variables belong together in the same sub-exercise, and every variable should be removed from the start code, meaning that each `Exercise` annotation is identical. This makes for easy copy-paste work, but does clutter up the annotated solution file a bit. In the start code for this sub-exercise, there is *one* comment: `//TODO - declare field variables`, where the field variables should be implemented. It is not possible to specify that all field variables should be replaced with only one piece of code or comment, since they are all annotated individually. One option is to make one variable be replaced with the comment, and specify that the rest of the variables should be removed.

Unfortunately, the implementation of FREJA assumes that the replace option is used for bodies or solutions, both of which are block statements in the AST. A field variable is either a variable declaration or an assignment statement, so this option does not actually work. However, the comment is left in the original solution project as well, so just having the comment above the field variables in the source project will correctly imitate the original start code and solution. Ideally, the comment would only be in the start code, but this is not possible due to the limitations of the replace option in this initial version. The generated solution code differed slightly from the original by having a line of white space between each field variable. This is because the annotations will leave an empty line when they are removed, but the difference this makes is inconsequential enough that a change in the implementation to address this issue will likely not be a good use of resources.

The rest of the sub-exercises for exercise 1 was to implement constructors, getters and setters, or other simple methods. The start code for all of these sub-exercises had the same pattern. The solutions were swapped out for a throw-statement that informs the student that a method is not implemented. An example of this can be seen Listing 7.1.

```
1 public int getId() {
2     throw new UnsupportedOperationException("TODO.method()");
3 }
```

Listing 7.1: The original start code where the solution is replaced with a throw statement.

This is to allow students to test their implementation as they go. If not for this, the tests would not compile for later exercises that have not yet been implemented. Instead, implemented exercises can be tested, while unimplemented exercises will throw harmless run-time errors with useful error messages, including information about the unimplemented method name, or possibly the constructor and class name. A class called `TODO` contained methods that used reflection to figure out this information, and was included in the start code so that the error messages does not need to be hard coded. FREJA eliminates the need for this, since the parser has access to this information when constructing ASTs. This is achieved by having it as the default configuration, i.e., setting `transformOption` to `REPLACE_BODY` or `REPLACE_SOLUTION` without specifying a replacement id will replace the solution with this standard throw-statement. The thought behind this is to increase ease-of-use, and eliminate repetition by taking use of the *convention over configuration*⁹ design paradigm.

The generated exercise descriptions revealed a few bugs with FREJA's implementation. This is mostly due to relying on nested lists in AsciiDoc to create a hierarchical structure of exercises and sub-exercises in the description text. It is difficult to properly structure these lists when adding code listings and attribute substitutions inside of them. This resulted in changing the implementation to use section titles and levels¹⁰ instead to structure the exercises in the assignment description. Sections are equivalent to headers in regular markdown. The original assignment description used alphabetical numbering for sub-exercises, e.g., the first sub-exercise of exercise 1 is referred to as 1a, and so on. Again, this was not possible with the first version of FREJA, but the framework was updated to switch between numerical and alphabetical numbering between each level of exercise and sub-exercise to accommodate for this.

Another insight that was gained from exercise 1 was the amount of repetition in the generated exercise description. There are 6 sub-exercises which are all in the same Java class, but the generated description for each sub-exercise include a brief explanation of which file and package the start code is in. This is obviously identical for each sub-exercise and can be seen as redundant. However, it is really useful to include this information when sub-exercises does span across several classes. Unsurprisingly, the generated template description did not match the original exercise descriptions perfectly, leading to manual changes to make the description mimic the original. Removing the unnecessary repetition of file name and package information was not a time consuming task when manually editing the description. It is quicker to have that information already available, and have the option to quickly manually remove any potential redundancy, than to not have the information available at all. In the latter case, this information must be written manually, which is undoubtedly slower than removing it.

Several more shortcomings of the framework was revealed by the second exercise. The task was to implement two subclasses of an abstract class, which included implementing two constructors in both subclasses. The solutions were replaced the same way as exercise 1, but the start code included an extra empty constructor that was not in the solution project. This is to prevent subclass compiler warnings concerning default constructors, and it is obviously impor-

⁹https://en.wikipedia.org/wiki/Convention_over_configuration

¹⁰<https://docs.asciidoc.org/asciidoc/latest/sections/titles-and-levels>

tant to include this constructor in the start code so that students do not think they may have done something wrong to cause errors. To be able to handle such situations, the `Remove` annotation was expanded to include the option of specifying which of the generated projects it should be removed from. The default option is still to remove it from both the start code and solution.

The original exercise description for exercise 2 is not split into sub-exercises, even though there are two separate subclasses that should be implemented. This is a problem for FREJA since it presumes a one-to-one relationship between a single exercise and a Java class. There is an argument to be made to change the implementation so that an exercise can be linked to several classes, but the problem can be simply solved by splitting the exercise into two sub-exercises. We could not think of any immediate disadvantage or limitations by enforcing the user to adhere to the one-to-one relationship rule, which resulted in us not changing this behavior. Instead, this keeps the implementation of FREJA simpler, while the assignment description will have a clearer structure.

After the entire solution project was annotated to accurately generate projects that mimic the original artefacts, another test was done to examine if the inclusion of the popular version control system, Git, would complicate matters. A Git repository was created for the annotated solution project, which was located in the same folder. Executing FREJA at this point would also copy over the hidden Git folder to both generated projects. There are clear differences between the content of the files in the generated projects, and the annotated solution project, suggesting that this is always an unwanted behavior. To accommodate for this, FREJA was changed to always ignore any Git folders in the annotated solution project when copying over files to the generated projects. Another change was added to ignore any potential Git folders in the generated projects when deleting files in the target folder, since this caused FREJA to crash when attempting to modify files that were restricted by the operative system (e.g., the `.git` folder).

Other drawbacks of the framework included not being able to remove the POM file from the generated projects, even if that was explicitly stated in the configuration. This was due to an oversight in the implementation. Also, the generated description templates are only available in English, even though the original description is in Norwegian. There is also no way to connect an exercise to a certain unit tests through FREJA's annotations. Because of this, there are no AsciiDoc attributes in the assignment description that can refer to test information. Every exercise have tests that accompanies them, and three exercises explicitly referred to their tests in the exercise description. This demonstrates an area where FREJA could further reduce hard-coded references in the assignment description.

Out of the 85 source code references in the original assignment text, 50 of them were automatically created as AsciiDoc attributes by FREJA. Their substitutions was also used in the template descriptions. Several more attributes were also generated, although not all were needed to match the original description. The other 35 references that did not have attributes created for them were mostly similar to each other, with 13 of them being method parameter references, 3 referred to the root class in subclass exercises, another 13 referred to methods, variables, or some other part of the `Innlegg` class of exercise 1. The

last 6 were mostly due to exercise 7 not really being an exercise, and therefore the files for it were not annotated.

Testing FREJA on this assignment also highlighted many benefits and useful functionalities of the framework. First of all, the start code did not contain any classes that were unrelated to the exercises, such as the original `TODO` class to infer names of methods and classes. Also, the order of the exercises within a class did not matter as well. For example, exercise 2 can be listed before exercise 1 in the same class without any problems, and the structure and ordering of the generated descriptions will also not be affected by this. Having additional annotations unrelated to FREJA on the same code structure as an `Exercise` annotation proved to be unproblematic as well. Executing the framework to generate the artefacts took about one to four seconds each time.

Implementing FREJA as a Java library allows the user to take advantage of the IDE's built-in functionality for Java. This includes syntax highlighting, code completion, and much more, which was very helpful when annotating the solution project. Installing an AsciiDoc plugin for the IDE was also useful when manually editing the assignment description, especially the smart suggestions and auto-completion of attributes. The custom run-time error messages were also immensely helpful to locate and understand the issue whenever a problem occurred.

The total amount of Java expressions was measured for each project in the assignment, including both the original projects, and the FREJA projects. This is to quantitatively compare the amount of code that is manually produced and maintained using different methods of assignment creation. Figure 7.1 shows a comparison of the number of expressions between each project.

