University of Groningen

Software Maintainance and Evolution
WMCS013-05

FAULTIFA

A De-Faultified Project

Authors:
Arjan Dekker (s3726169)
Jesse Maarleveld (s3816567)
Chris Worthington (s3715086)

Teaching Assistant:
Charles Alec Downs

Lecturer:
prof. dr. A. Capiluppi

January 28, 2022

university of groningen / faculty of science and engineering

Contents

Disclaimer: Diagrams that contain arrows of different types, such as dotted arrows, colored arrows, etc., have no semantic meaning. The different types of arrows are used for readability, e.g. to have a better visible difference between the arrows.

1 Introduction

This is our report for the course Software Maintenance and Evolution. For this course we were tasked with taking an existing project and building upon in a meaningful way. The main aims of this project revolve around maintenance activities. Namely, selecting maintenance activities to perform that involve either refactoring, adding new features and/or increasing the test coverage.

In this report we will give an overview and analysis of the existing project Faultify. Following this we will then provide a proposal and rationalisation of our chosen maintenance activities. We will also include an overview of how we carried out the maintenance activities alongside an evaluation of how effective they were. And finally, an outline is given on what issues still exist in addition to proposed future work.

1.1 Introducion to Faultify

Faultify is an RDW project for the mutation testing of C# projects. The concept of mutation testing is explained in Section 1.2.

RDW provide this project for students to work on and it has already been worked on by multiple student groups before now. Both a group of students from Windesheim, and another group of students from the RUG. The group from Windesheim were the initial creators of Faultify, who developed the base structure and functionality of the program. Then the group from the RUG expanded the code base with more usability features and more test hosts, such as NUnit. One of the Windesheim students is also supporting another version of Faultify as an open-source project on github `https://github.com/Faultify/Faultify`.

RDW are aiming for Faultify to deploy in their company in 2022. Since they want the tool to be used by employees they need it to be both efficient and reliable. If the tool fails to be useful when it is first introduced it will then be hard to sell it to the employees a second time, so it needs to be as good as it can be first time. However, considerable work still needs to be done to bring it to a releasable state. According to the previous RUG students, the code has quite a fragile implementation. The code has very high coupling and is also difficult to understand, maintain and test. These issues will be further outlined in Section 2.

1.2 Mutation Testing

Mutation testing is a technique used to verify the strength of automated tests. Faultify is intended to perform this specifically for a range of C# testing libraries. The general overview of the mutation testing process is illustrated in Figure 1.

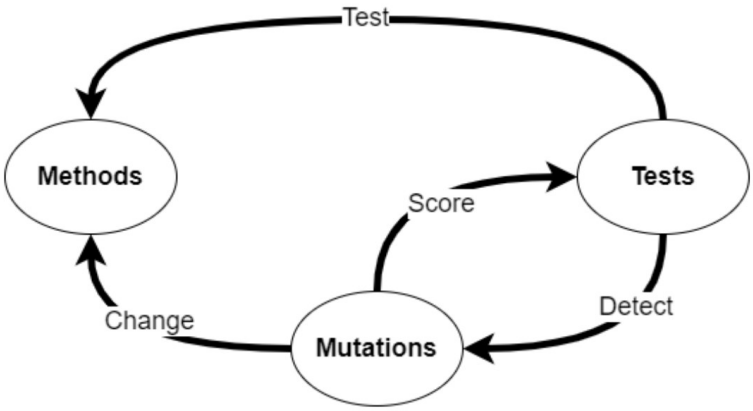

Figure 1: Mutation Testing Process

The technique involves inserting changes, referred to as 'mutations', into a program's methods and then running the automated tests. Faultify applies these mutations to the compiled byte code of the program. These small changes will then alter the functionality of the program and, in the best case scenario, be caught by failed tests. When a test fails due to a mutation, we refer to this test as having 'detected' or 'killed' the mutation. The more mutations detected the better, as it implies that the tests are strong enough to detect changes in functionality. With a set of mutations, a score can be given to a test set as a ratio from how many mutations were killed compared to how many mutations there are [1].

Each mutation is one small syntactic change to a method, such as an operator change. If tests are able to detect any of these atomic mutations, then they should also detect complex combinations of mutations by the Coupling Effect [2]. So it is not necessary to apply combinations of mutations at once on the same method.

Issues are highlighted when a mutation 'survives' as no tests were caused to fail because of it. An example of this is shown in Figures 2 and 3. Figure 2 shows the mutation applied to the Multiplication() method that alters the multiplication operator to be an addition operator, as it is presented in a HTML report from Faultify. This mutation should cause different output and a test for the method should therefore fail. However, the test for Multiplication(), shown in Figure 3, does not fail due to the test input 2 and 2 still receiving the same output of 4. This indicated to the developer of the test that different test input should be used. This example also shows how it is necessary to map mutations to methods and subsequently methods to tests in order to identify the problem when a mutation survives.
A more elaborate description of how mutation testing is done in Faultify, will be given in section 2.3.4.

Original Source

```
public int Multiplication(int lhs, int rhs)
{
    return lhs * rhs;
}
```

Mutated Source

```
public int Multiplication(int lhs, int rhs)
{
    return lhs + rhs;
}
```

Figure 2: Survived Multiplication Mutation as Displayed by a Faultify HTML Report

```
[Test]
public void TestMultiplication()
{
    BenchmarkTarget targets = new BenchmarkTarget();
    int actual = targets.Multiplication(2, 2);
    var expected = 4;
    Assert.AreEqual(expected, actual);
}
```

Figure 3: Test for Multiplication Method from the Source Code of Faultify

2 Analysis of Existing Project

In this section, we will be analyzing the existing project to determine if and what parts need improvement, and to get an idea of how things are tied together.

2.1 High-level overview

We start out with a high-level overview. Faultify comes with a total of ten different modules, which are summarized in table 1. The Faultify.Tests module contains the tests for Faultify. The Benchmark module contains code for benchmarking Faultify. However, the code does not seem to be functional at the moment. The remaining modules contain the actual implementation of Faultify and will be the focus of our analysis.

Module	**LOC**	**Executable LOC**
Faultify.Analyze	1753	351
Benchmark	2012	772
Faultify.Cli	370	105
Faultify.Core	309	49
Faultify.Report	277	88
Faultify.Injection	337	102
Faultify.TestRunner	1831	609
Faultify.TestRunner.Collector	168	55
Faultify.TestRunner.Shared	215	71
Faultify.Test	1901	676

Table 1: Visual Studio Code Metrics: Lines of Code

2.2 Static Code Analysis

This subsection contains the static code analysis that we performed on the provided version of Faultify. We started off by analyzing the project using SonarQube. However, SonarQube has a problem analyzing the project and therefore more than half of the project is not correctly analyzed by the program. Therefore we do not include an extensive analysis of SonarQube's results and it SonarQube will also not be used for the comparison between the old and the new version.

2.2.1 Code Coverage

Using the built in code coverage tool of IntelliJ Rider (dotCover [3]), a code coverage of 15% is measured. The available tests only test the module Faultify.Analyzer, which has a code coverage of about 80%. A screenshot of the code coverage analysis can be found below. Note that dotCover uses statement coverage [4].

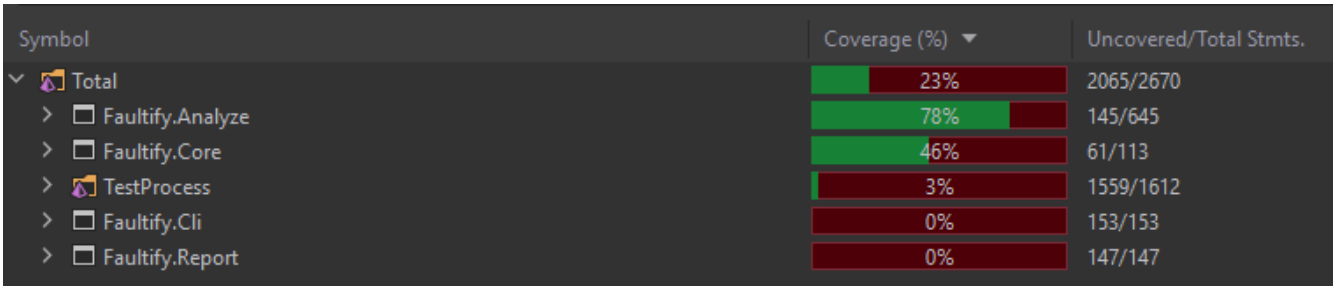

Figure 4: Initial code coverage analysis of dotCover

The amount of code coverage is thus very small. However, what could be considered the most important part of the program is tested: Faultify.Analyzer. The Faultify.Analyze module contains code for the mutations, which is essentially what Faultify is about. Hence, that could be the reason that only that module is tested. Another possibility is that the authors intended the Faultify.Analyze module to be its own separate package or utility. From the documentation, it is clear that they intended the module to be use-able separately from Faultify. To quote the existing documentation:

Analyzers can be used separately from Faultify.

However, that does not make it a valid reason to not test the rest of the program. Therefore increasing the code coverage should be considered as a valid maintenance activity.

2.2.2 Manual Look Through and Initial Test Phase

SonarQube did not point out a lot of problems with the code quality and a manual inspection confirmed that. We found a few methods that were too long (e.g. `Faultify.TestRunner/MutationTestProject.cs`) and also multiple classes in single files (e.g. `Faultify.TestRunner.Shared/MutationCoverage.cs`), which is very inconsistent with the rest of the code. However, most of the code was of sufficient quality.

The most problematic module, `Faultify.TestRunner`, was okay according to SonarQube. This could be due to the fact that SonarQube has difficulties analyzing this module (e.g. it does not show the lines of code). However, from a manual inspection it was clear that this module is troublesome.

Since a lack of code coverage is a problem for this project, we started with creating tests to get an idea of the feasibility of doing that. Furthermore, the idea behind writing tests first is that refactoring is safer, since the tests would detect any unforeseen changes of behaviour.

For the most basic classes this was very doable. However, for a lot of classes this is not a feasible approach. One of the things that we noted is a high coupling in the current source code architecture. Because of that, a vast amount of objects have to be instantiated to test a single method. Therefore writing tests took an unfeasible amount of time. This high coupling is explained in the architecture analysis section 2.3.

Decoupling the architecture is not the only solution to make the code more testable. Another solution is to use more interfaces. Using mocking frameworks such as Moq, a test can mock an interface without having to create all the underlying objects of that interface. Mocking is not possible with other classes or object. Therefore creating more interfaces for classes would make testing much more doable. Because of the fact that interfaces are used rarely in the provided state of Faultify, creating tests was considered unfeasible without changing the code.

2.2.3 Lines of Code

Using Visual Studio Code Metrics [5], the lines of code and executable lines of code can be calculated for each module. The lines of code is one of the problems with the module `Faultify.TestRunner`. It is about a third of the total lines of code, which is not in proportion to the rest of the modules (see table 1). However, that is not the biggest problem.

`Faultify.TestRunner` does too many things for a single module. One indication of the is the number of executable lines of code (see table 1). This is approximately double `Faultify.Analyze`, which is already huge compared to the other modules (3.5 times larger than the next largest module: `Faultify.Cli`). The analysis in section 2.3.5 goes deeper into the fact that `Faultify.TestRunner` has too many responsibilities.

2.2.4 Cyclomatic Complexity

Another metric that can be calculated by Visual Studio is cyclomatic complexity. It is a measure on how complex the code paths in the program are. A high complexity can be caused by e.g. a lot of if and else statements.

One of the problems with a high cyclomatic complexity is that a lot of tests are needed to cover the different code paths. A direct consequence of this is that the code becomes harder to test and also less maintainable [5].

Table 2 shows that there are two modules that have high cyclomatic complexity compared to the other modules. This means that those two modules are hard to test and also hard to maintain. Reducing the amount of modules with such a high cyclomatic complexity or reducing the cyclomatic complexity of those modules would therefore be helpful in the long term regarding maintenance.

Module	Cyclomatic Complexity
Faultify.Analyze	262
Faultify.Cli	45
Faultify.Core	70
Faultify.Report	77
Faultify.Injection	35
Faultify.TestRunner	302
Faultify.TestRunner.Collector	13
Faultify.TestRunner.Shared	29

Table 2: Visual Studio Code Metrics: Cyclomatic Complexity

2.2.5 Conclusion of Static Code Analysis

The general takeaway from the static code analysis is that the code is of good quality in general. However, most of the code is not tested. It is also hard to test the code due to a high coupling in the architecture and a lack of interfaces for classes. Therefore it is impossible to mock behaviour of components to make testing more feasible. Furthermore it is clear that there is a problematic module, Faultify.TestRunner. It has a very high cyclomatic complexity and also a lot of executable lines of code compared to the other modules. This is hinting towards a potential violation of the single responsibility principle. Another thing to notice is that there is a high coupling that should be fixed, which could further improve the testability.

2.3 Architecture Recovery

Because we found the available information about the architecture to be lacking (section 2.3.1), and because we were unable to observe a clear architecture because the relationship between a number of modules was unclear, we decided to perform an architecture recovery. This was done according to the method described in [6].

2.3.1 Available Architectural Information

We had access to documentation provided by the two previous groups of students who worked on Faultify. The documentation provided by the Windesheim students provided information about the intended functionality of the program, but contained little detail about the inner functioning or architecture as a whole.
The architectural documentation provided by the group of RUG students was more elaborate. However, we also found it to be incorrect and incomplete. The class diagrams only contained inheritance, implementation, and aggregation information, making them of relatively little use. A figure documenting the module dependency graph was also provided (see figure 5). However, after inspection, we found this diagram to be faulty. In stead, we identified the inter-modular dependencies given in figure 6.

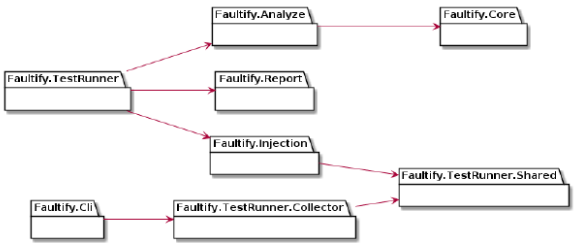

Figure 5: Inter-modular dependencies as documented by the RUG students.

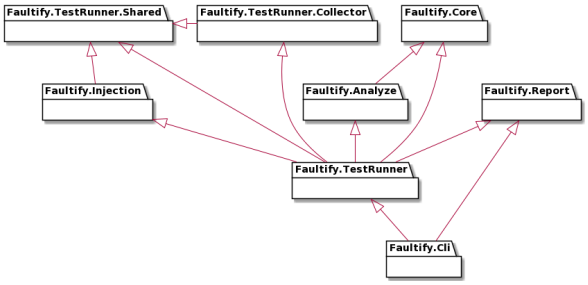

Figure 6: Actual inter-module dependencies of the old Faultify project.