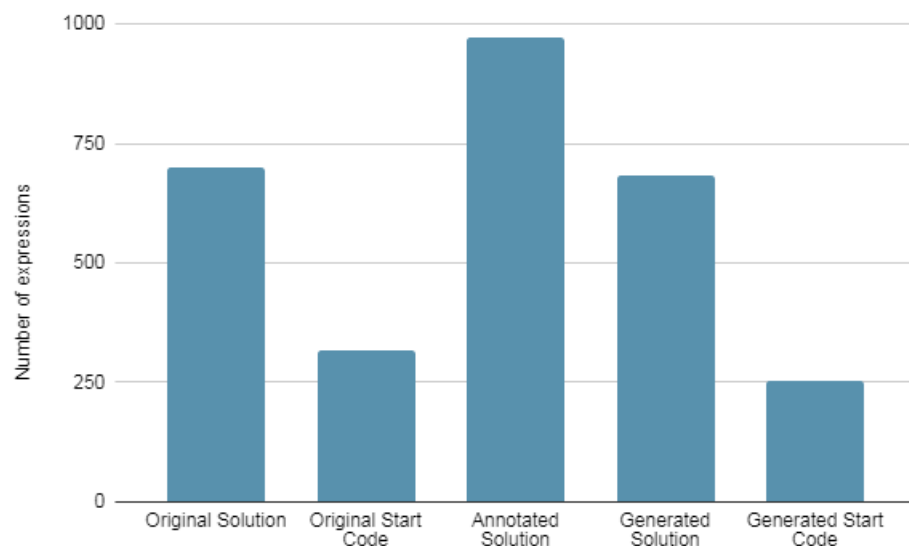


Figure 7.1: Comparison of the amount of expressions between the different projects.

Java expressions¹¹ are small code fragments like literals, method calls, and annotations. This felt like an appropriate unit of measurement that focuses more on the code that make up the logic of the program, compared to something like the lines of code metric, that would include things like comments and empty lines. Furthermore, FREJA annotations does not need to be declared on a separate line, potentially leading to some annotations not being counted as additional code using lines of code as a measurement.

The annotated solution project and the generated projects did contain the test files for the exercises, but these were not counted since they were not included in the original projects. The original solution project contained 701 expressions, while the original start code project had 317 expressions. The annotated solution had 49 FREJA annotations with 972 total expressions. The generated solution and start code projects contained 683 and 252 expressions, respectively. The reduction in expressions in the generated projects compared to the original ones comes from the removal of the `TODO` class, since it was no longer needed due to reasons discussed earlier.

384 expressions spread across 11 files were manually removed from the original solution project to create the start code. In total, 1018 expressions and 22 files in two separate projects needs to be manually maintained when the assignment is created the old traditional way. Although 271 additional expressions related to FREJA needed to be included in the annotated solution, there is only one project with 10 files and 972 expressions that need to be maintained manually. That is a 55% reduction in manually maintained files, and 4.5% reduction in manually maintained expressions.

7.2 A Second Evaluation of FREJA

Again, the results of this evaluation are uploaded to GitHub. One repository contains the annotated assignment solution¹², and a different repository has the generated output¹³. For the second evaluation of FREJA, an updated version of the framework was used to test it. This update included many changes from the first test, both new functionality and bug fixes. The most important changes were:

- A new annotation, called `DescriptionReference`, that can be used on any code structure to create arbitrary code references in the assignment description. Using the annotation on a piece of code will create an attribute in an exercise description that has the annotated code as its value. The annotation has an annotation element, called `exercises`, that takes an integer array as a value. The integers in this array should be root exercise numbers from an assignment, to specify which exercise description file to create the attribute in. For example, setting `exercises` to `{1,2}` would create an AsciiDoc attribute in the description file of exercise 1 and 2.

¹¹The exact definition of an expression in the Java grammar can be found here: <https://docs.oracle.com/javase/specs/jls/se7/html/jls-18.html>

¹²<https://github.com/ErlendBerntsen/dat100-javainnlevering2-solution-annotated>

¹³<https://github.com/ErlendBerntsen/dat100-javainnlevering2-output>

- The `Remove` annotation has an added annotation element, called `removeFrom`, that can be used to specify which of the generated projects the annotated code will be removed from. For example, setting `removeFrom` to `TargetProject.START_CODE` will remove the annotated construct only from the start code project, not the solution. Leaving it unspecified will use the default value, which is to remove it from all projects (this is how it worked previously).
- POM file can be removed, or ignored rather, from the generated projects.
- Fixed bugs with the generated descriptions.
- Many other general bug fixes, mostly concerning comments.

The first evaluation was done using IntelliJ IDEA, which is also the IDE that is used to develop FREJA. The first test went without any noticeable IDE issues, mostly due to them being discovered and accounted for during the development of FREJA. Eclipse is the second most popular Java IDE, and together with IntelliJ they account for 83% of the market share for Java IDEs [6]. The second evaluation was done using Eclipse to test FREJA in a different environment, which makes it easier to assess the fulfilment of the *Accessibility* requirement. Being synergetic with the IDEs that the majority of Java developers uses would certainly help in this aspect. A different assignment from the DAT100 2021 fall semester was used to test the framework a second time. Another benefit of using Eclipse for the evaluation is that all DAT100 assignments are created using Eclipse. Therefore this second test will give insight into how well the framework works with the same development environment as the assignment was created in.

7.2.1 Details of the Assignment

This assignment is much smaller than the previous one, but still an assignment that should be solved in groups. There are only two Java files that should be implemented, one for each exercise in the assignment. Two additional Java files are also included, containing tests for each of the exercises. There are only two projects for the assignment, one for the start code, and one for the solution. The start code is again shared to a public GitHub repository¹⁴, which the students must fork and clone. The solution repository¹⁵ was publicly shared to the students after the assignment deadline, so that they could compare their own solutions to the intended one. The description text for the exercises differ from the previous assignment by being located in two entirely separate repositories, one for the first exercise¹⁶, and one for the second exercise¹⁷, as opposed to being in one large file. This is due to exercises being part of two different voluntarily weekly exercise sets that were handed out a few weeks before the assignment. The text is still written in GitHub flavored markdown.

¹⁴<https://github.com/dat100hib/dat100-javainnlevering2>

¹⁵<https://github.com/lmkr/dat100-javainnlevering2-solution>

¹⁶<https://github.com/dat100hib/dat100public/blob/master/programming/jplab5/JP5.md#obligatoriske-oppgave-o1-a>

¹⁷<https://github.com/dat100hib/dat100public/blob/master/programming/jplab6/JP6.md#obligatorisk-oppgave-o1-b>

There is one less project compared to the assignment in the previous evaluation, since the unit tests are bundled with the start code, instead of being stored in a separate repository. The start code repository has a README file that explains different details about the assignment, such as how to import the start code locally, how to give access to other group members, submission details and hyperlinks to each exercise description. There are four references to the source code in this README file, all of which reference the file name for each exercise and their test classes. The structure of the project is identical between the start code and the solution project, with both having one folder for the Java class and its test class for the first exercise, and another folder for the second exercise.

The purpose of the assignment is to teach the students about arrays, with exercise 1 focusing on one-dimensional arrays, while exercise 2 covers two-dimensional arrays. There are a total of 15 sub-exercises, with exercise 1 being split into eight sub-exercises and exercise 2 split into seven, with the last two being optional. The task for each sub-exercise is to implement a single method. The start code for each sub-exercise follows the same pattern, which is to keep the skeleton of the method, while replacing its body with a throw statement. This throw statement is similar to those in the previous assignments, though in this case the error message is hard-coded, and not created with a dedicated helper class. For example, one exercise solution from the assignment can be seen in Listing 7.2, and its corresponding start code is shown in Listing 7.3.

```
1 public static int[] reverse(int[] array) {
2     int length = array.length;
3     int[] newArray = new int[length];
4     for (int i = 0; i < length; i++) {
5         newArray[i] = array[length-1-i];
6     }
7     return newArray;
8 }
```

Listing 7.2: The solution to an exercise for reversing an array.

```
1 public static int[] reverse(int[] array) {
2     throw new UnsupportedOperationException("reverse is not
3     implemented");
}
```

Listing 7.3: The start code for the reverse method.

The description for each sub-exercise also follows the same pattern. They are short and concise, and tells the student to implement a specific method, by referencing the method definition in the text. Afterwards is a small explanation of what the method should do. The description for the above exercise can be seen in Fig. 7.2.

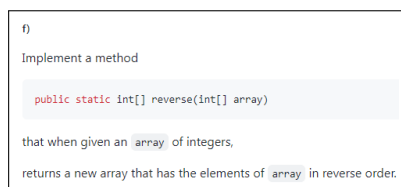


Figure 7.2: The description of an exercise where the goal is to reverse an array.

In total there are 28 references to the source code of the assignment (32 if including the README file in the start code repository). 14 of these are references to various method parameters in the exercises. All of these are copied from the source code and pasted into the description text. The disadvantage of this can be seen in the description of the fourth sub-exercise for exercise 2, as seen in Fig. 7.3. The text in this description refers to the parameters `a` and `b`, while the method definition has the parameters `mat1` and `mat2`.

```

d)
Implement a method

public static boolean areEqual(int[][] mat1, int[][] mat2)

that decides if two matrices, given by the parameters a and b, are equal.

```

Figure 7.3: The code reference to the method definition is incorrect.