2.3.2 Class Diagram

We started out by investigating all the relationships between classes in the project, and combining all of these into a single class diagram. We used Structure101[1] in order identify static relationships between classes. Additionally, we found a number of dynamic relationships by going through the code. Specifically, we found out that the so-called `DotnetTestHostRunner` class depends on the `CoverageDataCollector` and `TestDataCollector` classes. The resulting class diagram it too large to fit into this report, and is stead provided as a separate image. It is not understand to look at the class diagram in order to understand the analysis – it was merely used as a stepping stone for the remainder of the architecture recovery.

2.3.3 Identifying Processing Components

The first thing we did, in accordance with [6], was identifying the processing components. The result of following the process has resulted in the diagram presented in figure 7. The numbers in the diagram denote the amount of dependencies (in terms of classes) between components. The names of the components are based on our understanding of their purpose.
We also visualized how these components map onto the original source code modules. This is depicted graphically in figure 8. As we can see, some components, such as "TestRunners", consist of multiple modules (Faultify.TestRunner, Faultify.TestRunner.Shared, Faultify.TestRunner.Collector, Faultify.TestRunner.Report). On the other hand, some modules contain source code for many different components (e.g. Faultify.TestRunner contains code for the following components: MutationSessionProgressTracker, ProjectDuplicator, MutationTestProject, ProjectAnalyzer, MutationMetadataCollector, TestRunners). This hints at the fact their might be some problems with the division of responsibilities throughout the project. Additionally, the TestRunners component still contains a substantial amount of classes without a clear picture of how they work together or what their exact responsibility is.

[1] `https://www.structure101.com/`

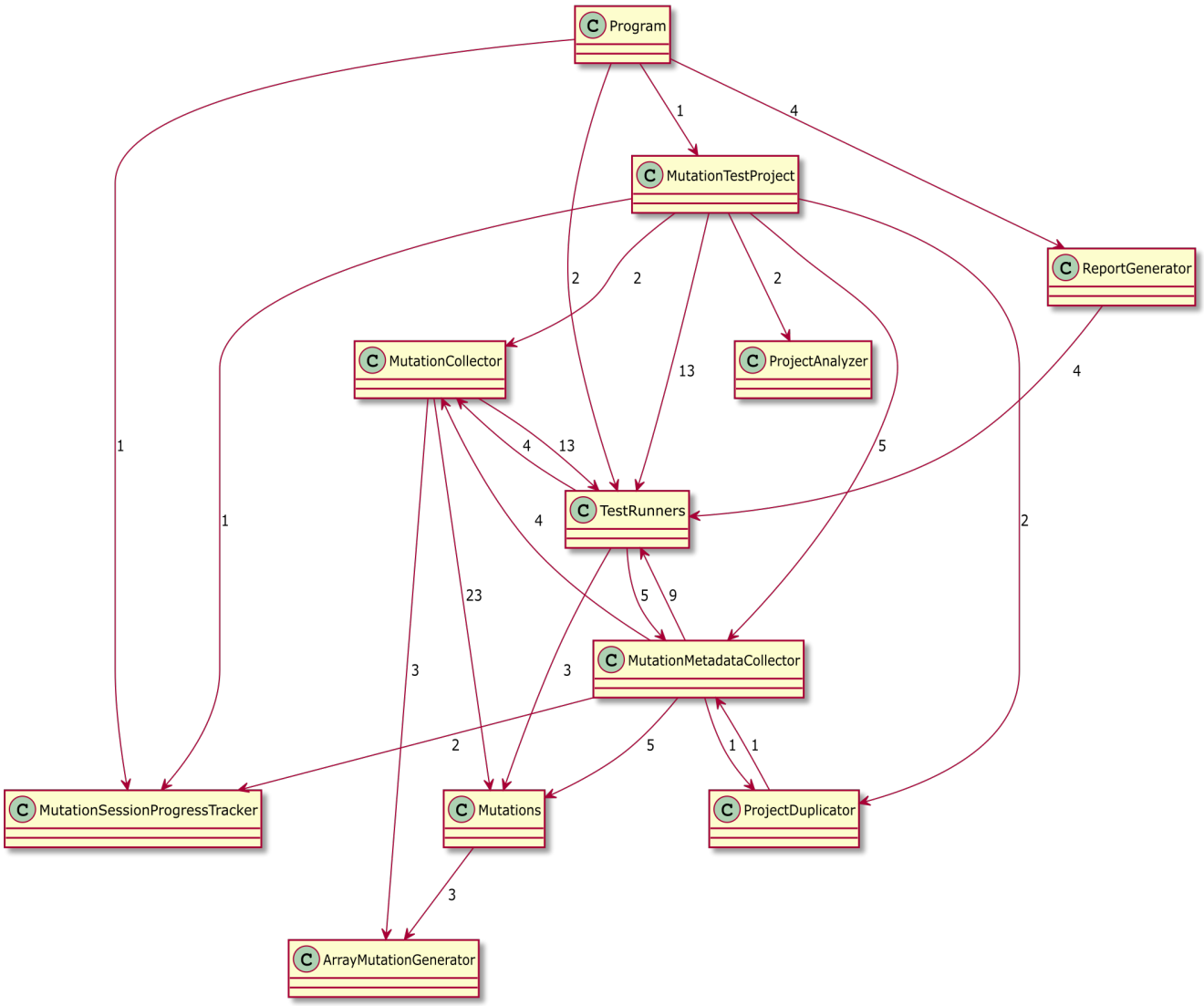

Figure 7: Processing components identified during the architecture recovery process. Numbers denote amount of dependencies (in terms of classes) between components)

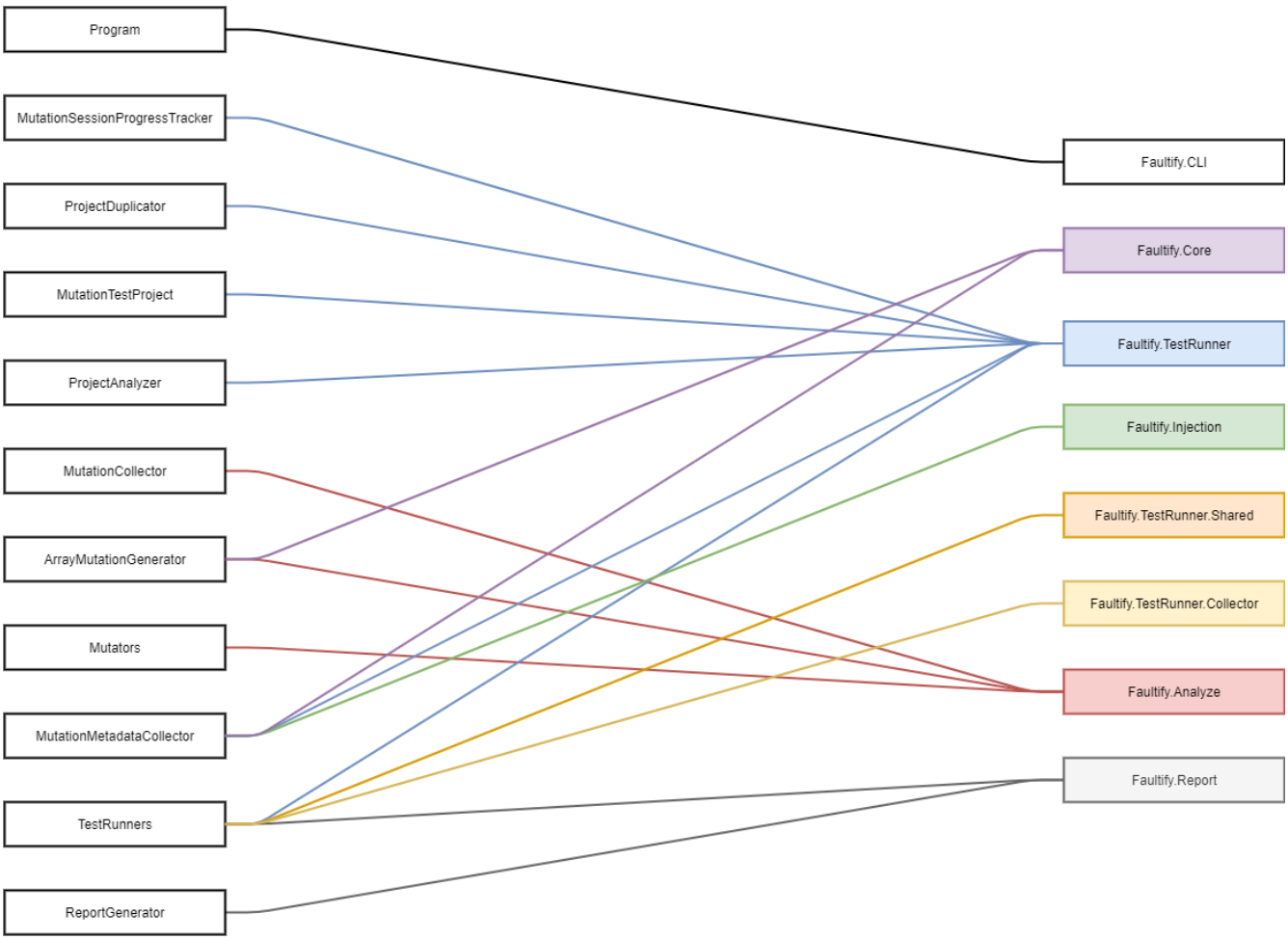

Figure 8: Mapping of source modules to components identified during architecture recovery.

2.3.4 Identifying Main Responsibilities

In section 2.3.3, we identified the processing components. We also concluded that it is not still clear what all components do. Because of this, we tried to identify the main responsibilities in the program. *Responsibilities* may be a somewhat ill-defined term. By responsibilities, we just mean high-level tasks in the program, whose combination gives a sufficient understanding of how the program works on a somewhat low level, without going into too much detail.

We tried to identify these responsibilities by analyzing component interactions. In [6], it is said that the first step in analyzing interactions, key use cases must be identified. Faultify has only one function and only one main sequence of events ("running the mutation test process") is present. As such, we just created a sequence diagram describing all component interactions happening when invoking Faultify. The results of this process are depicted in figure 9. The diagram has been annotated with colored rectangles identifying the main responsibilities, because the diagram is difficult to read due to its size. A version without these annotations can be found in appendix A. The meaning of the annotations in figure 9 are further explained in table 3.
We also want to bring attention to the nesting in the diagram. Some rather distinct tasks (e.g. collecting mutations and scheduling test runs) are done through nesting in stead of sequentially. The process executed by Faultify is linear in nature, with only the tests being ran in parallel. However, this is not strongly reflected in the figure 7. In stead, responsibilities are grouped together through nesting, which makes it more difficult to understand the flow of the program.

Color	Responsibility	Description
	Logging	Show logging messages to the user.
	Project Building	Build the target project, so that the program has access to the compiled byte code.
	Project Duplication	Create copies of the compiled project, which can be mutated in parallel, without leaving any artifacts in the original project.
	Coverage Collection	Collect coverage information, so that the program knows what tests invoke what methods in the program.
	Mutation Collection	Collect all mutations which can be applied to the target program.
	Test Run Scheduling	Combine mutations in batches which can be tested at the same time. Every mutation in such a batch ("test run") affects a unique set of tests, so that a test failure can be connected to a single mutation.
	Source Code Collection	Decompile byte code and obtain the (text) source code. Used for reporting.
	Mutation Applying	Apply the actual byte code mutations to a project.
	Timeout tracking	Track timeouts. Some mutations can cause a timeout (e.g. some while loop mutations). If a mutation to some opcode causes a timeout, that opcode is not mutated again.
	Test Running	Run the test suite to test whether mutations can be killed.
	Report Generation	Generate the report for the user.

Table 3: Legend explaining the annotations in figure 9

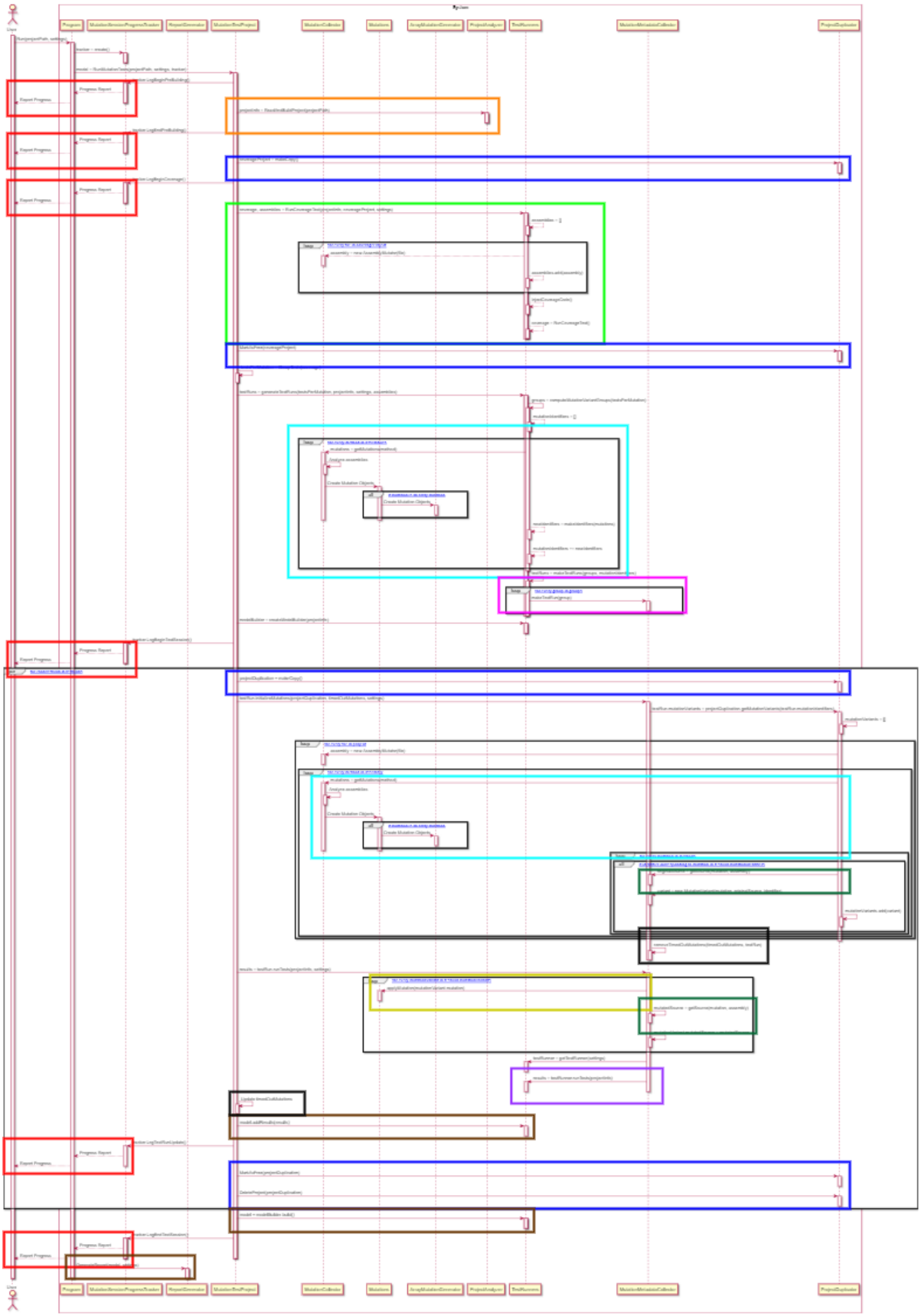

Figure 9: Sequence diagram identified during the recovery process, with main responsibilities annotated.