Looking at the source code of the exercise, as seen in Listing 7.4, reveals that it is the method definition in the description that is incorrect. There are two such mistakes in total for the assignment, most likely due to changes to the code after the description texts are created.

```

1 // d)
2 public static boolean areEqual(int [][] a, int [][] b) {
3     throw new UnsupportedOperationException("areEqual is not
4     implemented");
5 }

```

Listing 7.4: The source code of the exercise shows that correct parameter names are `a` and `b`.

7.2.2 Testing and Results

We also had access to all the artefacts for this assignment as well, so we knew exactly how they should look like when finished. The same approach was used as for the first test, by cloning the original solution project and injecting the necessary annotations. As mentioned, DAT100 assignments are created in Eclipse so there was no trouble importing the assignment, however, it still needed to be converted to a Maven project to use FREJA. Eclipse fortunately provides the ability to easily convert Eclipse projects to Maven projects. Once converted, the `Exercise` annotation seen in Listing 7.5 was added to the method for the first sub-exercise for exercise 1. The FREJA Maven plugin was then executed for the first time for this assignment.

```

1 @Exercise(id= {1,1}, transformOption= TransformOption.REPLACE_BODY)

```

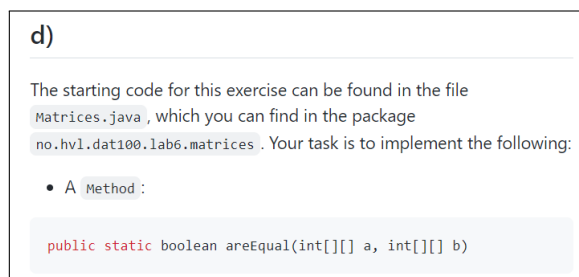
Listing 7.5: The `Exercise` annotation for the very first sub-exercise.

Afterwards, the generated projects were opened in Eclipse to ensure the generated start code was created as intended. The method body in the start code was replaced with something along the lines of what is seen in the `areEqual` method in Listing 7.4 from before, which is correct. While checking that the

code in the generated solution project was also correct, a new issue was discovered. Eclipse gave compilation errors that complained about the Java class for the exercise was already defined. It is true that the same class is defined both in the solution project and the start code project, but this problem never occurred while using IntelliJ. The root folder for both of the generated projects was selected when opening them as a project for the first time in Eclipse. To try to fix this issue, an attempt was made at opening them as two separate projects instead, but Eclipse would not allow this for some reason. In the end, the cause of the problem was that both projects were listed as source folders on the build path for the root folder. To avoid the issue, only one of the projects can be listed as the source folder at a time. Luckily, it is very fast to change the defined source folder in Eclipse. The rest of the sub-exercises in exercise 1 were almost identical, i.e., the task is to implement a single method, and the start code follow the same pattern. This made annotating the rest of the exercises very easy since the annotation from Listing 7.5 above could be copy and pasted each time. Only the `id` value of the annotation needed to be changed.

An Eclipse AsciiDoc plugin¹⁸ was installed to be able to work with the generated exercise descriptions. This plugin worked fine to write regular AsciiDoc text, but lacked useful attribute functionality compared to the IntelliJ plugin, such as attribute suggestions and substitution preview. Another problem occurred when trying to write the description in Norwegian (due to the special letters æ-ø-å). This was fixed by setting the text encoding in Eclipse to UTF-8 instead of the default CP-1252. Although IntelliJ provided a somewhat smoother experience overall, this test showed that FREJA manages to work with Eclipse just fine.

The generated template descriptions for each exercise were a little more verbose than the original. For example, the original description for one of the sub-exercises was seen in Fig. 7.3 on the previous page, while the FREJA generated description of the same exercise can be seen in Fig. 7.4, which includes more boilerplate information about the location of the exercise. However, the referenced method definition in the generated description does not include the wrong parameter names as the original did.



```
d)

The starting code for this exercise can be found in the file
Matrices.java , which you can find in the package
no.hvl.dat100.lab6.matrices . Your task is to implement the following:

• A Method :

public static boolean areEqual(int[][] a, int[][] b)
```

Figure 7.4: An automatically generated description template.

Editing the template to emulate the original is easy since the unnecessary information can quickly be removed, but the explanation of the method behavior in the original description references the parameter names. The new

¹⁸<https://marketplace.eclipse.org/content/asciidoc-editor>

`DescriptionReference` annotation was used on both parameters to able to reference them without hard-coding them in the text. This ensures that there can not be any mismatch between the method definition and the parameter references, such as the mistake in the original description.

All 28 code references in the original exercise descriptions have attributes created for them using FREJA. 14 of these were created automatically and the other 14 references needed 11 `DescriptionReference` annotations to produce the necessary attributes. The result of this is that the FREJA exercise descriptions corrected both code references in the original descriptions that were inconsistent with the source code. These attributes will also automatically update the description again if the source code is at some point changed.

Although the `DescriptionReference` annotation ensures more consistency between description text and source code, it still had room for improvement. The attribute key name for a description reference attribute is automatically created, which is just `DescriptionReference_` followed by a number, e.g., `DescriptionReference_1`. An exercise description file can have multiple description reference attributes, where each have a different suffix number that counts upwards from 1. The numbering is decided by the order that the annotations are parsed in. This was a little confusing for exercise 1 which had 8 description references, with some even having the same attribute value. This was due to some methods sharing the same name for certain parameters. It was difficult to figure out which attribute corresponded to which value when using them in the description text, without double-checking the attribute definition and the source code. A new annotation element, called `attributeName`, was added to the `DescriptionReference` annotation to fix this, which allows the user to define the attribute key name for a specific description reference. At first this was an optional setting, defaulting back to the previous method if not specified. This was quickly changed to being required, since relying on the parsing order of the annotations to create the numbering for the attribute names is very unintuitive for the user. There is also potentially a big problem in the case that a new `DescriptionReference` annotation is added at a later stage when creating an assignment, by possibly creating inconsistencies in older description reference attribute names if the new annotation is parsed before the old ones.

Lastly, the number of Java expressions were also measured for this test, and a comparison of these numbers between each project can be seen in Fig. 7.5 at the following page. All projects had only 4 files, which includes test files since those were part of the original solution and start code. For this assignment, the generated projects managed to match exactly the same amount of expressions that were in the original projects. Both the original and generated solution had 847 expressions. The original start code, as well as the generated, had 425 expressions. The FREJA annotated solution contained 964 expressions, an increase of 117 expressions, or 13.8%, compared to the original solution. The increase in expressions is due to the 25 FREJA annotations. That is about an additional 8 expressions per sub-exercise, on average. In total, there are 1272 expressions, 8 files, and two projects, that needs to be manually maintained with the original assignment. With FREJA, the amount of manually maintained expressions are reduced to 964, contained in one project with four files. That is a 24% reduction in expressions, and a 50% reduction in files and projects.

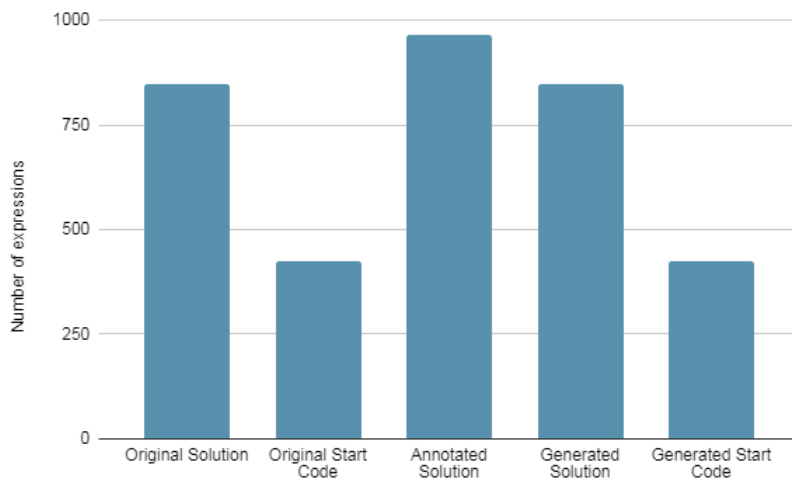


Figure 7.5: Number of expressions among the different projects.

7.3 Evaluating a Language Agnostic Framework

The prototype of the language agnostic framework was tested on the same DAT100 assignment that FREJA based most of its design and implementation on. Since the implementation of the prototype is much simpler, it can only handle one programming file at a time, instead of a complete project. Most of the examples given here are taken from the first exercise of the assignment, which concerns implementing field variables, a constructor, get- and set methods, and other simple methods for a Java class. The language agnostic framework is compared against FREJA and the conventional method for this first test, only to evaluate its performance as an assignment framework in the context of other methods. Afterwards, its generic capability is evaluated by testing the prototype on exercises written in different programming languages.

7.3.1 Using the Language Agnostic Framework DSL

With this method, the solution file is easily imported into MPS with a single button click that generates a new `File` instance. Marking the start and end of a solution is fast and easy with keyboard shortcuts, or by using the right-click popup menu. Being able to include any line as part of a solution gives much flexibility, but opens the possibility for generating syntactically incorrect programs, e.g., accidentally including an additional curly bracket as part of a solution, causing an imbalance in the brackets in the generated file. This may be hard to spot since the preview of the generated file does not give any warnings or red lines about incorrect syntax. Additionally, MPS has syntax highlighting for Java programs, even in previews, so this may give a false sense of security that the program is syntactically correct.

Creating replacement code is also straightforward, and the ability to preview the generated code makes it easy to ensure the new code is created as intended. The code that is replacing a solution may need accompanying import statements so that the generated file will compile, which is an issue if the solution and

replacement code is in different files. This is the case for the first sub-exercise of the assignment, but this dependency relationship is not possible to portray in the DSL currently. Though, a workaround is to include the import statements in the solution file before the generation. Once the user is happy with preview, the file can be created with a simple right-click.

7.3.2 Using FREJA

Instead of importing the solution file into another program, this method works directly on the same file by expanding it with annotations. These annotations are similar to the concepts in the DSL, denoting meaning related to programming assignments. Defining what should be part of a solution is more limited since the file still needs to be syntactically correct to be able to compile. This is done by either annotating entire method bodies, or by wrapping the solution between specific statements that will be recognized later in the transformation phase.

Everything that should be implemented in an exercise needs to be annotated individually, which can create a more bloated file, and most likely take more time than the DSL method. For example, the field variables can be grouped together and be thought of as one thing to implement with the DSL, but needs a separate annotation for each field variable when using FREJA.

Exercise annotations in FREJA must reference an identifier when replacing a solution, but there is no way to guarantee that the specified identifier actually exists before running the program. As shown earlier, FREJA does not complete the generation in this scenario, but instead throws a custom exception at run-time, explaining the issue in detail. In contrast, the DSL programmatically changes the editor to only give the possibility of referencing a `SolutionReplacement` when the `transformOption` is set to `REPLACED`. The reference also refers to the concept node directly, instead of through an intermediate string identifier, ensuring that it actually exists. Aside from that, replacing solutions works practically the same way.

7.3.3 Using the Conventional Method

The basis of this method is to copy the entire solution file to another file, and edit it afterwards. This is done by manually removing the implemented exercise solutions from the new file copy, or replacing the solutions with some different code. An advantage of this is having full control over how the start code should be, without any of the limitations from the previous methods. There is also the benefit of working with the output file directly, since this easily confirms whether the start code can be compiled or not, instead of having to wait after generating the start code file to confirm it does work as expected.

7.3.4 Comparing the Different Methods

This section highlights the differences between each method in the context of the requirements that were described earlier in the *Requirements* section of chapter 3. With the language agnostic framework, the actual solution file would be created independently before importing it into MPS, since the DSL does not

provide syntax highlighting, code completion or other similar functions for the source language of the assignment that other IDEs might do. This means that refactoring the solution file after importing it would require re-importing it into MPS, and edit it from scratch again, or manually refactor the MPS solution file directly. There is no possible way MPS can give any automatic refactoring help, since the underlying assignment code is represented as regular text strings within the DSL. Additionally, manual refactoring creates a big opportunity for causing inconsistencies between the solution and start code, and re-importing would overwrite any previous work done on the MPS file. The latter option is obviously a big time waste if the synchronization between an updated solution file and an updated MPS file simply ignores the modified content of the MPS file.

In terms of bidirectional transformations, restoring consistency through a re-import, where the old MPS file is taken into consideration, is more similar to a *put* function than a *get* function. As discussed, defining and implementing a satisfying *put* function can be incredibly difficult. FREJA restores consistency between artefacts in a manner comparable to a *get* function, while the language agnostic framework did not manage to do this, since the solution is developed outside MPS.

One of the biggest motivations behind creating a programming assignment framework was to remove such inconsistency and redundancy issues. The description for any exercise must also be independently created using the DSL, which is no better than the conventional method of creating assignments. This is due to the loss of meta-information by having a generic framework, since it has no knowledge of the syntactical structure of the underlying program. It is clear that this prototype has failed to meet the *Consistency* requirement, and it is no better than the conventional method concerning the *Non-redundant* requirement either.

On the other hand, the main advantage of FREJA is that it fulfills these two important requirements. Consistency is semi-automatically guaranteed whenever changes are made, and there is only a single source of information, since everything is (mostly) contained within one project that generates the other assignment artefacts. Both the annotation-free solution and start code is generated, therefore it is necessary to update information in only one place when changing an assignment. Additionally, the exercise descriptions are generated based on a description template, by receiving meta-information from the annotated code. This includes information about what to implement, e.g., a method or a field variable, its name, file location, and so on. Propagating changes from the source project to the other assignment artefacts is done by simply executing the FREJA Maven plugin, which will promptly restore consistency between the artefacts again.

This difference in performance between each assignment development method, concerning consistency and redundancy, is mostly due to how each method create assignment artefacts. With each method, there are different relationships between assignment artefacts, since some artefacts are dependent on information from others. For example, the assignment solution must be created first with the DSL method, since the MPS file is completely dependent on it. A comparison of how each assignment development method create assignment artefacts can

be seen in Fig. 7.6. Artefacts that are used to create other artefacts, either partially or completely, are marked with an outgoing arrow into the other one. If the line is dashed, it is a manual creation, otherwise, the artefact with an incoming arrow is generated. An artefact that do not have any arrow to it is created in isolation.

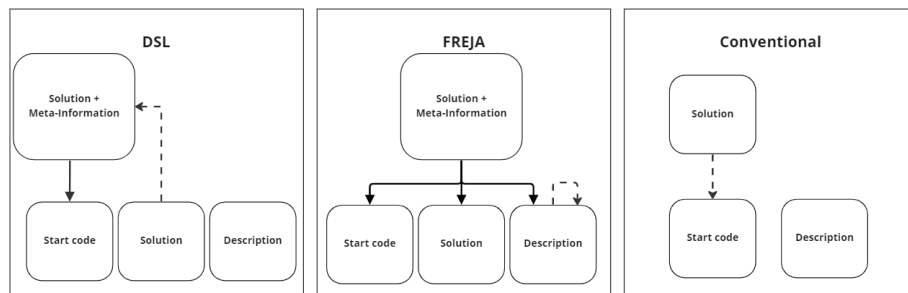


Figure 7.6: Comparison of how assignment artefacts are created using different assignment development methods.

The DSL does not entirely satisfy the *Accessibility* requirement either. Although it is independent of the underlying operating system, it is completely dependent on the MPS IDE. On the other hand, the other methods mostly use tools from the Java language, which the user most likely already have at their disposal. Additionally, both methods are independent of the IDE being used, as well as the operating system. Concerning the *Dependencies* requirement, FREJA does rely on Maven to execute the program, and an AsciiDoctor engine to process AsciiDoc files. Though, the latter is easily accessible as an IDE plugin, and both dependencies are also platform independent, so they do not sacrifice the accessibility of FREJA. JavaParser is not counted as a dependency since it is bundled with the installment of FREJA. The conventional method clearly does not have any external dependencies, while the DSL method depends on MPS, as mentioned.

The *Learning ability* requirement is where the DSL method come up short compared to the other methods. This is due to the user needing to learn MPS, which differs greatly from regular text-based editors, and has a steep learning curve. However, once the user is comfortable with MPS, the language itself should not be difficult to learn. It should be about as much overhead as learning how to use the annotations properly with FREJA. FREJA is not difficult to learn either since it is so small, and it has meaningful names for annotations, annotation elements, and other configuration settings. The entire framework that the user interacts with is composed of only six annotations, seven annotation elements, two Java enums, and three different Maven configuration options. The DSL is also small in size, and has only 6-7 important language concepts, one enum, and an additional concept for import configuration.

FREJA also necessitates learning Maven, but does not require a deep knowledge of it. The user must also learn the AsciiDoc language if they want to manually expand the description files. Learning a new markup language should not be a difficult task if the user is already familiar with one, such as the widespread

Markdown language. Furthermore, FREJA generates a template description that demonstrates how to use different syntax and common features of AsciiDoc. The conventional method thrives the most in this aspect, as it requires nothing new to be learned.