2.3.5 Mapping Components to Responsibilities

In section 2.3.3 we identified the processing components and in section 2.3.4 we identified the interactions between those components. In this section, we provide a more thorough analysis of the division of responsibilities amongst the components. Figure 10 shows how the modules and identified components map onto the responsibilities we identified.

First, we consider the mapping from modules to responsibilities. We observe that some responsibilities (e.g. coverage collection) are divided amongst many different modules (Faultify.Injection, Faultify.TestRunner.Shared, Faultify.TestRunner.Collector, Faultify.TestRunner). On the other hand, the Faultify.TestRunner modules is fully or partially responsible for nine different identified responsibilities. This shows that when looking at the modules, there is no clear division of responsibilities. Some modules have more than one, while others have to be combined in order to fulfil even one responsibility.

The story is similar, though a bit more complicated when looking at the mapping from components to responsibilities. We still have components handling many responsibilities (e.g TestRunners), and other responsibilities which are handled by multiple components (e.g. mutation collection). This shows that also when looking at the components, there is no proper adherence to single responsibility.

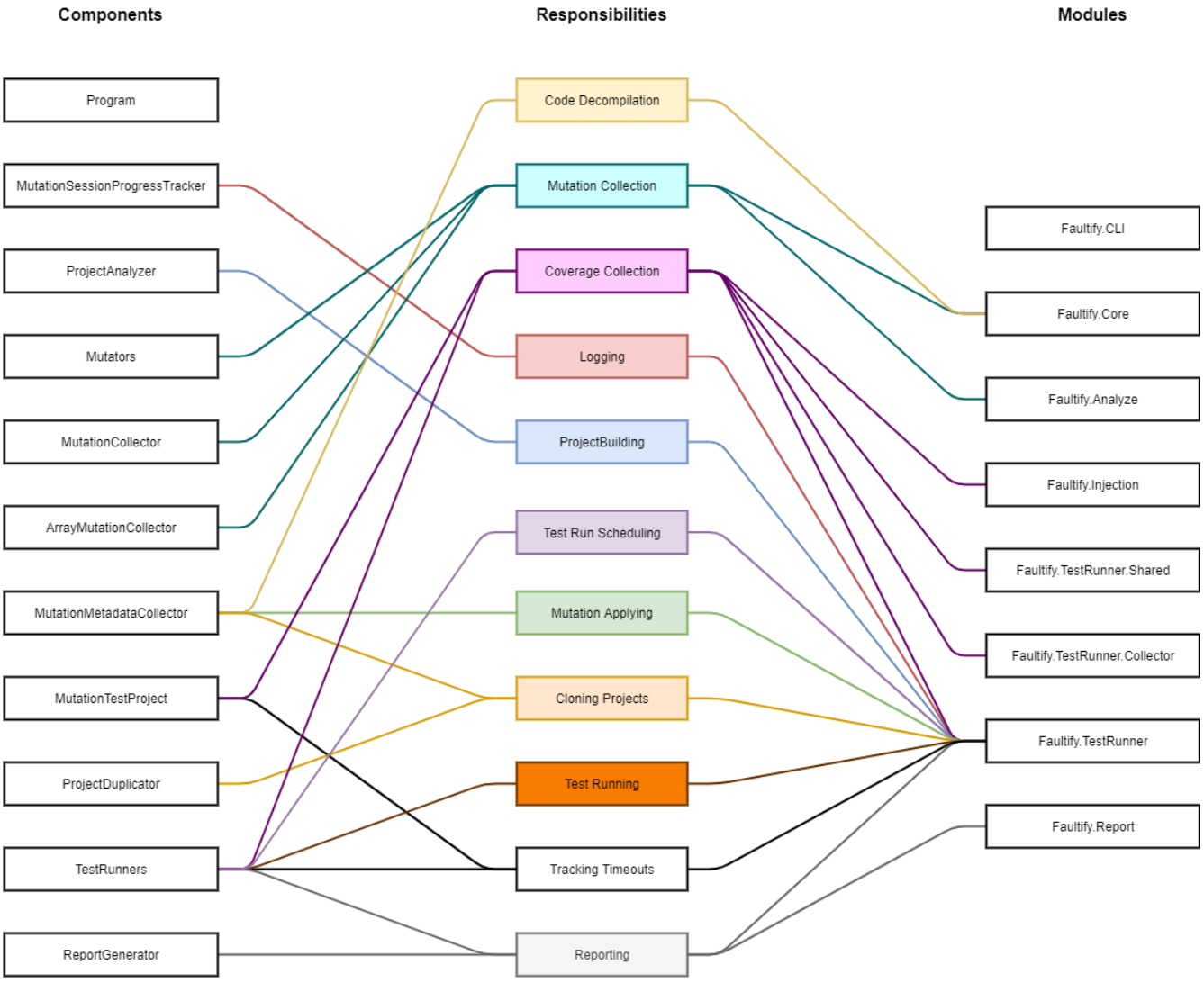

Figure 10: Mapping of components/modules to identified responsibilities.

2.4 Coupling Analysis

We can also draw some other observations based on the architecture recovery we did. In figure 7, we could also see the amount of dependencies between components. Additionally, in figure 12 we saw that modules map to components in a somewhat irregular way; some components consist of multiple modules, and some modules contribute to multiple components. Finally, in figure 10 we saw that similar things can be said about the mapping from modules to responsibilities.
Because of this confusing of responsibility, we were also curious whether there were any substantial problems with coupling in the system. As such, we also tried to analyze the coupling in the existing project.

We analyzed the coupling between classes using the *Coupling Between Objects (CBO)* metric. We chose this metric because it was easy to automate the analysis (in order to eliminate human error) using the class diagram we created earlier. The CBO metric is computed by simply counting the amount of classes a class is coupled to [7]. The results of this analysis are given in figure 11 (left). A CBO of 8 or higher is considered to be severely problematic [8]. We can see that a substantial amount of classes in figure 11 actually exceeds this threshold.
We did a similar analysis for the modules. For this, we counted the amount of outgoing dependencies from classes within a module to classes in other modules. The results are shown in figure 11 (right). We could not find any particular thresholds for this case in literature. An overview of all the module coupling is given in figure 12.
However, we once again, want to draw specific attention to the Faultify.TestRunner module. This module has the most dependencies of all modules: 61. It also contains the `MutationTestProject` class, which, with its 25 dependencies, has the most dependencies out of all classes. We consider this to be problematic, because it (significantly) exceeds the amount of dependencies of other modules. The Faultify.TestRunner modules is in fact responsible for 71% of all cross-module dependencies in the project.

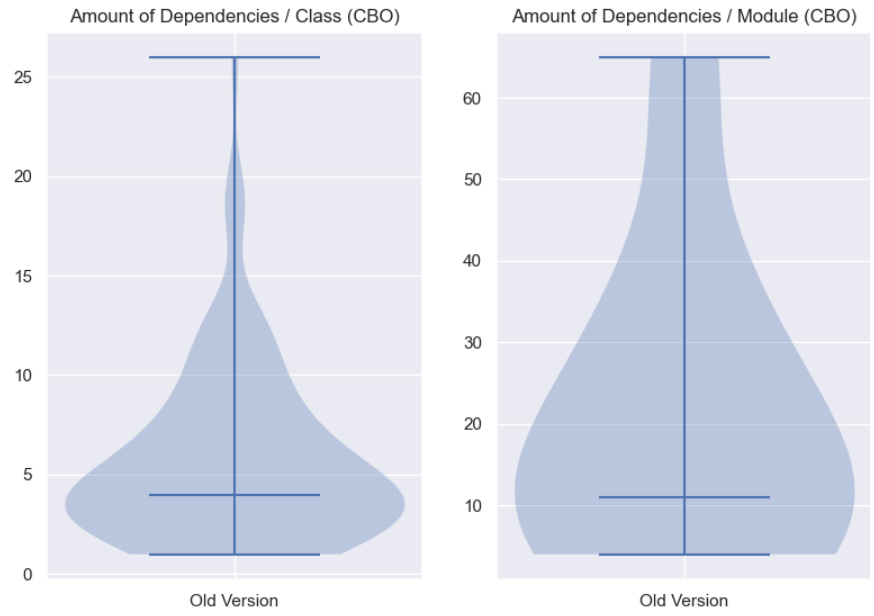

Figure 11: The CBO metric, computed for the old version of Faultify.

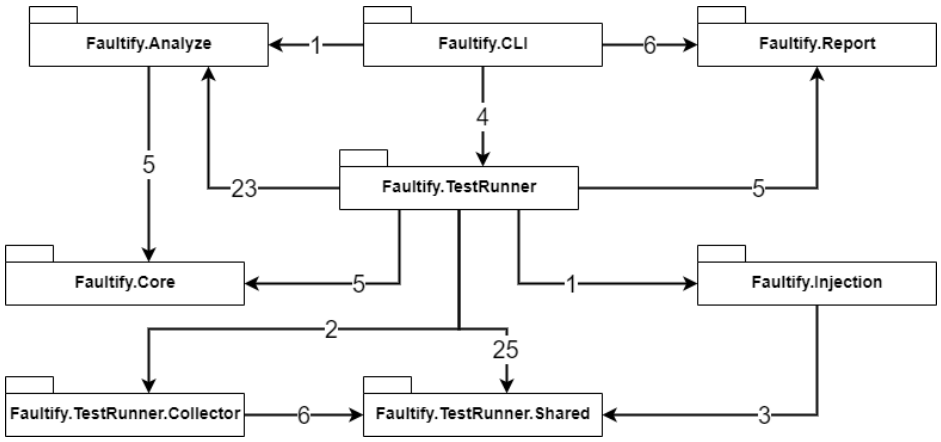

Figure 12: Coupling between modules in Faultify before the refactoring activities.

3 Proposal of Maintenance Activities

In this section, we will be discussing what maintenance activities we think best fit Faultify. We will be considering refactoring, testing, and adding new features.

In section 2.3, we observed that there are problems with responsibilities. We saw that there there is considerable confusion of responsibilities. Some modules have multiple responsibilities, while some responsibilities have been spread out across multiple modules. This is in fact a significant problem. Research has shown that a lack of well-divided responsibilities decreases programmer comprehension of a program [9, 10]. Especially large monolithic modules containing with many different responsibilities are detrimental to comprehension [10]. This can be a problem for maintainability, because it becomes more difficult to reason about a program.
We imagine that comprehension problems could be especially troublesome for a project such as Faultify. Because it is a student project, many different groups of students work on it. For a successful project, it is important that these students can understand the existing code. This could be more difficult than necessary because of the lack of proper adherence to single responsibility.
We also observed that the Faultify.TestRunner module had especially many responsibilities. We also observed that this module has a lot of dependencies (section 2.4), and contains by far the largest amount of code (section 2.2.3). These could be additional indicators that there might be too much going on in the Faultify.TestRunner module.
Therefore, we propose to restructure Faultify, in order to better divide the responsibilities across different modules.

In section 2.4, we looked at the coupling within the systems. We also saw that there is a substantial amount of coupling between classes. A substantial amount of classes exceeds the literature-suggested CBO threshold of 8 ([8]). Additionally, we saw that especially the Faultify.TestRunner module has a problematic amount of dependencies on classes outside its own module. It is a well-known phenomenon that a high amount of coupling reduces the maintainability of a system [11].
Because of this, we propose to refactor Faultify to reduce the amount of coupling between classes and modules.

In section 2.3.4, we noted that the sequence diagram for Faultify is not exactly linear, while the sequence of responsibilities is highly linear. We want to change the design so that this sequences becomes more apparent. Therefore, we propose to refactor the program so that the linear sequence of event becomes more clear.

In section 2.2.1, we saw that the current code coverage is rather low. Because of this, it would be beneficial to increase the amount of tests for the project. For instance, increasing code coverage is beneficial for issue resolving [12]. Hence, existing issues could be resolved faster, and it would be easier to detect regressions introduced during development.
However, during a manual look-through of the code, we found that the current code is very difficult to test (see section 2.2.2). As such, we propose to refactor the code such that it becomes more testable.
However, writing the actual tests was considered to be out of scope for this project. This was because we lost almost half of our group. As such, performing two maintenance activities was deemed infeasible.

Finally, during the static code analysis, we mentioned that there were a few instances of classes or methods with a high complexity (e.g. section 2.2.2). We will not explicitly address these concerns in this work, because we expect the

most problematic code to be refactored and changed along the way anyway. For instance, the `MutationTestProject` class was determined to be problematic both in terms of complexity (section 2.2.2) and coupling (section 2.4). Finally, we also saw the the MutationTestProject class takes care of multiple responsibilities (section 2.3.5). We expect that resolving the coupling and responsibility issues, will also resolve a large amount of the (cyclomatic) complexity issues for this class and other classes, based on the fact that we expect these classes to be taken apart and divided over multiple modules. Put short, complexity is a problem which should be addressed, but which we will be looking to fix while trying to resolve larger problems – and complexity will thus not be our main priority.

We will also briefly touch upon the feature evolution of this project. Faultify is in a reasonably mature state feature-wise. The main work left to do, is extending Faultify to work with multiple test frameworks. Most of the code for this is already present – just not fully functional. Additionally, the group of RUG students left a list of known issues which could be worked on – However, they recommended a major refactoring as the main future work. They did mentioned integration with Azure Devops Report and Pipeline.
However, based on our analysis, we believe it is more important to first refactor the system before adding any additional features. Additionally, we believe it is more important that the system is properly tested before adding new features; underlying issues must be resolved first, and it would be beneficial to detect any potentially introduced regressions.

4 Proposed Changes

In this chapter, we will be explaining the proposed changes to the code. First, we will be covering a re-division of responsibilities. After that, the methods for reducing coupling will be discussed.

4.1 Division of Responsibilities

The first thing we want to address is the poor division of responsibilities. In order to fix this, we want to restructure the existing code so that every responsibility is handled by a single module. The proposed restructuring is visualized in figure 13. We can see how the both components and responsibilities identified during the architecture recovery map onto the proposed architecture. We can see that most responsibilities are neatly divided.

We made two exceptions. First of all, both the coverage collections and test running are divided over two modules. This is because for both of these responsibilities, the underlying test framework has to be invoked. This shared functionality has been placed in a single separate module.

Additionally, mutation collections has been split into two separate parts. We did this to separate the two steps in collection mutations. First of all, we have a low-level collector (Faultify.MutationCollector). This module analyzes the byte code directly and generates the mutation. The Faultify.AssemblyDissection modules works at a high level; it operates at the assembly level (in C#, an assembly is a compiled collections of one or multiple classes). It can be used to deconstruct the assembly and obtain its classes, methods, and class fields. It thus operates at a higher level of abstraction than Faultify.MutationCollector.
We had two reasons for this division. Fist of all, we thought it would be more clear to divide the high and low level collection. Additionally, the Faultify.AssemblyDissection module can now be used by other modules for other purposes than mutation collection (e.g. checking the amount of classes in a project); the latter is something which occasionally happened, so we thought it would be cleaner to have this separate module.