It is not only negatives for the DSL method. MPS have almost endless possibilities when it comes to customization of the editor, giving the creator of a DSL an opportunity to make the language as user-friendly as possible. Marking solutions, specifying options and any potential replacements, is effortless with the DSL because of this. FREJA needs to comply with the Java grammar, which makes this process a little more rigid. On the other hand, FREJA has an enforced syntax, which benefits from Java's type safety and the underlying IDE functionality pertaining to Java's type system. The framework also provides detailed explanations whenever there is an error, allowing the user to quickly locate the issue and determine its cause. In comparison, the DSL is just a prototype of a language agnostic framework, and implementing a fully fledged type-system with extensive error handling has not been prioritized, but MPS does provide the tools to do this. FREJA also incorporates the *convention over configuration* paradigm wherever possible, to reduce the amount of editing that needs to be done. As for the conventional method, it is very easy to use, since it mostly boils down to manually copy-pasting code, and removing certain pieces of code. However, it is an undeniably tedious and error-prone way of creating programming assignments. In the end, all three methods satisfy the *Ease of use* requirement, with no real distinction between their performances in this regard.

The *Effectiveness* requirement is arguably the most important one, and is where FREJA outshines the other methods. Both time and effort are saved by generating a description template directly from the source code, with semi-automatic updates to any code references in the text. In comparison, both the DSL and the conventional method has to write the assignment description manually from scratch, and also manually update it if there is any change in the referenced source code. FREJA can also handle an entire assignment project, as opposed to a single file at a time with the DSL. In addition, the evaluations of FREJA presented quantitative results that highlighted the reduction of elements that needed to be manually maintained, such as the number of projects, files, and expressions. As discussed, the DSL needs to develop the solution in isolation before importing it into MPS, and therefore there is no such reduction compared to the conventional method. In fact, the DSL method might cause an increase in this number. The effectiveness of FREJA is best felt whenever changes to an assignment needs to be made, especially late in the development.

7.3.5 Other Programming Languages

The main purpose of the DSL was to be more generic than FREJA, i.e., it can be used with other programming languages as well. We did not have the same level of access to non-Java programming assignments, so we used HackerRank¹⁹ to find programming exercises in different languages. Simple programming exercises for Python and Haskell was used to test the broadness of DSL. Python is an object-oriented, dynamically typed language that relies on indentation to

¹⁹<https://www.hackerrank.com>

define structure in a program. A small rewrite of the import and export logic was needed to ensure that the indentation was properly kept, both when importing the programming file and generating the start code. After that, the DSL was just as effective as with Java. On the other hand, Haskell is a functional programming language with a very different style and design compared to the other languages. In the end, it is just lines of text, and the effectiveness of the DSL was about the same as with the other languages. An example of using Haskell and Python exercises with the DSL can be seen in Fig. 7.7. The figure shows two solutions to a simple HackerRank exercise²⁰, where the goal is to sum up the numbers in a list of integers. Both solutions are outlined by the DSL and replaced with a comment, just how the start code to the exercise is given. On the left is the solution and start code written in Python, while Haskell is on the right side.

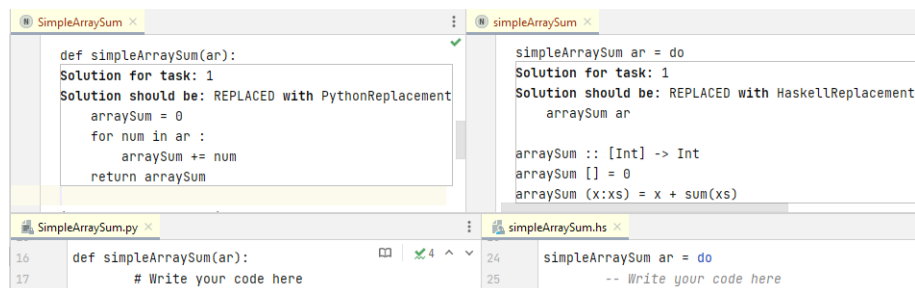


Figure 7.7: The solution and start code for an exercise in two different languages. Python is on the left and Haskell on the right.

Every programming language is presented in the MPS editor with no distinction between syntactic concepts within the language. This makes it impossible to generate descriptions the same way FREJA does. There could be a chance of implementing this by including the AST of the underlying program, as with the initial design, but it would still be incredibly difficult finding a general solution for all programming languages. This syntactic indifference also makes it unreasonable to develop the solution to an exercise within the DSL.

Another possibility is using the Language Server Protocol (LSP)²¹, which is a standardized protocol for bringing editorial features such as auto complete, code suggestions, and go-to definition, to different editors and languages with minimal effort. Support for many languages have already been implemented, but the list of supporting tools is not that big yet. None of JetBrains's IDEs are supported, including MPS, so the integration of LSP would be in the hands of the developers at JetBrains, or as a community developed 3rd party plugin, if possible. LSP is also based on positions in text files, which may be difficult to integrate with MPS's way of treating files like ASTs.

Developing programming exercises within MPS, using LSP or not, forces the user to also sacrifice traditional text-based development for MPS's code com-

²⁰<https://www.hackerrank.com/challenges/simple-array-sum/problem>

²¹<https://microsoft.github.io/language-server-protocol>

pletion²², which may be too unconventional for some. In short, code completion forces the user to explicitly select the specific language concept to insert from a popup menu, when editing a file or program. Additionally, the program for an exercise cannot be compiled in MPS either, forcing the user to use another IDE anyway if they want to check that the program runs as expected.

²²<https://www.jetbrains.com/help/mps/auto-completing-code.html>

Chapter 8

Conclusions and Future Work

This chapter gives a brief summary of the entire thesis by recapitulating the most important details from each chapter. The main results from our tests are condensed into tables to quickly summarize the performance of each framework. Following this is an analysis of those results in the context of the research questions defined earlier, ending with a discussion of the main conclusions of this thesis. Lastly, there is a section detailing the next logical steps forward for FREJA, and a description of ideas and other relevant work that could be interesting to pursue further.

8.1 Summary

The inspiration for this thesis is rooted in the subpar methods that are currently used for creating programming assignments. Developing a programming assignment is composed of developing several smaller artefacts that make up a complete assignment, such as a solution project, a start code project, and an assignment description that explains the exercises of an assignment. Even though these artefacts are deeply related to each other due to the numerous references and repetition of code between them, they are still completely independent. Maintaining consistency between them is tedious and error-prone, since any potential modification to an assignment may need to be manually reflected in multiple artefacts.

There is a lot of related work on programming assignments in an educational context, at least in the area of feedback and automatic evaluation of student solutions. To our knowledge, there has not been any previous work done on streamlining the process of creating assignments that are specifically aimed at computer science majors. This thesis has taken a first step in this regard, by developing a new framework that can centralize assignment development to a single source, both to reduce redundancy, and ensure consistency between assignment artefacts. The design of the framework is based on previous assignments from courses at the university level, while some of the functionality has been imple-

mented by identifying the needs of assignment creators. Requirements such as effectiveness, consistency, non-redundancy, accessibility, having few dependencies while still being easy to learn and use, were imposed on the framework to increase its chance of success and adoption in a real-life setting.

The result of this work is FREJA, a framework for creating Java assignments. The implementation for this framework draws inspiration from several areas, such as Model Driven Software Engineering, the Single Source of Truth architecture, Bidirectional Transformations, and earlier work on code generation, code transformation, and code documentation. FREJA is centered around the idea of having custom Java annotations that can mark code fragments with special meaning within a programming assignment. The development of a programming assignment with FREJA is limited to a single project, which contains the assignment solution intertwined with FREJA annotations. Annotations are also used to define meta-information with transformation rules, which are used to specify how to generate the regular solution project, start code, and assignment description.

A small prototype of another framework was also implemented, to determine the feasibility of creating a programming language agnostic version of FREJA. This prototype is a DSL, created using the MPS language workbench. The DSL can in theory handle any regular text-based programming language by treating a programming file simply as a collection of text lines. Code fragments can be marked by selecting multiple lines, and transformed by utilizing special language concepts that correspond to FREJA's annotations.

FREJA was evaluated using two earlier assignments from the introductory programming course DAT100. The goal of these tests were to mimic the original artefacts, exclusively using the functionality of FREJA, while also limiting redundancy and ensuring consistency between assignment artefacts. The ensuing results were discussed to gauge how well FREJA performed in this aspect. These tests also served as a way of improving the framework in an iterative process. Shortcomings or other lacking features that were identified by these tests led to implementation changes to further enhance FREJA. The language agnostic prototype was tested with exercises from multiple languages, such as Java, Python, and Haskell, to determine its generic capability. The process of creating programming assignments using FREJA, the DSL, and the conventional method, were all compared against each other in context of our requirements for an assignment development method. This presented both pros and cons of each method, though FREJA came out ahead for most requirements.