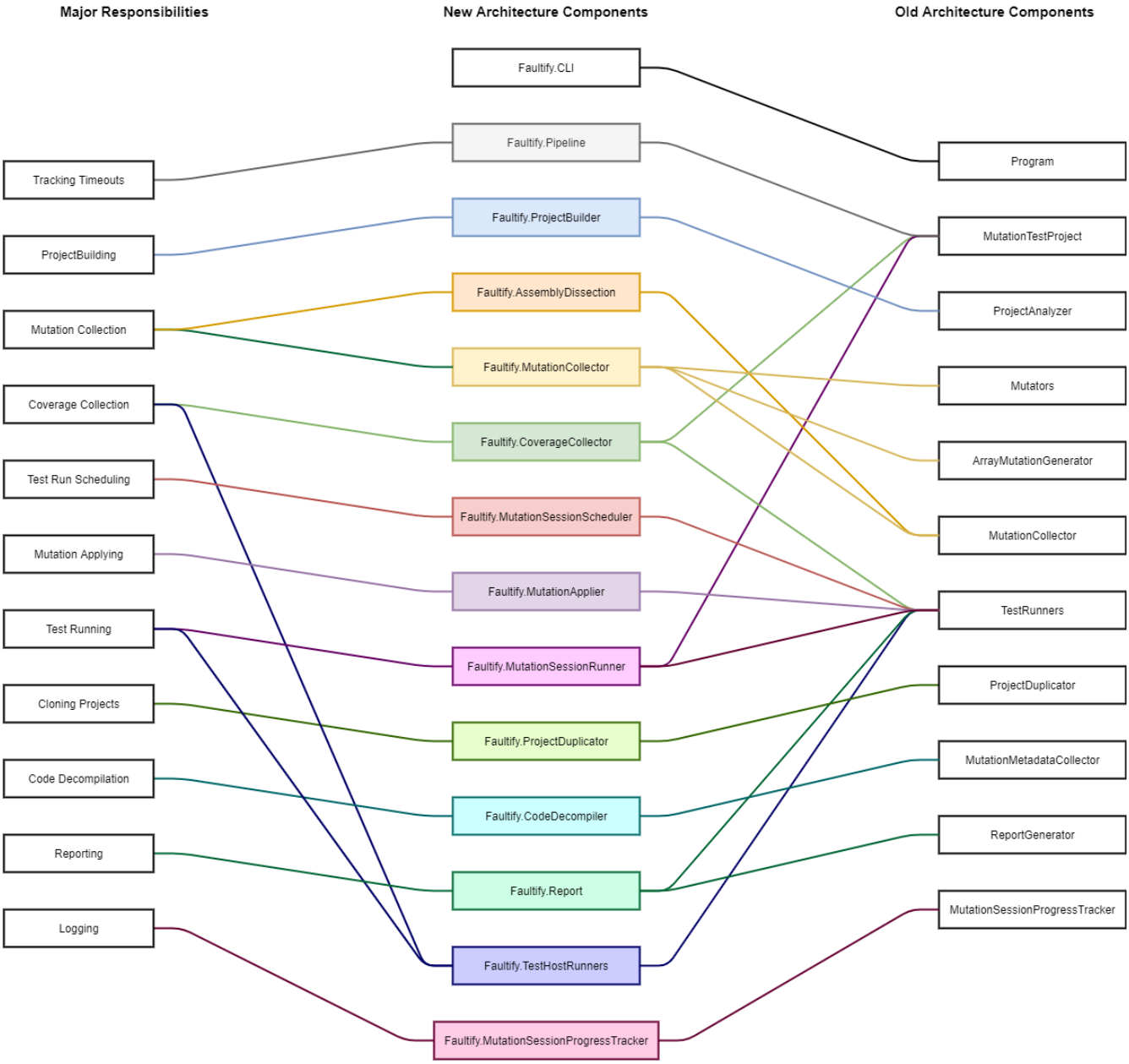

Figure 13: Proposed Module Restructuring.

4.2 Reducing Coupling

One of our objectives is to reduce the coupling in the project. In this section, we want to explain how we plan to achieve this.

4.2.1 Module Interfaces

One way we want to reduce coupling, is by minimizing the amount of functionality exposed from any given module. The functionality which is exposed, should be exposed through the use of interfaces in order to increase abstraction. One of our key observations is that most modules take in some inputs, and produce some results. Because of this, we decided to apply the factory pattern. Figure 14 shows how this works in practice. In every module, we create a factory which produces some result. Then, we identify an interface which the result object must implement. Then, we implement one or multiple classes implementing this interface, which can be returned by the factory [13]. Either the factory or those concrete result objects can use additional (helper) functionality contained within the module.

Classes external to the module should only depend on the interface for the result object, and possibly the factory object.

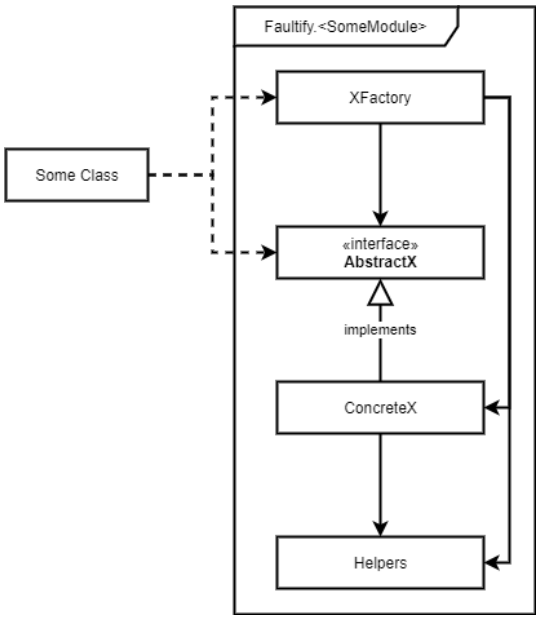

Figure 14: Application of the factory pattern. Based on [13]

4.2.2 Simplify Mutation Handling

Another way we want to reduce coupling is by simplifying the way mutations are handled and represented within Faultify. Currently, there are four classes in use for this purpose. `IMutation` objects contain the information required to apply a mutation. `IMutation` objects are stored inside `MutationGroup` objects, which contain the name and description of the analyzer (factory) used to find this mutation. Next, there are `MutationVariantIdentifiers`, which are classes which provide a unique identifier for mutations. They also contain the tests affected by a given mutation. Finally, there is the `MutationVariant` class, which contains a `MutationVariantIdentifier`, analyzer information, and various other metadata properties about a mutation. All of this is also represented in figure 15 (top part).
We wanted to simplify the way mutations are handled, by storing everything in one place: the `IMutation` object. This would reduce coupling, because less classes would be involved in places where mutations are handled. The plan to achieve this, consists of multiple steps:

1. Store analyzer name and description on the mutation objects, so that `MutationGroup`s can be replaced with simple `IEnumerable`s.
2. Store entity handles (reference to compiled objects) inside mutation objects in stead of the `MutationVariant` object
3. Store the name of the assembly containing a mutation on the mutation object
4. Associate mutations with their corresponding tests using tuples `(IMutation, Set<string>)`.
5. Use the `IMutation` object directly everywhere where a `MutationVariantIdentifier` is used. In practice, the identifiers are only used for easy representation, and later mutations have to be mapped to the identifiers again. By using the mutation objects directly everywhere, this mapping becomes redundant.
6. Create a `ReportData` object, which can contain all mutation source code, but also all (possibly not directly mutation related) data required for reporting. This class is necessary because we need a new way to track source code.

The new layout of the mutation object is given in figure 15 (bottom part).

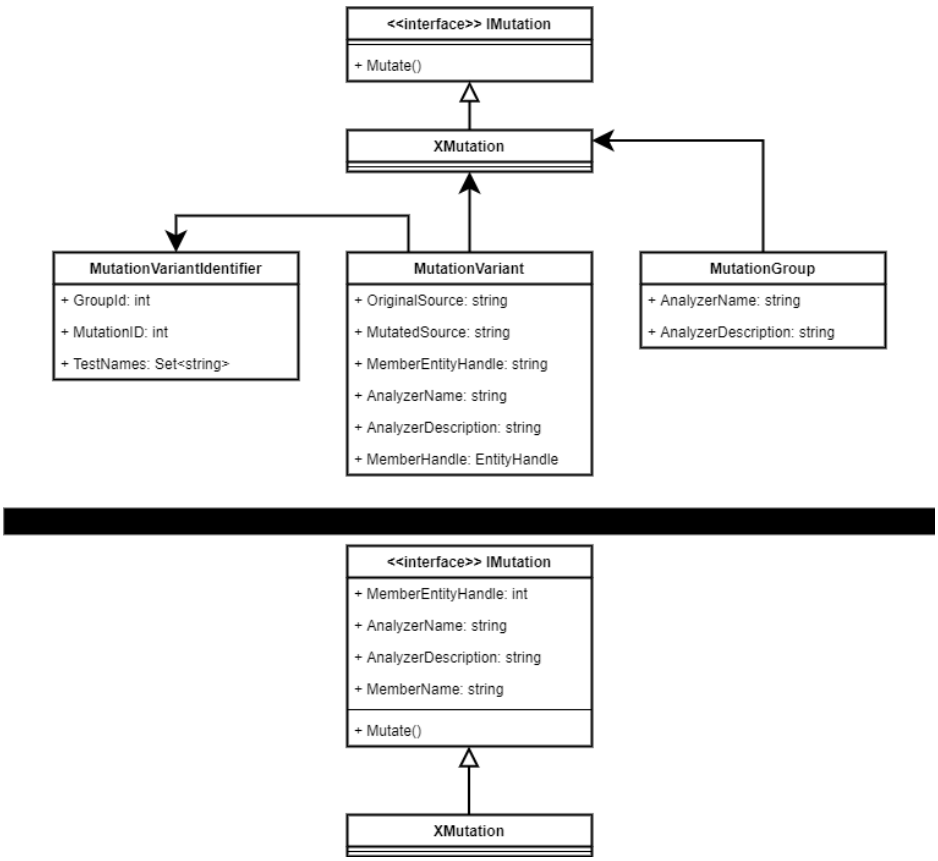

Figure 15: Old and new versions of the mutation handling. The top part is representing the old way of handling mutations, while the bottom part represents the new design.

4.2.3 Simplifying Return Objects

The final way we will try to eliminate coupling, is by simplifying returned objects. At times, dedicated classes containing results are returned, when simple built-in types would suffice. For instance, test runs are currently wrapped in a dedicated `DefaultMutationTestRun` object, while they could be represented as a simple list of mutations.

4.3 Linear Sequence of Steps

We also mentioned that we want to linearize the sequence of events in Faultify. We plan on achieving this using a pipeline, which invokes the modules one by one; it collects results and passes them to the next components in the pipeline. The pipeline will also be responsible for handling parallelism in the pipeline. All other modules will be unaware of the parallelism. Any required locking or similar activities will be handled in the pipeline, in order to not clutter the modules with details about parallelism.
Implementing this, together with the separation of concerns into separate modules, would lead to the sequence diagram given in figure 16. Note that this sequence diagram is not intended to be realistic. In stead, it is intended to represent the rough outline of the functionality of the pipeline, and how it will call different modules. It has been annotated with the same colors as figure 9 to denote responsibilities (see table 3). An unannotated version can be found in appendix A.
It should also be noted that we did not fully plan out the handling of timeouts in figure 16. At the design stage, we were unsure how and where we wanted to implement the tracking of timed out mutations. In stead, we evaluated the best approach after implementing a substantial amount of the changes. At that moment, is was decided that the MutationSessionRunner would return (as one of its results) a list of timed out mutations. A list of all timed out mutations are then tracked by the Pipeline itself.

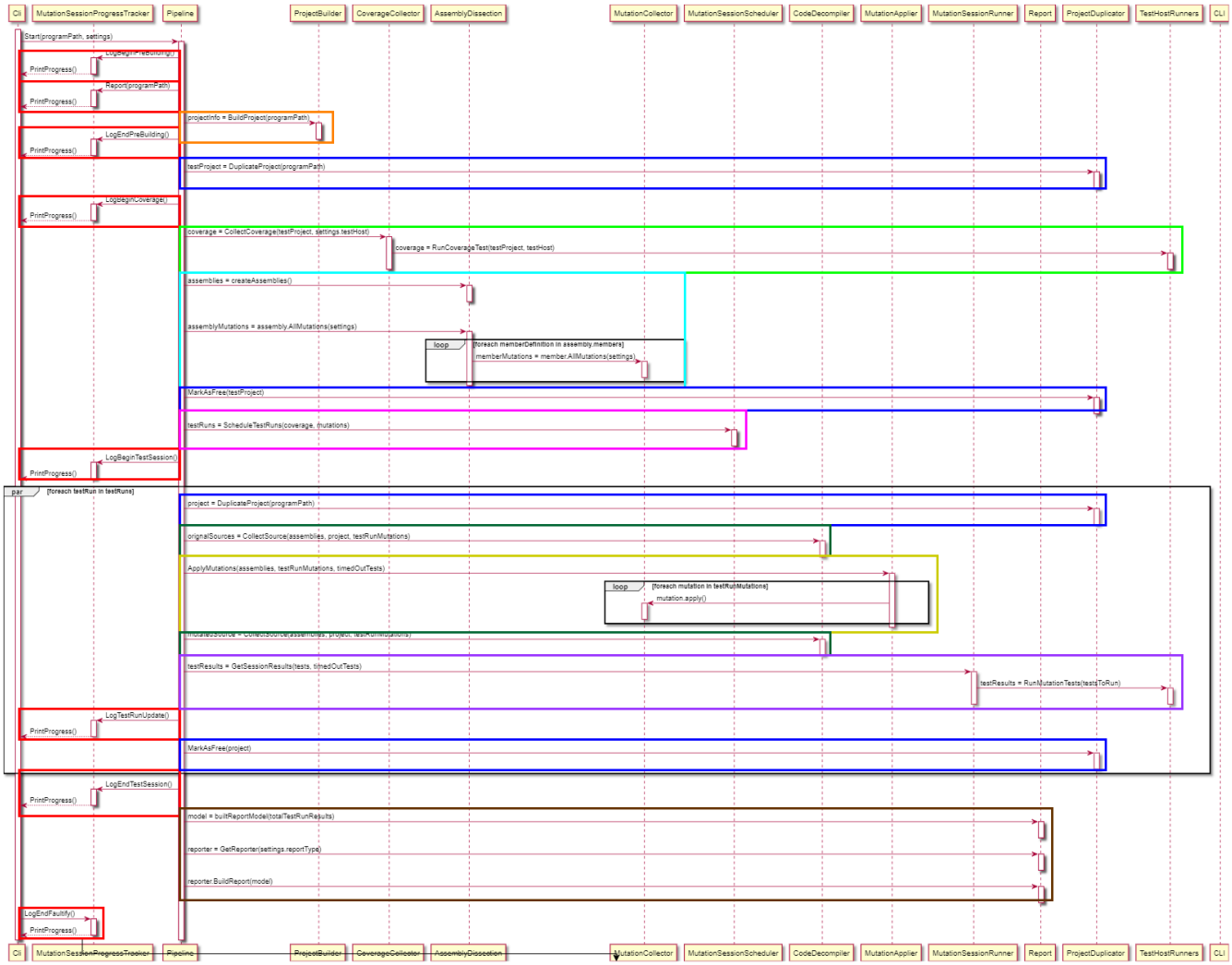

Figure 16: Conceptual design for the pipeline and its interactions with other modules

4.4 New Architecture

A complete design for the new architecture can be found in appendix B. We arrived at this architecture by applying the other changes proposed in this section to the existing architecture.

5 Overview of Development Process

In this section, we will be describing the outcomes of the development process. We will be looking at the final architecture, and will explain the reason for the major differences we can observe.

5.1 Final Architecture

First, we will cover the actual architecture we ended up with after performing the maintenance activities. A simplified version of the final class diagram is given in figure 30 (appendifx C). A complete version of this diagram with all classes and dependencies can be found in appendix D.
On a high level, we successfully implemented the architecture proposed in figure 29. However, some dependencies were wrong. Either we expected a dependency, which was not needed or vice versa.
The first difference between the conceptual and actual architecture is that the actual architecture has a dependency from the `ProjectDuplicator` to the `ProjectBuilder`. That is because the `ProjectBuilder` needs the

`IProjectInfo` interface with our current implementation. In hindsight, this dependency is not needed. See section 8 for more details about this.
Another unexpected dependency is the dependency from `Report` to the `MutationCollector` module. Because of the fact that we store an `IMutation` in the report data, we need that dependency. We inserted that dependency, since the report needs a large amount of data that the `IMutation` module contains. Therefore, it would have been counter intuitive to not have the dependency there; we would have had to extract a lot of data from the mutation beforehand and store it in some other object. This would increase the maintenance burden since adding any additional metadata from the mutation to the report would then in turn also require changing the reporting object.
There is also a dependency that turned out to be unnecessary. The dependency from `MutationSessionRunner` on `ProjectDuplication` was not needed. Since only the file path of the test project was needed in the `MutationSessionRunner`, we could pass the file path as a string, which removes the dependency. The same was the case for the `CoverageCollector`.
Another difference is that the conceptual architecture did not have the `CoverageMapper`. This was a class we did not think of when creating the conceptual architecture. Fortunately it is a rather small class that has a dependency on only the `IMutation` interface. It turns out this class was needed as a bridge between the CoverageCollector and the MutationSessionScheduler. It's return value is in a format which can be use efficiently by the latter of the two. As such, it was added as part of the MutationSessionScheduler module.

5.2 Sequence Diagram

We also created a sequence diagram denoting the interactions between modules in the new architecture. For clarity, we once again annotated this diagram with colored boxes to make clear where the main responsibilities are performed (legend in table 3. The diagram is presented in figure 17. An unannotated version can be found appendix A.
We can see that the diagram corresponds relatively nicely to the one shown in figure 16. Of course, this one is more detailed. There are two major difference.
We start by explaining the second addition, because it is required to understand the first one. The second one is the addition of the `EquivalentMutation()` method. This is a method which copies a mutation in such a way it can be applied to a different project than the original mutations. When mutations are created, they are given the information needed to apply the mutation to the project currently being analyzed. However, Faultify runs multiple test sessions in parallel. This means that projects are copied. However, this means that the set of mutations that initially is generated, only applies to one of those copies. To fix this problem, we added the `EquivalentMutation()` method which generates a mutation applicable to the current copy. We also call this re-binding mutations. This process was actually present in the original Faultify. However, we miss-understood its purposes and thought it was not necessary. As such, it was removed. Only later during development did we find out that this step was in fact necessary. We then introduced the `EquivalentMutation()` method. The introduction of this method meant that we now also store the class name of the class containing a mutation on the mutation object. This allows us to easily find the class in which the mutation occurred in the copy of the project.
The first difference with the conceptual sequence diagram one is the order of the coverage collection and initial round of mutation collection has been switched. This was done because the coverage collector inserts byte code into the program which actually collects the coverage data. This byte code would also be mutated, and would result in failure to re-bind mutations later on in the program (because the program would try to re-bind the mutations to projects without the injected byte code).

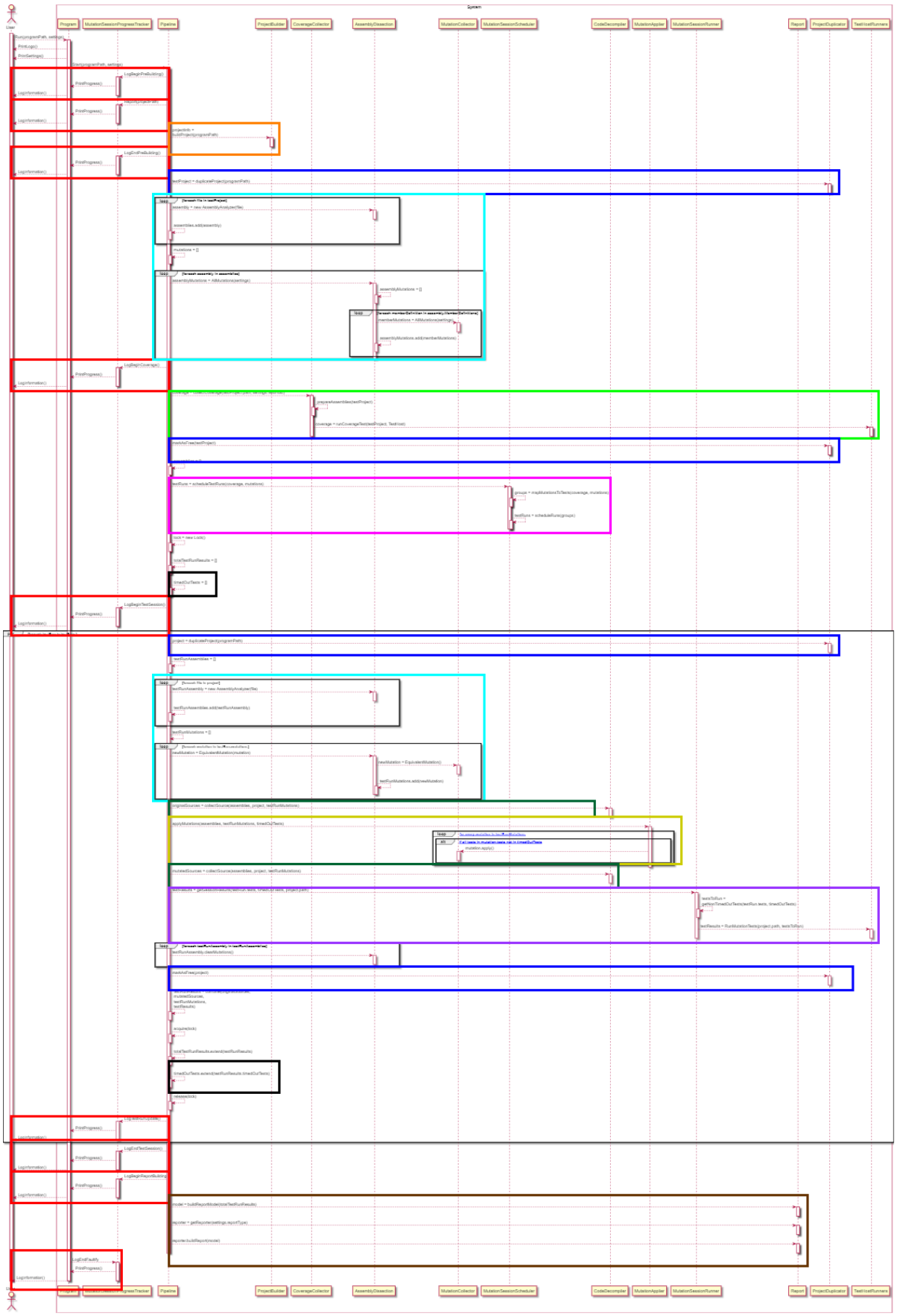

Figure 17: Sequence diagram denoting the interactions between modules in the new architecture.

5.3 Factory Pattern

As mentioned in section 4.2, we wanted to employ the factory pattern in order to reduce the amount of coupling and improve the maintainability of the system. We will show one particular example of this which we encountered. The particular example is illustrated in figure 18. Originally, we had the architecture presented on the left size of this diagram. All classes were located in a single module (Faultify.Analyze).

The main problem with the old architecture is that the `AssemblyMutator`, `TypeScope`, `FieldScope`, and `MethodScope` classes all need references to their various assembly analyzers. This can be reasonably manageable with the current amount of analyzers. However, adding a new analyzer (and possibly mutation type) requires updating many different classes.

We observed that we could avoid this problem by introducing a `MutationFactory` class. This class stores all analyzes and has methods which can be used to invoke all those analyzers for a given method or field. We even eliminated a dependency on the `OpcodeAnalyzer` class. This class is an abstract base class, which is only used for type definition where lists of its subclasses are required. However, we realized that all analyzers implement the `IAnalyzer` interface (not displayed to prevent cluttering the diagram). By using this interface, we were able to store any combination of analyzers in a list (which is often done in Faultify.)

Additionally, we were able to make sure that the `MutationFactory` needs no knowledge of all different mutation classes. In the old code, there already was an interface `IMutation` implemented by all mutations. However, it was only used effectively outside of the Faultify.Analyze module. Inside the module itself, the different mutation classes were often hard-coded directly (especially in type definitions). We could avoid this by simply using the existing `IMutation` interface.

It can easily be seen that this approach greatly decreased the amount of intra-modular dependencies. Additionally, it avoids needing many inter-module dependencies after splitting up the existing Faultify.Analyze module into Faultify.AssemblyDissection and Faultify.MutationCollector. The classes which belong to these modules have been highlighted with red and green, respectively.

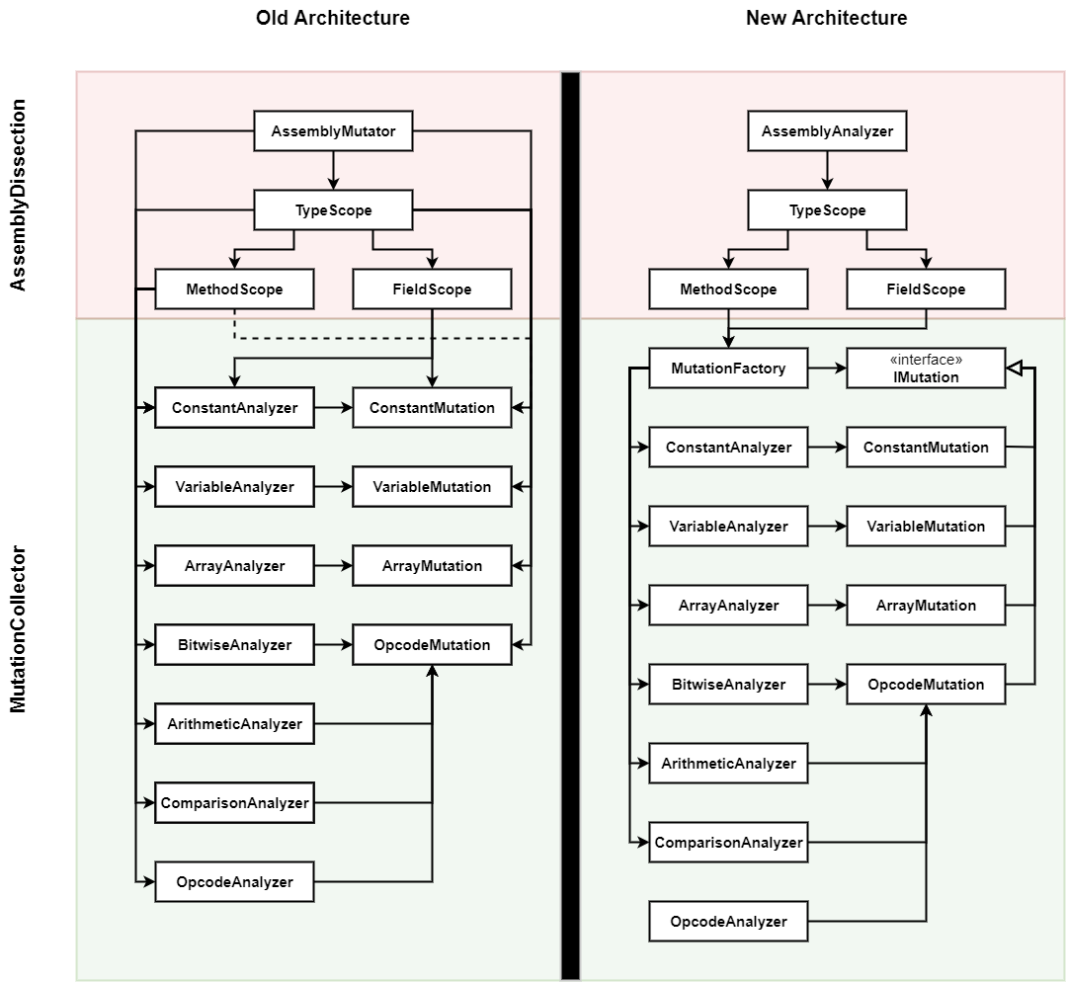

Figure 18: Example of an application of the factory pattern for generating mutations.

5.4 Detailed Change List

A detailed list of the changes per group member can be found in appendix E.

6 Evaluation

6.1 Static Code Analysis

We start out the evaluation by performing a static code analysis.

6.1.1 Code Coverage

Since increasing code coverage was not one of our maintenance tasks, the code coverage is not expected to be increased. However, it was not our purpose to break the tests. Therefore we have adapted the tests to the new code.

Most of the tests passed after the names and imports were corrected. However, there was a test for bitwise analyzers that returned an empty group when no mutations were found. In the new code, the group is not created at all. Although this is a slight change in behaviour, it is not a problem. Hence the test was changed to correspond to the changes we made.

The result of adapting the tests to the new code is that the code coverage barely changed (23% vs 20%, 605 lines vs 546 statements), as shown in figure 19 and figure 20. The difference in statements can be explained by the addition of statements in order to move more metadata into the mutation object themselves.