8.2 Conclusions

Testing FREJA on multiple assignments showed that it can handle differently structured assignments with different types of exercises. Although the first evaluation identified some shortcomings of the framework, an update to the implementation addressed most of these issues. However, FREJA did not manage to completely recreate some of the original exercise descriptions, but this was mostly due to the unorganized structure of them. We think it is reasonable that users must adhere to some ground rules, which in this case will enforce a clearer

structure of assignment descriptions. Thus, we did not change the implementation to allow for complete freedom in this regard. As for the second evaluation, FREJA could fully emulate the original artefacts, and link them together so they remain consistent. There was no missing functionality for FREJA to recreate the second assignment, but small changes were made to improve the ease of use.

The requirements of FREJA are mostly fulfilled by looking at the results of these tests. A summarization of how each assignment development method fulfill the requirements can be seen in Tab. 8.1. FREJA is clearly the method that best satisfies the prescribed requirements, especially the first three. These are arguably the most important requirements to meet, since they essentially capture the goal of this thesis.

Requirement	FREJA	DSL	Conventional
Effectiveness	High	Ok	Ok
Consistency	Semi-automatically guaranteed	Not guaranteed	Not guaranteed
Non-redundant	Yes	No	No
Accessibility	Good	Poor	High
Dependencies	Maven AsciiDoctor	MPS	None
Learning ability	Easy	Difficult	Very easy
Ease of use	Easy	Easy	Easy

Table 8.1: Comparison of how the different methods meet each requirement.

The amount of files and projects that needed to be manually maintained using FREJA were halved in both evaluations, as well as a reduction in Java expressions. With the help of AsciiDoc attributes, the amount of hard-coded references to the source code in the assignment description was massively decreased, and outright nullified for the second test. Table 8.2 show the reduction in manually maintained components for each evaluation.

First Evaluation			
Component	Original	FREJA	Reduction
Projects	2	1	50%
Files	22	10	55%
Expressions	1018	972	4.5%
Hard-Coded References	85	30	65%
Second Evaluation			
Component	Original	FREJA	Reduction
Projects	2	1	50%
Files	8	4	50%
Expressions	1272	964	24%
Hard-Coded References	28	0	100%

Table 8.2: Reduction in the number of components that must be manually created and maintained for both evaluations.

These reductions make it much simpler to perform changes to an assignment. All assignment information is constrained to a single source, and consistency between the different artefacts can be restored through a single command. The benefit of this could be seen in the assignment description for the second test, where FREJA removed the inconsistent code references. By identifying the different artefacts in a programming assignment, and the relationships between them, we have found an answer to the first research question:

RQ1: How can we centralize programming assignment development to a single source to ensure consistency and avoid redundancy?

The key to this answer is to limit the assignment development to only the solution project. This is because the entire start code can be extracted from the solution project. Code references in the assignment description can also be solely created from the solution project. However, the solution project must be expanded with additional meta-information that allows for generating the other assignment artefacts in the manner that the user wishes. Centralizing assignment information this way encapsulates the Single Source of Truth architecture, and guarantees a semi-automatic consistency between the assignment artefacts. FREJA has also taken inspiration from the simplicity of restoring consistency with *get* functions in asymmetric lenses. However, consistency insurance is also equally contingent on a suitable solution for the second research question:

RQ2: How can we automatically, or semi-automatically, generate the necessary programming assignment artefacts from that source?

For FREJA, this problem was predominantly solved by the Java annotation API. Custom annotations allows the user to mark code fragments that carry special meaning within a programming assignment. Annotations can also store small amounts of data, granting the user the means to specify transformation rules for generating the start code and assignment description. Additional configuration settings are also specified in the Maven POM file.

The other piece of the puzzle is having some way of parsing an annotated assignment, transforming it, and finally generating the desired output artefacts. Fortunately, there has been a lot of work done already in regards to parsing source code, so there was no need to reinvent the wheel in this aspect. Java-Parser provided all the necessary tools to do this for Java files, along with the ability to generate Java files from AST objects. It was mostly the transformation process that needed to be developed from the ground up. Even though the generation process is not fully automatic, since the user needs to manually execute the framework to start the generation, it is not necessarily a bad thing. The manual labor required to execute the framework is composed of writing a single command in the terminal, and 1-4 seconds of waiting while the program runs, which is negligible in the grand scheme of things. Also, a semi-automatic process lets the user start the generation whenever *they* think it makes sense, instead of a fully automatic system that might start the generation process with an incomplete or erroneous input assignment.

The evaluations for FREJA proved that it can successfully be used to create Java assignments, at least very well for the DAT100 course. Testing the framework on more assignments from different courses would likely reveal new issues or lacking features that could be improved upon. As with most software, this framework

could constantly evolve and improve with more testing and feedback. But for the scope of this thesis, we are happy with the current effectiveness of FREJA as an initial release. However, the language agnostic framework did not provide anywhere near the same capability as FREJA did. Our results point to a bleak response to the last research question:

RQ3: To what degree is making a general language agnostic framework possible?

This thesis showed that it is *technically* possible to create such a framework. There is no denying the generic quality of the prototype, showing about equal effectiveness for programming assignments written in very different programming languages, such as Java, Python and Haskell. Both advantages and disadvantages become apparent when comparing the prototype to other methods of creating programming assignments. Relying on a niche development platform, such as MPS, makes it very inaccessible, but allows for creating a rich tailor-made editor. This makes the language itself very easy to learn and use. However, the striking difference between FREJA and the generic prototype is FREJA's ability to constrain assignment development to a single source. On the other hand, the language agnostic framework has multiple places of development, and even some artefacts must be created in isolation. This isolated development causes a big concern for time waste and inconsistency issues when refactoring. As discussed earlier, developing the solution to programming assignments within MPS is highly unimaginable, leaving the DSL method with this isolation problem no matter how the language is designed. LSP does potentially bring an opportunity for developing the underlying assignment source code in other editors, however, it lacks support for language workbenches such as MPS.

The usefulness of a programming language agnostic framework for creating assignments is questionable with these results. The comparative examples showed that a generic framework accomplishes less than a language specific framework in almost every way. Furthermore, a teacher might only use one programming language in their own courses, resulting in them needing a framework to just handle that language. Then the benefits of having a generic framework is entirely useless for them. On the other hand, FREJA is the culmination of trying to solve the problems posed in the first two research questions. Our results showed that FREJA is a sufficient solution to these problems. The framework is efficient and provides valuable benefits in assignment development. Our results suggest that making use of FREJA to create programming assignments in a real-life setting will prove itself useful. We hope that the work in this thesis is only the first step of many in the neglected, yet undervalued task of improving assignment development.

8.3 Future Work

It is difficult to determine the impact and practicality of FREJA from the results of this thesis alone. Experiences from a real scenario would be important to conclude its effectiveness in an authentic setting. A sensible next step would be to employ FREJA into the development of assignments at university-level

courses. This would provide a good opportunity to perform qualitative research by getting feedback from teachers and students in the form of questionnaires.

As mentioned earlier, software is seldom complete, in the sense that there is almost always room for improvement. This extends to FREJA also, and feedback from a real-life setting would certainly reveal areas of improvement. FREJA is open source, and we welcome anyone to continue enhancing the framework if they wish to. This thesis also serves as a starting guide in this aspect, with multiple chapters that covers installation steps, implementation details, and code architecture. Currently, FREJA has only a basic integration with Git. Although it is possible to have a separate Git repository for each generated project, propagating changes from the source project to each online cloud repository still requires the user to manually push the changes for each repository. Improving FREJA in this aspect could be very useful. Another area of improvement for FREJA is to define a pure *get* function for consistency restoration between assignment artefacts. This requires the assignment description to be entirely generated from the source project, and not depend on previous generated descriptions. Although a challenging task, combing FREJA with natural language models such as OpenAI Codex could bring forth an interesting solution.

Unfortunately, the inferiority of a language agnostic framework compared to a language specific one seems to be almost intrinsic, suggesting that focusing on developing and improving language specific frameworks is more beneficial going forward. We discussed the logical steps forward for FREJA, but Java is just one programming language of many. Python is also a prominent language in education, so building a Python specific framework may also be a sensible next step. However, if development of a generic framework were to continue, then focusing on enabling development of DSL programs within a regular IDE, instead of a language workbench, could potentially be fruitful. That would allow taking advantage of LSP, which strengthens the possibility of combining the development of DSL programs and assignment solutions to the same source.

List of Figures

1.1	Workflow of FREJA compared to the conventional method of creating programming assignments.	8
3.1	An example of an exercise description.	25
3.2	An exercise description that explains the semantics of the exercise.	27
3.3	A model of the artefacts for a FREJA assignment. The generated artefacts have an incoming arrow.	28
4.1	A simplified view of the assignment object hierarchy.	37
4.2	The transformation pipeline from source code to the DSL, and back.	41
4.3	An example of how the first prototype could look like.	42
4.4	The transformation pipeline for the second prototype.	44
4.5	The starting line of a solution being marked.	45
4.6	The end line of a solution being marked.	45
4.7	The resulting <code>Solution</code> concept that was created.	46
4.8	The top part shows a <code>File</code> editor with a solution to a Java exercise. Below that is the code that the solution should be replaced with. At the bottom is a preview of the generated output.	47
5.1	A console showing a <code>NodeException</code> error message.	55
5.2	The definition of the <code>File</code> concept.	58
5.3	The editor definition for the <code>File</code> concept.	58
5.4	A keybinding definition and the code that gets executed by it.	59
5.5	The <code>TextGen</code> definition for generating a <code>File</code> node into text.	60
6.1	A screenshot of the IntelliJ AsciiDoc plugin editor window.	66
6.2	A screenshot of the AsciiDoc output window.	67
6.3	A screenshot of the updated AsciiDoc output window.	68
6.4	Only the attribute values at line 4 and 5 has changed, while the attribute references at line 16 and 19 remain unchanged.	68
6.5	The <code>Import</code> editor.	70
6.6	The editor for the <code>HelloWorld</code> file.	71
6.7	The marked solution and a preview of the start code.	72
6.8	The editor for the <code>SolutionReplacement</code>	73
6.9	The updated solution that should now be replaced, and its corresponding output below.	74

7.1	Comparison of the amount of expressions between the different projects.	80
7.2	The description of an exercise where the goal is to reverse an array.	83
7.3	The code reference to the method definition is incorrect.	84
7.4	An automatically generated description template.	85
7.5	Number of expressions among the different projects.	87
7.6	Comparison of how assignment artefacts are created using different assignment development methods.	90
7.7	The solution and start code for an exercise in two different languages. Python is on the left and Haskell on the right.	92

List of Tables

8.1	Comparison of how the different methods meet each requirement.	97
8.2	Reduction in the number of components that must be manually created and maintained for both evaluations.	97

Listings

3.1	An excerpt of the start code for an exercise.	24
3.2	An excerpt of a test to the above exercise.	24
3.3	Solution to a simple Java exercise.	27
3.4	Start code for the above exercise.	27
4.1	The definition of the <code>Exercise</code> annotation.	32
4.2	An example of the <code>Exercise</code> annotation being used.	32
4.3	An example of the <code>ReplacementCode</code> annotation being used . . .	33
4.4	The solution to an exercise that highlights which statements are part of the exercise solution through comments.	33
4.5	The start code to the above exercise with only the solution removed.	34
4.6	The solution wrapped in annotation variables	34
4.7	The generated start code with a syntax error.	35
4.8	A simple exercise.	39
4.9	An exercise description with an attribute reference.	39
4.10	An exercise description with a substitution for the attribute ref- erence.	39
4.11	An XML AST representation of the Java statement <code>a = x + 1;</code>	42
5.1	The build lifecycle in the FREJA Maven plugin <code>pom.xml</code> file. . .	50
5.2	A snippet of the <code>FrejaMojo</code> class and its field variables.	51
5.3	The <code>build</code> method of the <code>AssignmentBuilder</code> class.	52
5.4	The <code>TaskOperations</code> interface.	53
5.5	A method for transforming a <code>Task</code> node.	54
5.6	A method for finding the value of a annotation element/member. .	54
5.7	An example of test data that is identified by the <code>TestId</code> annotation.	56
5.8	The test data from Listing 5.7 is retrieved using the <code>getNodeWithId</code> method.	57
6.1	The default JDK version in the <code>pom.xml</code> file.	61
6.2	FREJA annotations Maven dependency.	62
6.3	The FREJA Maven plugin specifying the target folder.	62
6.4	Different types of glob patterns for the <code>ignore</code> configuration. . .	63
6.5	FREJA tutorial POM file.	64
6.6	The simple <code>HelloWorld</code> class so far.	64
6.7	The <code>helloWorld</code> method with the <code>Exercise</code> annotation.	65
6.8	Configuration settings.	67
6.9	A main method.	68
6.10	The final source code.	69
6.11	The start code after the solution has been replaced.	69

7.1	The original start code where the solution is replaced with a throw statement.	77
7.2	The solution to an exercise for reversing an array.	83
7.3	The start code for the reverse method.	83
7.4	The source code of the exercise shows that correct parameter names are a and b	84
7.5	The Exercise annotation for the very first sub-exercise.	84

Acronyms

API Application Programming Interface.

AST Abstract Syntax Tree.

BX Bidirectional Transformations.

DRY Do Not Repeat Yourself.

DSL Domain Specific Language.

DW Data Warehouse.

EMF Eclipse Modeling Framework.

ESB Enterprise Service Bus.

FREJA **FR**amework for **En**gineering **J**ava **A**ssignments.

LSP Language Server Protocol.

M2M Model-To-Model.

M2T Model-To-Text.

MDSE Model Driven Software Engineering.

MOJO Maven plain Old Java Object.

POM Project Object Model.

SOA Service Oriented Architecture.

SSOT Single Source of Truth.

Appendix A

Source Code

The source code for FREJA is available at this URL: <https://github.com/ErlendBerntsen/freja>.

The source code for the language agnostic framework is available at this URL: <https://github.com/ErlendBerntsen/Paastel>.

Bibliography

- [1] Nahla Abid et al. “The evaluation of an approach for automatic generated documentation.” In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 307–317.
- [2] Faris Abou-Saleh et al. “Introduction to Bidirectional Transformations.” eng. In: *Bidirectional Transformations*. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 1–28. ISBN: 3319791079.
- [3] F Bancilhon and N Spyrtos. “Update semantics of relational views.” eng. In: *ACM transactions on database systems* 6.4 (1981), pp. 557–575. ISSN: 0362-5915.
- [4] Elisa Baniassad et al. “STOP THE (AUTOGRADER) INSANITY: Regression Penalties to Deter Autograder Overreliance.” In: *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. SIGCSE '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 1062–1068. ISBN: 9781450380621. DOI: 10.1145/3408877.3432430. URL: <https://doi.org/10.1145/3408877.3432430>.
- [5] Jacob Beningo. “Documenting Firmware with Doxygen.” In: *Reusable Firmware Development*. Springer, 2017, pp. 121–148.
- [6] Andrew Binstock and Simon Maple. *The Largest Survey Ever of Java Developers*. URL: <https://blogs.oracle.com/javamagazine/post/the-largest-survey-ever-of-java-developers> (visited on Oct. 6, 2022).
- [7] Anastasiia Birillo et al. “Hyperstyle: A Tool for Assessing the Code Quality of Solutions to Programming Assignments.” In: *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*. 2022, pp. 307–313.
- [8] Douglas S Blank et al. “nbgrader: A tool for creating and grading assignments in the Jupyter Notebook.” In: *The Journal of Open Source Education* 2.11 (2019).
- [9] Hannah Blau and J Eliot B Moss. “Frenchpress gives students automated feedback on Java program flaws.” In: *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*. 2015, pp. 15–20.
- [10] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. “Model-driven software engineering in practice.” In: *Synthesis lectures on software engineering* 3.1 (2017), pp. 1–207.
- [11] Jordi Cabot. “Positioning of the Low-Code Movement within the Field of Model-Driven Engineering.” In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and*

- Systems: Companion Proceedings*. MODELS '20. Virtual Event, Canada: Association for Computing Machinery, 2020. ISBN: 9781450381352. DOI: 10.1145/3417990.3420210. URL: <https://doi.org/10.1145/3417990.3420210>.
- [12] Mark Chen et al. “Evaluating large language models trained on code.” In: *arXiv preprint arXiv:2107.03374* (2021).
- [13] Lucas Cordova et al. “A Comparison of Inquiry-Based Conceptual Feedback vs. Traditional Detailed Feedback Mechanisms in Software Testing Education: An Empirical Investigation.” In: *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. SIGCSE '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 87–93. ISBN: 9781450380621. DOI: 10.1145/3408877.3432417. URL: <https://doi.org/10.1145/3408877.3432417>.
- [14] Paul Denny et al. “Codewrite: supporting student-driven practice of java.” In: *Proceedings of the 42nd ACM technical symposium on Computer science education*. 2011, pp. 471–476.
- [15] D. Dig and R. Johnson. “The role of refactorings in API evolution.” In: *21st IEEE International Conference on Software Maintenance (ICSM'05)*. 2005, pp. 389–398. DOI: 10.1109/ICSM.2005.90.
- [16] Mickaël Duruisseau et al. “VisUML: a live UML visualization to help developers in their programming task.” In: *International Conference on Human Interface and the Management of Information*. Springer. 2018, pp. 3–22.
- [17] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
- [18] Yaroslav Golubev et al. “One thousand and one stories: a large-scale survey of software refactoring.” In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021, pp. 1303–1313.
- [19] Jim Gray et al. “The Dangers of Replication and a Solution.” In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. SIGMOD '96. Montreal, Quebec, Canada: Association for Computing Machinery, 1996, pp. 173–182. ISBN: 0897917944. DOI: 10.1145/233269.233330. URL: <https://doi-org.galanga.hvl.no/10.1145/233269.233330>.
- [20] Wang Haoyu and Zhou Haili. “Basic Design Principles in Software Engineering.” In: *2012 Fourth International Conference on Computational and Information Sciences*. 2012, pp. 1251–1254. DOI: 10.1109/ICCIS.2012.91.
- [21] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. “Symmetric Lenses.” eng. In: *SIGPLAN notices* 46.1 (2011), pp. 371–384. ISSN: 0362-1340.
- [22] Bettina Kemme and Gustavo Alonso. “Database Replication: A Tale of Research across Communities.” In: *Proc. VLDB Endow.* 3.1–2 (2010), pp. 5–12. ISSN: 2150-8097. DOI: 10.14778/1920841.1920847. URL: <https://doi-org.galanga.hvl.no/10.14778/1920841.1920847>.
- [23] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. “A tutoring system to learn code refactoring.” In: *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 2021, pp. 562–568.
- [24] Zeba Khanam. “Barriers to Refactoring: Issues and Solutions.” In: *2454-4248* 4 (Feb. 2018), p. 232.