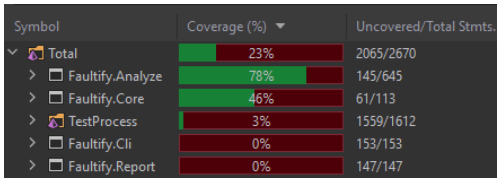

Figure 19: Code coverage old version

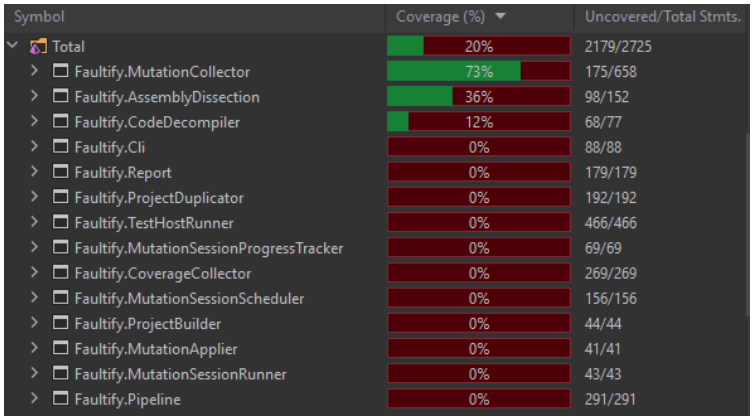

Figure 20: Code coverage new version

6.1.2 Lines of Code

The tables below show the (executable) lines of code in the old project (table 4) and in the new project (table 5).

Module	LOC	Executable LOC
Faultify.Analyze	1753	351
Faultify.Cli	370	105
Faultify.Core	309	49
Faultify.Report	277	88
Faultify.Injection	337	102
Faultify.TestRunner	1831	609
Faultify.TestRunner.Collector	168	55
Faultify.TestRunner.Shared	215	71

Table 4: Visual Studio code metrics: lines of code old version

Module	LOC	Executable LOC
Faultify.AssemblyDissection	453	94
Faultify.Cli	185	57
Faultify.CodeDecompiler	162	38
Faultify.CoverageCollector	496	152
Faultify.MutationApplier	54	20
Faultify.MutationCollector	2117	354
Faultify.MutationSessionProgressTracker	183	34
Faultify.MutationSessionRunner	81	17
Faultify.MutationSessionScheduler	318	76
Faultify.Pipeline	665	179
Faultify.ProjectBuilder	111	33
Faultify.ProjectDuplicator	398	79
Faultify.Report	367	98
Faultify.TestHostRunner	905	272

Table 5: Visual Studio code metrics: lines of code new version

As can be seen in the tables above, the problematic module `Faultify.TestRunner` has been split up. Therefore there is not a single module anymore that has over 600 lines of executable code. The `Faultify.Analyze` module

has essentially been changed to `Faultify.MutationCollector` and `Faultify.AssemblyDissection`. Since we did not plan to touch the part of the program that deals with mutations during the architecture refactoring, this module still contains about the same number of executable lines of code.

Violin plots have been created to give a visual representation of the executable lines of code for the modules. These can be found below in figure 21. Also in that figure it is clearly visible that the executable lines of code per module has been greatly reduced. This could indicate that the modules are more focused towards a single task and thus better adhere to the single responsibility principle [14].

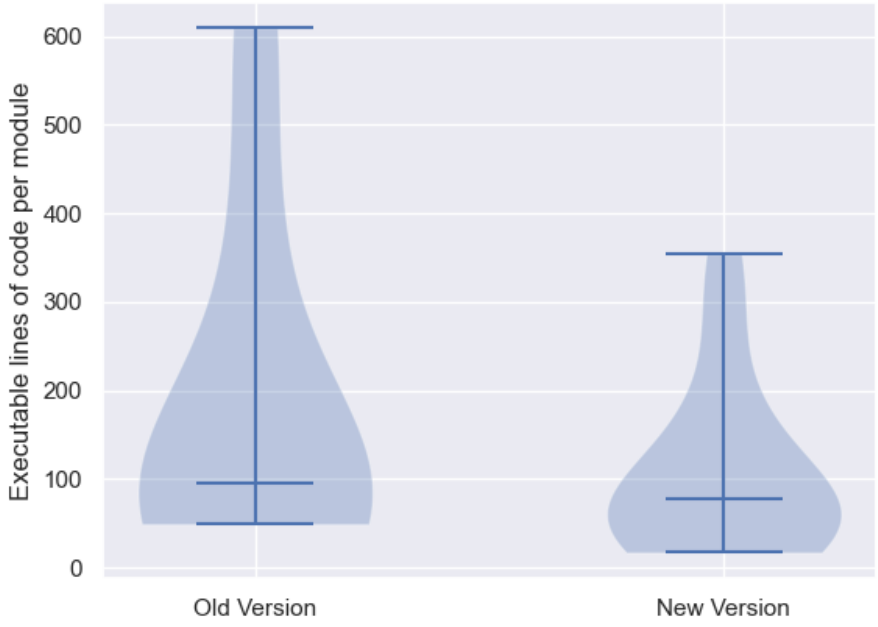

Figure 21: Executable lines of code per module, old version vs new version

6.1.3 Cyclomatic Complexity

The cyclomatic complexity shows how many different code paths there are in the program. The tables below contain the cyclomatic complexity of the old and the new version.

Module	Cyclomatic Complexity
Faultify.Analyze	262
Faultify.Cli	45
Faultify.Core	70
Faultify.Report	77
Faultify.Injection	35
Faultify.TestRunner	302
Faultify.TestRunner.Collector	13
Faultify.TestRunner.Shared	29

Table 6: Visual Studio code metrics: cyclomatic complexity old version

Module	Cyclomatic Complexity
Faultify.AssemblyDissection	64
Faultify.Cli	15
Faultify.CodeDecompiler	14
Faultify.CoverageCollector	47
Faultify.MutationApplier	5
Faultify.MutationCollector	285
Faultify.MutationSessionProgressTracker	35
Faultify.MutationSessionRunner	9
Faultify.MutationSessionScheduler	32
Faultify.Pipeline	57
Faultify.ProjectBuilder	25
Faultify.ProjectDuplicator	79
Faultify.Report	86
Faultify.TestHostRunner	80

Table 7: Visual Studio code metrics: cyclomatic complexity new version

The cyclomatic complexity of the `Faultify.MutationCollector` slightly increased compared to `Faultify.Analyze`. In the old version there is the `Faultify.TestRunner` module that has a cyclomatic complexity of 302. This highly complex module does not exist in the new version.

When comparing the two most complex modules from the old version with the two most complex version of the new version, we see massive improvement. In the old version the most complex modules had a complexity of 302 and 262, whereas in the new version the most complex modules have a complexity of 285 and 86. This is because a lot of modules, including `Faultify.TestRunner` have been split up into more modules where each module has a better defined responsibility.

To give a visualization of the numbers, violin plots have been created below. The plots clearly demonstrate a decrease in cyclomatic complexity. This is beneficial for reusability, understandability, maintainability and also testability [15, 5].

Figure 22: Cyclomatic complexity per module, old version vs new version

6.2 Division of Responsibilities & Linearization of responsibilities

In figure 17, we can see that the responsibilities in the system have been reasonably well divided. Every component performs its assigned task, in accordance with figure 13. In accordance with that diagram, only coverage collection, mutation collection, and running tests is done by multiple modules. We conclude that the responsibilities have been divided successfully, without any major unexpected cross-module interactions

In figure 17, we can also see that all responsibilities are executed sequentially by the pipeline. This means that we also succeeded in linearizing the sequence of events. This is also confirmed by the amount of nesting in the sequence diagram. In figure 9, the deepest level of nesting was 7, while it is 4 in figure 17.

6.3 Coupling

One of the goals of the refactoring was reducing the amount of coupling in the system. In this section, we will be performing an analysis to what extent this process succeeded. In order to do this, we once again compute the CBO values for all classes in the project. The results are presented in figure 23. We can see that the amount of dependencies has been substantially reduced when compared to the version of Faultify before the maintenance activities. The median amount of dependencies has been reduced. We can also see that a larger amount of the classes has a CBO below the threshold of 8 ([8]).

We also computed the amount of decencies on classes in other modules, for every module. The results of this analysis can be found in figure 24. As we can see, the amount of dependencies has been substantially reduced. In fact, the total amount of dependencies in this diagram has been reduced to 59, which is two less than the module which originally had the most dependencies (Faultify.TestRunner). An overview of all dependencies per module can be found in figure 25.

The class with the largest amount of dependencies in this new version of Faultify, is the `Pipeline` class. This is because this class calls every module in sequence. It thus depends on at least one class in every module. This explains the large dependency count. We tried to make up for this by making the code as clear as possible.
The Faultify.Pipeline modules is also the one with the largest amount of dependencies in the current project. It accounts for 51% of the cross-module dependencies. This is mostly because of the fact it contains the `Pipeline` class. It is still considerably better than the old Faultify.TestRunner module, which had 61 cross-module dependencies.

As part of our analysis, we will see how the dependencies of Faultify.TestRunner were reduced. According to figure 10, the Faultify.TestRunner module was divided across eight different modules. We can in turn use figure 25 in order to investigate how many all those modules have. These results can be found in table 8. We can see that almost all modules have very few or no dependencies. Only the Faultify.Pipeline module has many, but that is because of its intended purpose: calling all other modules to perform the mutation testing process.

In the end, we conclude that we successfully reduced the amount of coupling. Even the most problematic modules (in terms of coupling) have been split up into modules with few dependencies. Additionally, we were able to reduce most CBO scores to a value less than or equal to the advised threshold.

Module	**Dependencies**
Faultify.Pipeline	27
Faultify.CoverageCollector	5
Faultify.ProjectBuilder	0
Faultify.MutationSessionScheduler	3
Faultify.MutationApplier	1
Faultify.MutationSessionRunner	3
Faultify.Report	1
Faultify.ProjectDuplicator	0
Faultify.TestHostRunners	0
Faultify.MutationSessionProgressTracker	0

Table 8: Amount of dependencies of all modules which are (partially) composed of code from the Faultify.TestRunner module.

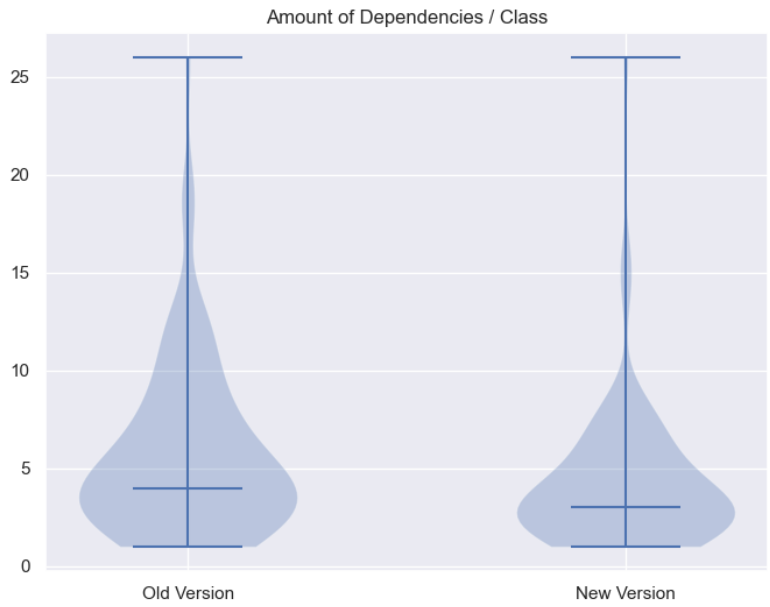

Figure 23: Comparison of the amount of dependencies before and after the maintenance activities.

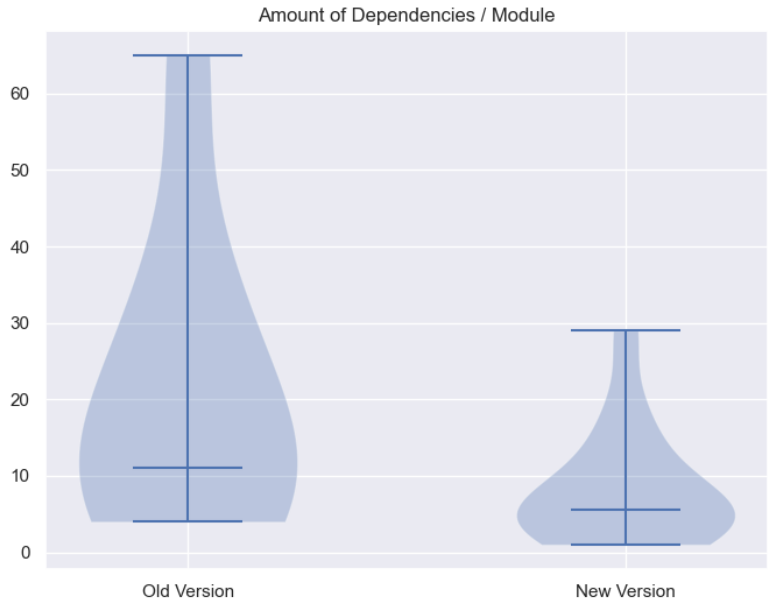

Figure 24: Comparison between the versions of Faultify before and after the maintenance activities, of the amount of dependencies on class in another module per module.

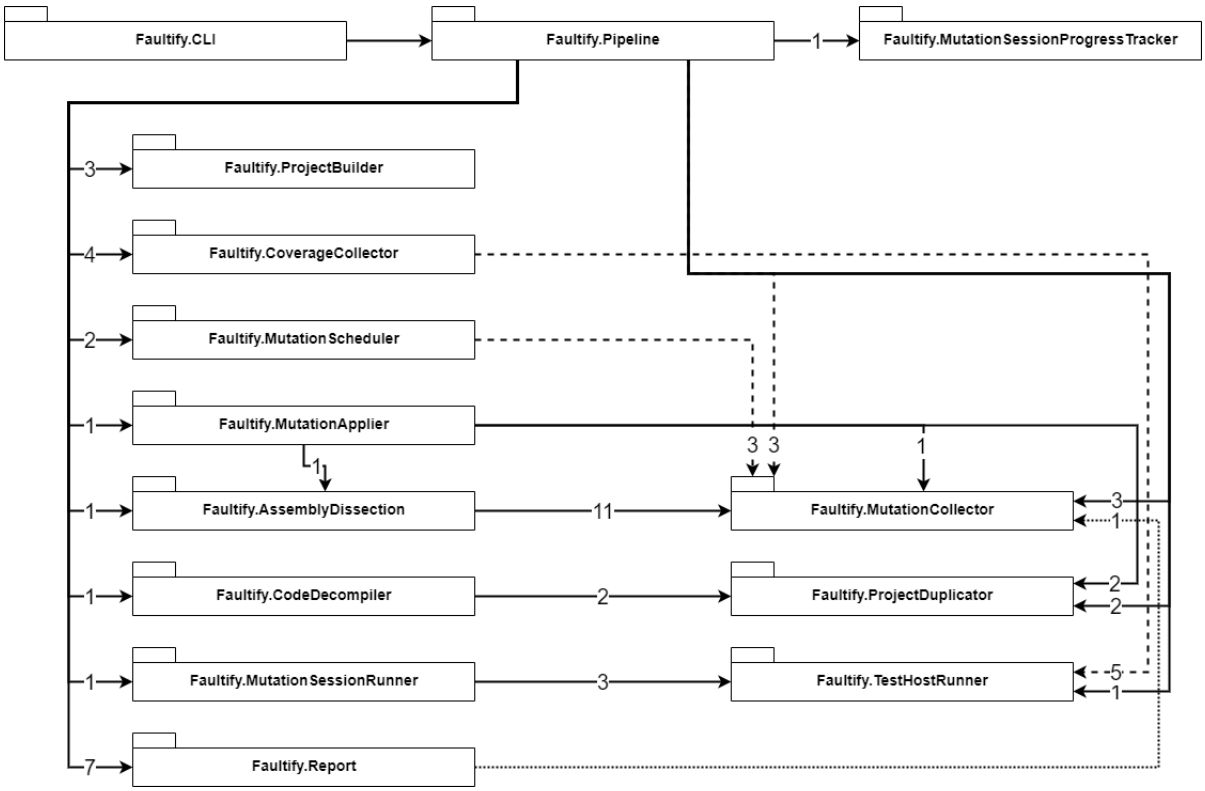

Figure 25: Amount of cross-module dependencies for all modules in the project.

6.4 Testing the New Architecture

The architecture of the program received a major overhaul in our development process. To verify that all the changes have not led to a malfunctioning program, we have performed some tests to make sure that this is not the case.

First of all, a small part of the original version contained unit tests. Since we were supposed to change the architecture and essentially not change the behaviour of the program, at least to some extend, the unit tests should still pass. Most of them did, of course after adapting them to the new architecture. Only one unit test was failing due to a slight change in behaviour. However, that change in behaviour is only on a low level and hence there is no change in behaviour of the program on a high level. Therefore this part is a success.

Furthermore we wanted the project to be in a buildable state. As expected, this was not possible in a first try and therefore we had to make some slight changes to remove the build errors. Fortunately, this process was far easier and took much less time than expected.

Since fixing build errors did not take a lot of time, we had more time to make the program runnable. Again and as expected, this was not possible in a first try. However, after 2 sessions of pair programming, we made the program runnable. We verified the outcome of the program using the provided test project. We verified that the output of the old version is the same as the output of the new version. Furthermore there were some problems with console logging. Apparently this was already a bug in the old version. However, we created a fix for this that improved console logging by a lot. Therefore this test can also been seen as a success.

7 Conclusion

In our analysis of Faultify, we found that responsibilities were ill-divided, and there was a lot of coupling between classes and modules. As a fix, we proposed a restructuring, a refactoring of module interfaces (by introducing factories), and a simplification of the objects used within the system (mutations and return objects). Finally, we ended up with an architecture where responsibilities are evenly divided across modules, with only one responsibility per module. Additionally, we greatly reduced the amount of coupling in the system. We also got Faultify in a working state again after all the major refactoring and restructuring activities. During the project, we did spot

some opportunities for future improvement, which are outlined in section 8.

8 Future Work

In this section, we will be presenting some immediate future work we identified while working on the project.

8.1 Known Issues

This section lists all the issues we found with Faultify while working with it. Debugging was considered to be out of the scope of this project. Hence, these issues have to be resolved later.

8.1.1 Bugs

During the initial test runs of the provided version of Faultify (also called old version in the document), we encountered an issue with the test framewrok MSTest. If Faultify needs to use this test framework, it crashes during the coverage analysis. The same behaviour is present when using XUnit as the test framework.
The only test framework that correctly works is NUnit. However, it only works on the provided benchmark project. We have tried several other projects that use NUnit as testing framework. Either the runs crash because the program fails to build the assemblies, with bad or no error reporting. Furthermore it sometimes fails to find mutations or tests on other projects than the benchmark.
Regarding logging there is one bug we have found. The last message of the console logger is not always printed, since the program exits too early. This should be resolved by waiting until the last message is logged.

8.1.2 Feature improvements

At the moment, only the `MethodScope` mutations are used in the `TypeScope`. The `FieldScope` mutations are ignored. These should be used as well. However, this might be difficult to implement, because the coverage analysis only tracks methods, and not fields. Hence, there would be no way to link fields to tests.
The branching analyzer was not used in the provided version of Faultify. Since we were not supposed to change the behaviour and wanted to keep the behaviour the same to test if our refactoring was successful, we did not enable this. In future work this could be investigated and possibly enabled. This can be done by adding the braching analyzer to the list of analyzers in the `MutationFactory` class.
Furthermore, the code coverage of the tests is small. Only one module is mostly tested, now the `Faultify.MutationCollector` module. Since we did not have the time to do this due to a small group size, this can be improved in future work.
There is also still the possibility to reduce the amount of dependencies. We found that `Faultify.ProjectDuplicator` does not have to depend on the `Faultify.ProjectBuilder` by passing around a string and a IEnumerable¡string¿ in the pipeline.

References

[1] Yue Jia and Mark Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678. DOI: 10.1109/TSE.2010.62.

[2] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer". In: *Computer* 11.4 (1978), pp. 34–41. DOI: 10.1109/C-M.1978.218136.

[3] JetBrains. *dotCover: The .NET Unit Test Runner and Code Coverage Tool*. URL: https://www.jetbrains.com/dotcover/. accessed: 17.12.2021.

[4] IntelliJ. *Basic Terms — dotCover*. URL: https://www.jetbrains.com/help/dotcover/dotCover__Basic_Concepts.html. accessed: 26.01.2022.

[5] Microsoft. *Code metrics values*. URL: https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022. accessed: 22.01.2022.

[6] Nenad Medvidovic and Vladimir Jakobac. "Using software evolution to focus architectural recovery". In: *Automated Software Engineering* 13.2 (Apr. 2006), pp. 225–256. ISSN: 1573-7535. DOI: 10.1007/s10515-006-7737-5. URL: https://doi.org/10.1007/s10515-006-7737-5.

[7] S.R. Chidamber and C.F. Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 476–493. DOI: 10.1109/32.295895.

[8] Satwinder Singh and K. S. Kahlon. "Object oriented software metrics threshold values at quantitative acceptable risk level". In: *CSI Transactions on ICT* 2.3 (Nov. 2014), pp. 191–205. ISSN: 2277-9086. DOI: 10.1007/s40012-014-0057-1. URL: https://doi.org/10.1007/s40012-014-0057-1.

[9] Apostolos Ampatzoglou et al. "Applying the Single Responsibility Principle in Industry: Modularity Benefits and Trade-Offs". In: *Proceedings of the Evaluation and Assessment on Software Engineering*. EASE '19. Copenhagen, Denmark: Association for Computing Machinery, 2019, pp. 347–352. ISBN: 9781450371452. DOI: 10.1145/3319008.3320125. URL: https://doi.org/10.1145/3319008.3320125.

[10] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. "The Effect of Modularization and Comments on Program Comprehension". In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE '81. San Diego, California, USA: IEEE Press, 1981, pp. 215–223. ISBN: 0897911466.

[11] Melis Dagpinar and Jens H Jahnke. "Predicting maintainability with object-oriented metrics-an empirical comparison". In: *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.* IEEE Computer Society. 2003, pp. 155–155.

[12] Dimitrios Athanasiou et al. "Test Code Quality and Its Relation to Issue Handling Performance". In: *Software Engineering, IEEE Transactions on* 40 (Nov. 2014), pp. 1100–1125. DOI: 10.1109/TSE.2014.2342227.

[13] Craig Larman. *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Iterative Development*. 3rd ed. Pearson Education, Inc., 2011.

[14] Robert C Martin, James Newkirk, and Robert S Koss. *Agile software development: principles, patterns, and practices*. Vol. 2. Prentice Hall Upper Saddle River, NJ, 2003.

[15] A. Madi, O.K. Zein, and Seifedine Kadry. "On the improvement of cyclomatic complexity metric". In: *International Journal of Software Engineering and its Applications* 7 (Jan. 2013), pp. 67–82.

A Unannotated Sequence Diagrams

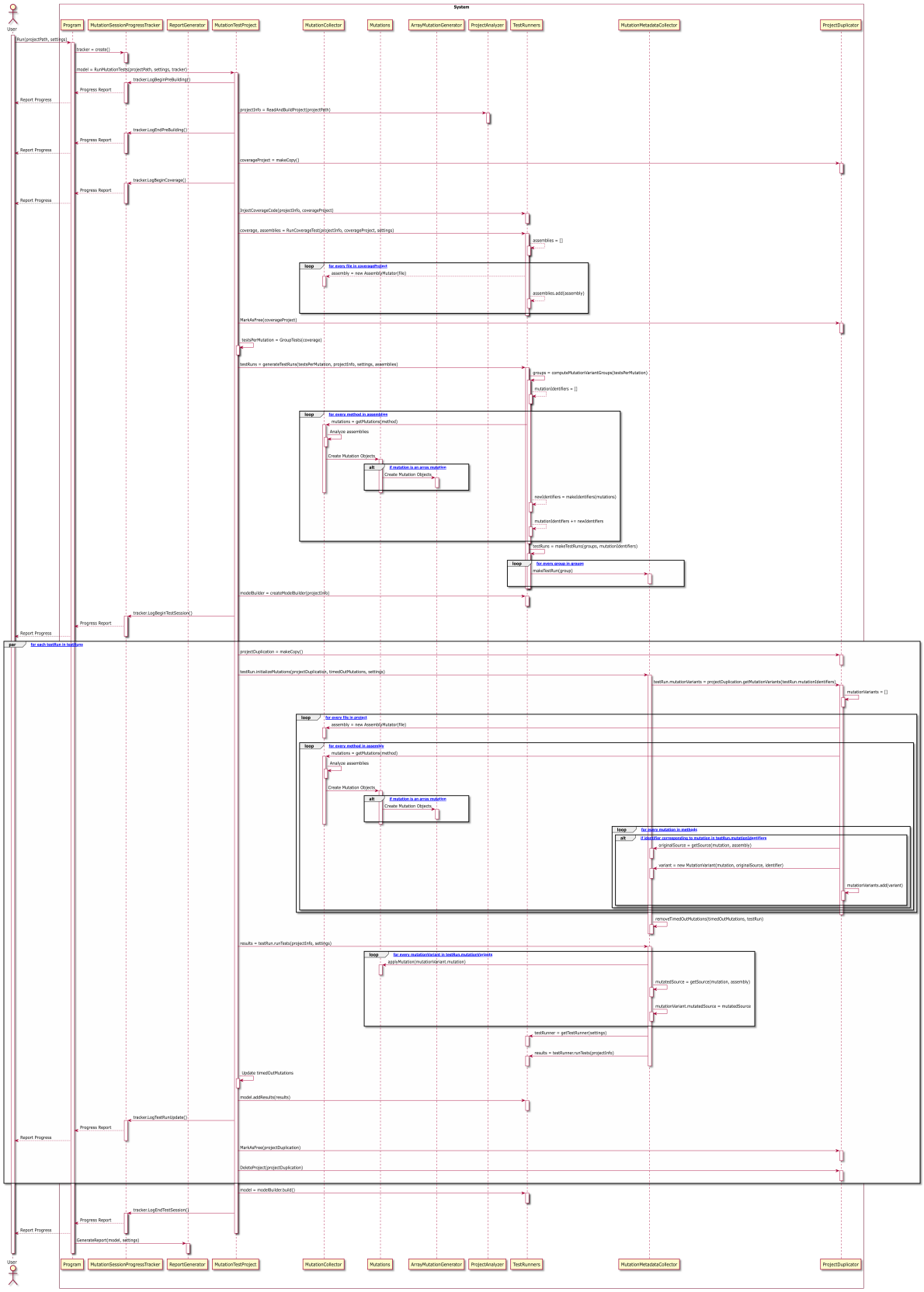

Figure 26: Unnannotated sequence diagram denoting interactions between components in the old architecture.

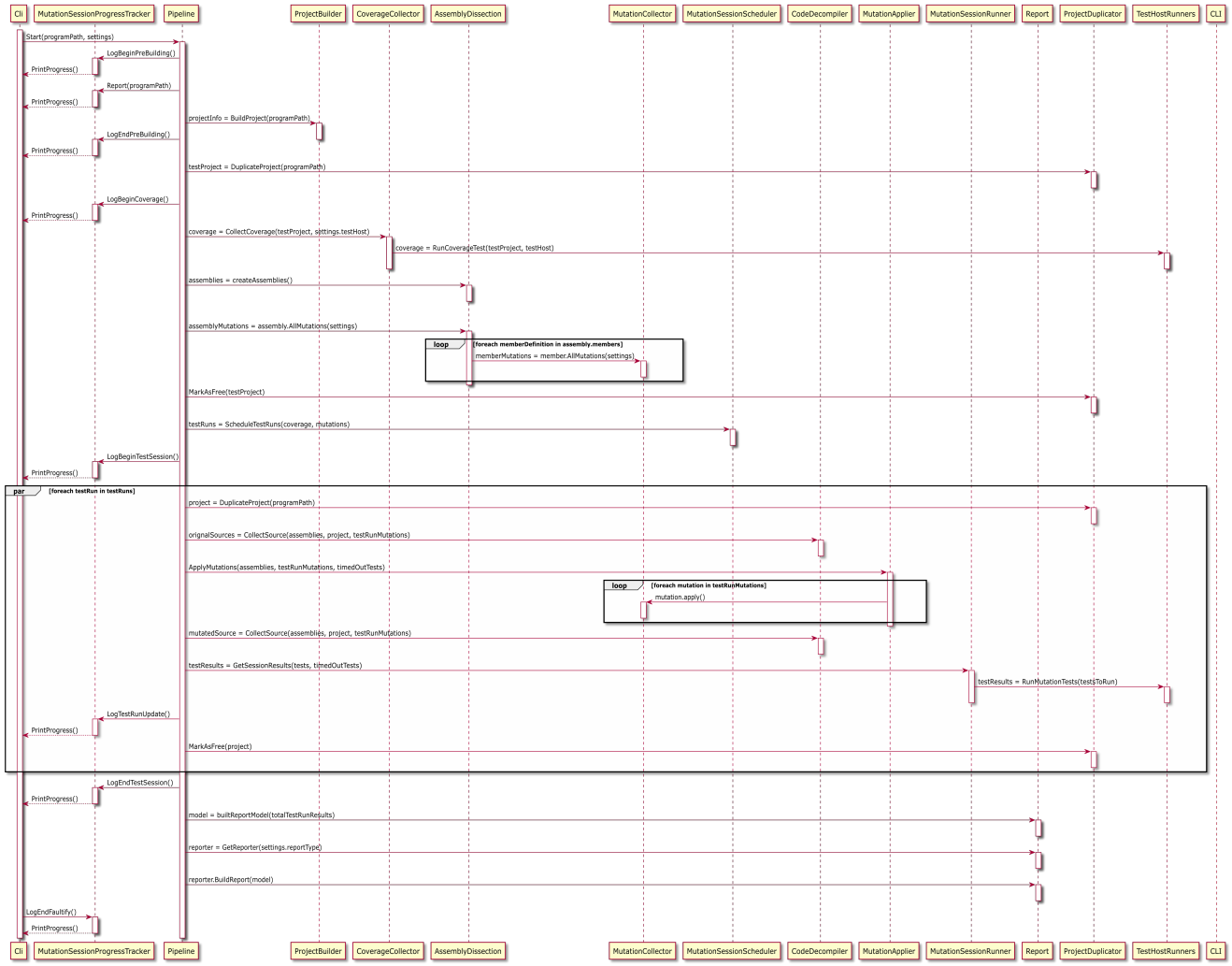

Figure 27: Unnannotated sequence diagram denoting interactions between components in the proposed/conceptual architecture.