- [25] Diksha Khurana et al. “Natural language processing: State of the art, current trends and challenges.” In: *Multimedia Tools and Applications* (2022), pp. 1–32.
- [26] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. “A Field Study of Refactoring Challenges and Benefits.” In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE ’12. Cary, North Carolina: Association for Computing Machinery, 2012. ISBN: 9781450316149. DOI: 10.1145/2393596.2393655. URL: <https://doi-org.galanga.hvl.no/10.1145/2393596.2393655>.
- [27] Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. “Rascal: A domain specific language for source code analysis and manipulation.” In: *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2009, pp. 168–177.
- [28] Donald Ervin Knuth. “Literate programming.” In: *The computer journal* 27.2 (1984), pp. 97–111.
- [29] Gábor Kövesdán and László Lengyel. “Meta3: a code generator framework for domain-specific languages.” en. In: *Software & Systems Modeling* 18.4 (Aug. 2019), pp. 2421–2439. ISSN: 1619-1374. DOI: 10.1007/s10270-018-0673-6. URL: <https://doi.org/10.1007/s10270-018-0673-6> (visited on Mar. 3, 2022).
- [30] Douglas Kramer. “Api documentation from source code comments: a case study of javadoc.” In: *Proceedings of the 17th annual international conference on Computer documentation*. 1999, pp. 147–153.
- [31] Algirdas Laukaitis. “Code Transformation Pattern Alignments and Induction for ERP Legacy Systems Migration.” In: *Perspectives in Business Informatics Research*. Ed. by Raimundas Matulevičius and Marlon Dumas. Cham: Springer International Publishing, 2015, pp. 228–240. ISBN: 978-3-319-21915-8.
- [32] Haden Hooyeon Lee. “Effectiveness of Real-Time Feedback and Instructive Hints in Graduate CS Courses via Automated Grading System.” In: *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. SIGCSE ’21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 101–107. ISBN: 9781450380621. DOI: 10.1145/3408877.3432463. URL: <https://doi.org/10.1145/3408877.3432463>.
- [33] Yannis Lilis and Anthony Savidis. “A survey of metaprogramming languages.” In: *ACM Computing Surveys (CSUR)* 52.6 (2019), pp. 1–39.
- [34] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. USA: Prentice Hall PTR, 2008. ISBN: 0132350882.
- [35] Robert Cecil Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.
- [36] Florencia Miranda. “Using Failing Test Cases to Semi-Automate Feedback for Beginner Programmers.” In: *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. SIGCSE ’21. Virtual Event, USA: Association for Computing Machinery, 2021, p. 1384. ISBN: 9781450380621. DOI: 10.1145/3408877.3439696. URL: <https://doi.org/10.1145/3408877.3439696>.
- [37] Stas Negara et al. “A Comparative Study of Manual and Automated Refactorings.” In: *ECOOP 2013 – Object-Oriented Programming*. Ed. by

- Giuseppe Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 552–576. ISBN: 978-3-642-39038-8.
- [38] Christian D. Newman et al. “Simplifying the construction of source code transformations via automatic syntactic restructurings.” In: *Journal of Software: Evolution and Process* 29.4 (2017). e1831 JSME-16-0041.R1, e1831. DOI: <https://doi.org/10.1002/smr.1831>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1831>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1831>.
- [39] Nhan Nguyen and Sarah Nadi. “An empirical evaluation of GitHub copilot’s code suggestions.” In: *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022, pp. 1–5.
- [40] Candy Pang and Duane Szafron. “Single Source of Truth (SSOT) for Service Oriented Architecture (SOA).” In: *Service-Oriented Computing*. Ed. by Xavier Franch et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 575–589. ISBN: 978-3-662-45391-9.
- [41] Andrei Papancea, Jaime Spacco, and David Hovemeyer. “An open platform for managing short programming exercises.” In: *Proceedings of the ninth annual international ACM conference on International computing education research*. 2013, pp. 47–52.
- [42] Nea Pirttinen et al. “Crowdsourcing programming assignments with CrowdSorcerer.” In: *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. 2018, pp. 326–331.
- [43] IBM Redbooks. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. eng. 1st ed. IBM Redbooks. Durham: I B M, 2004. ISBN: 9780738453163.
- [44] Markiyani Rizun, J-C Bach, and Stéphane Ducasse. “Code transformation by direct transformation of asts.” In: *Proceedings of the International Workshop on Smalltalk Technologies*. 2015, pp. 1–7.
- [45] Rohan Roy Choudhury, Hezheng Yin, and Armando Fox. “Scale-driven automatic hint generation for coding style.” In: *International Conference on Intelligent Tutoring Systems*. Springer. 2016, pp. 122–132.
- [46] Sami Sarsa et al. “Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models.” In: *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*. 2022, pp. 27–43.
- [47] Valeriy V Savchenko et al. “NOBRAINER: A Tool for Example-Based Transformation of C/C++ Code.” In: *Programming and Computer Software* 46.5 (2020), pp. 362–372.
- [48] Robert M Siegfried et al. “Trends Of Commonly Used Programming Languages in CS1 And CS2 Learning.” In: *2021 16th International Conference on Computer Science & Education (ICCSE)*. IEEE. 2021, pp. 407–412.
- [49] Cleidson R. B. de Souza and David F. Redmiles. “On The Roles of APIs in the Coordination of Collaborative Software Development.” In: *Computer Supported Cooperative Work (CSCW)* 18.5 (Sept. 2009), p. 445. ISSN: 1573-7551. DOI: 10.1007/s10606-009-9101-3. URL: <https://doi.org/10.1007/s10606-009-9101-3>.
- [50] Peter Sovietov. “Automatic generation of programming exercises.” In: *2021 1st International Conference on Technology Enhanced Learning in Higher Education (TELE)*. IEEE. 2021, pp. 111–114.

- [51] Daniel E Stevenson and Paul J Wagner. “Developing real-world programming assignments for CS1.” In: *ACM SIGCSE Bulletin* 38.3 (2006), pp. 158–162.
- [52] Hiroyuki Takizawa et al. “A Use Case of a Code Transformation Rule Generator for Data Layout Optimization.” In: *Sustained Simulation Performance 2016*. Ed. by Michael M. Resch et al. Cham: Springer International Publishing, 2016, pp. 21–30.
- [53] Hiroyuki Takizawa et al. “Xeolver: An XML-based code translation framework for supporting HPC application migration.” In: *2014 21st International Conference on High Performance Computing (HiPC)*. 2014, pp. 1–11. DOI: 10.1109/HiPC.2014.7116902.
- [54] Robert Waszkowski. “Low-code platform for automating business processes in manufacturing.” In: *IFAC-PapersOnLine* 52.10 (2019). 13th IFAC Workshop on Intelligent Manufacturing Systems IMS 2019, pp. 376–381. ISSN: 2405-8963. DOI: <https://doi.org/10.1016/j.ifacol.2019.10.060>. URL: <https://www.sciencedirect.com/science/article/pii/S2405896319309152>.
- [55] Daniel Weise and Roger Crew. “Programmable syntax macros.” In: *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*. 1993, pp. 156–165.
- [56] Jacqueline L Whalley et al. “An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies.” In: *Conferences in Research and Practice in Information Technology Series*. 2006.
- [57] Wikipedia. *Single source of truth — Wikipedia, The Free Encyclopedia*. <http://en.wikipedia.org/w/index.php?title=Single%20source%20of%20truth&oldid=1120485355>. [Online; accessed 09-November-2022]. 2022.