Figure 28: Unnannotated sequence diagram denoting interactions between modules in the new architecture.

B Conceptual Architecture

In this section, we present the conceptual architecture which we want to implement.
By combining all our proposed changes, we come to the new proposed architecture represented in figure 29. This diagram for the architecture describes the new modules and the (rough) dependencies between them. The most important classes and interfaces are shown. Important dependencies between modules have also been included in the diagram. This denotes that a significant portion of left-out classes in a module depends on classes in another module.
We can clearly see how the pipeline depends on almost all modules, because it will have to call them sequentially. We also see the auxiliary modules TestHostRunner and ProjectDuplicator.

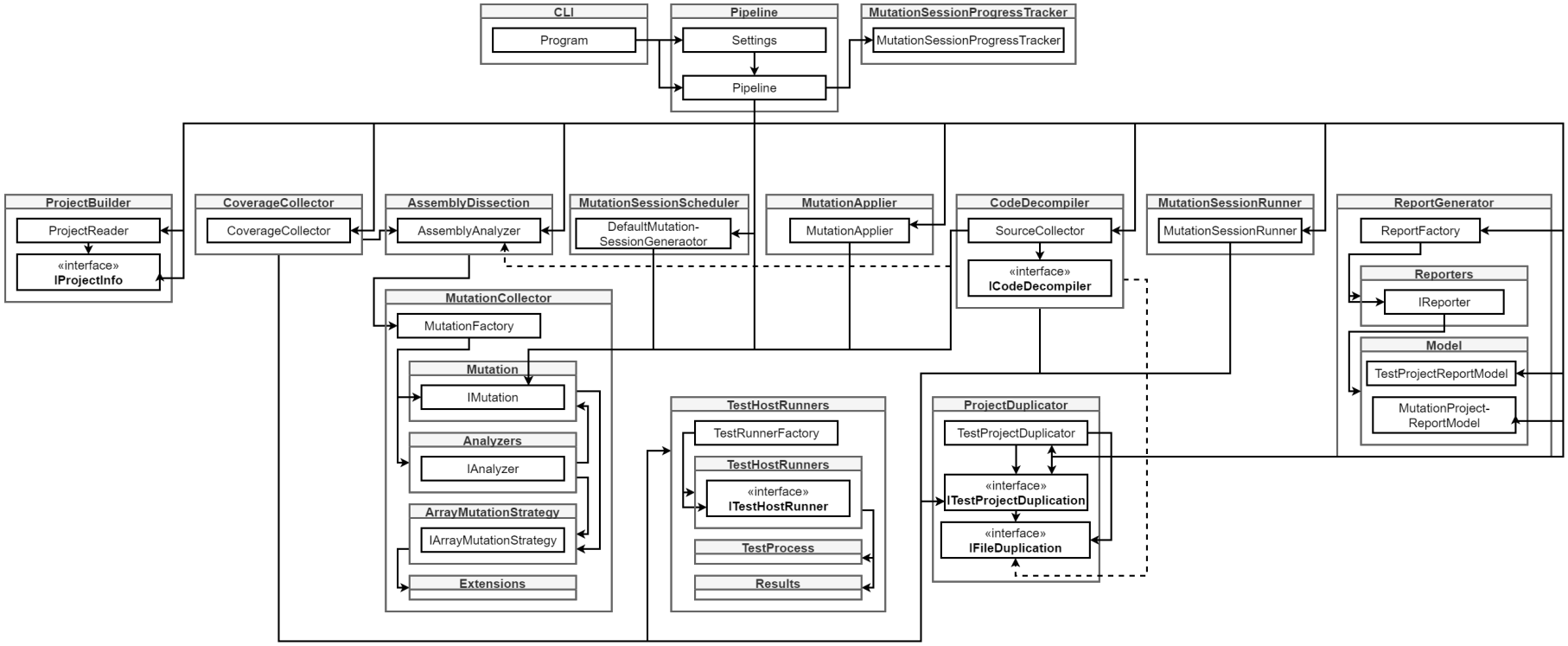

Figure 29: Conceptual architecture, to be implemented during the refactoring.

C Simplified Final Architecture

Simplified version of the final architecture presented in appendix D

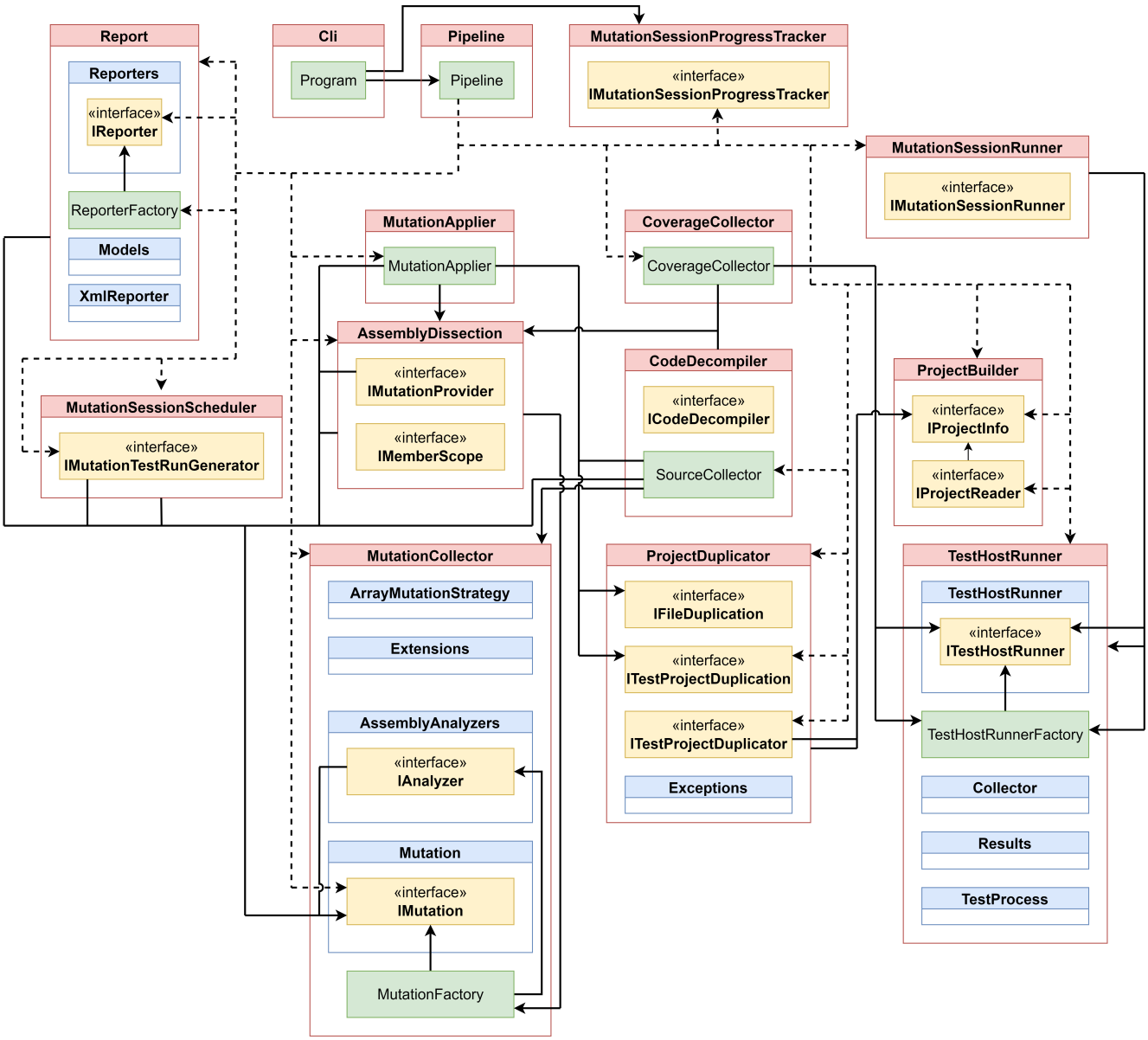

Figure 30: Resulting architecture after performing the maintenance activities

D Full Class Diagram

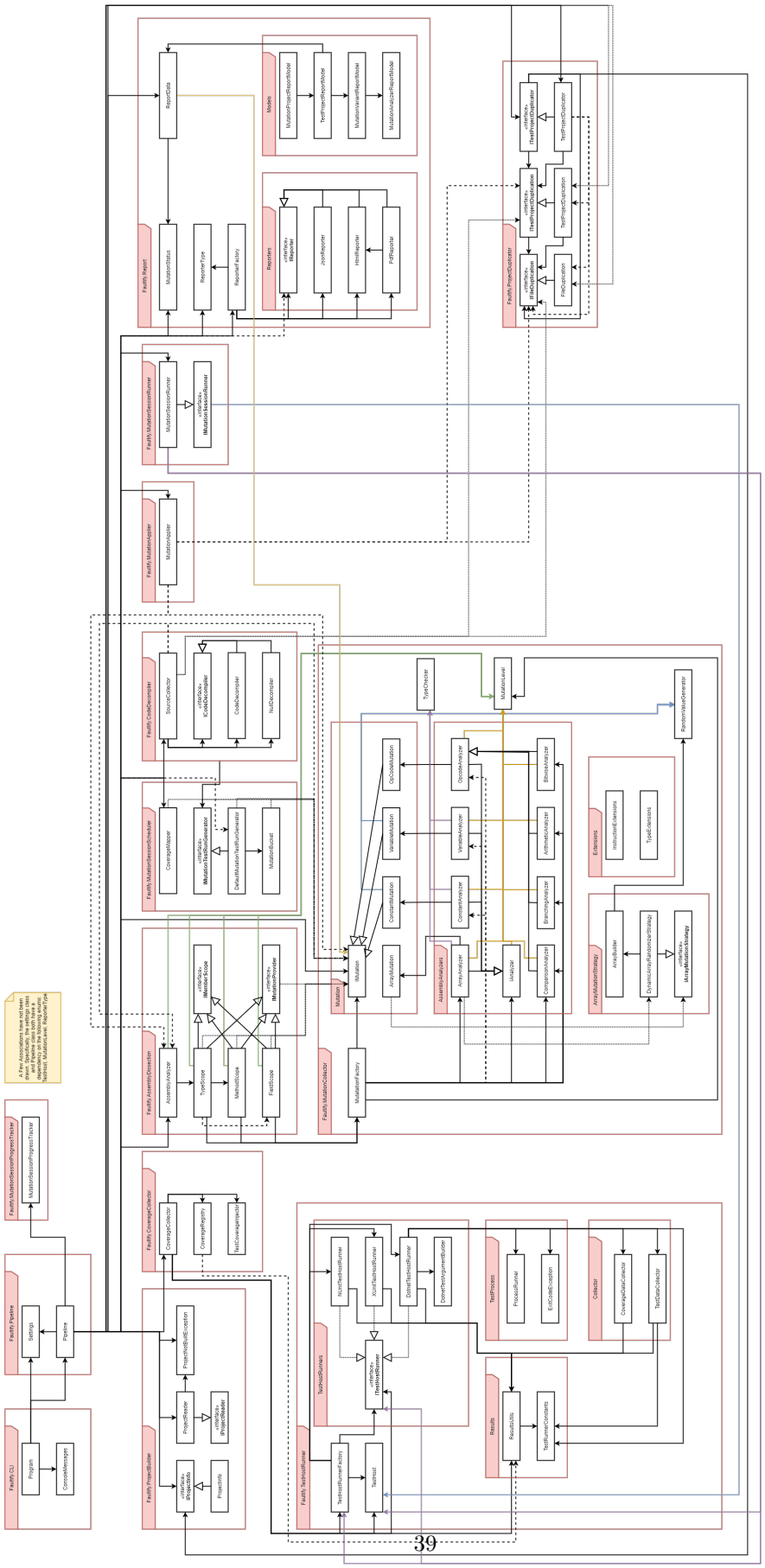

Figure 31: Complete class diagram of the project after performing the maintenance activities. Line style (connected, dotted, dashed) do not have any meaning, and are just for an easier understanding of the diagram.

E Changelog

This module contains two tables listing the work done by the team members. The first table give a rough outline of the work and timeline of the actual coding work. This table was reconstructed from the Git history. Only major (architecturally significant) commits were included.
The second table contains a full list of changes, all annotated with the member who implemented said change.
To refer to members, we use the following abbreviations:

- JM: Jesse Maarleveld
- AD: Arjan Dekker
- CW: Chris Worthington

Member	Date	Affected Files/Modules	Description
JM	23/12/2021	`Faultify.Analyze`, `Faultify.AssemblyDissection`, `Faultify.MutationCollector`, `Faultify.MutationCollector/MutationFactory.cs`	Divide contents of the `Faultify.Analyze` module across `Faultify.AssemblyDissection` and `Faultify.MutationCollector`. Add the `MutationFactory` class. Add analyzer metadata to the `IMutation` interface and implementing classes.
AD	23/12/2021	`Faultify.ProjectDuplication`, `Faultify.TestHostRunner`	Moved the parts of the `Faultify.TestRunner` module to their separate modules.
JM	24/12/2021	`Faultify.sln`	Resolve multiple merge conflicts
JM	24/12/2021	`Faultify.MutationCollector/Analyzers/LogicalAnalyers.cs`	Removed `LogicalAnalyzer` class because it is redundant.
JM	24/12/2021	`Faultify.Core.Extensions`, `Faultify.MutationCollector.Extensions`	Move the `Extensions` module from `Faultify.Core` to `Faultify.MutationCollector`
AD	24/12/2021	`Faultify.CodeDecompiler`, `Faultify.Report`	Moved parts of the `Faultify.TestRunner` module to their separate modules.
AD	26/12/2021	`Faultify.CoverageCollect`	Moved the coverage collecting from `Faultify.Injection` and `Faultify.TestRunner` to its own module.
JM	01/01/2022	`Faultify.TestRunner/TestRun/MutationVariant.cs`, `Faultify.TestRunner/TestRun/MutationVariantIdentifier.cs`	Remove `MutationVariant` and `MutationVariantIdentifier` because they will be made redundant.
JM	01/01/2022	`Faultify.AssemblyDissection/AssemblyMutator`	Rename `AssemblyMutator` class `AssemblyAnalyzer`
JM	01/01/2022	`Faultify.MutationCollector/Analyzers`	Re-introduce mutation groups (mutations are grouped by analyzers).
JM	02/01/2022	`Faultify.Core`, `Faultify.ProjectBuilder`	Move content from `Faultify.Core` to `Faultify.ProjectBuilder`
JM	04/01/2022	`Faultify.TestRunner/TestRun/DefaultMutationTestRunGenerator.cs`, `Faultify.TestRunner/TestRun/IMutationTestRunGenerator.cs`, `Faultify.TestRunner/TestRun/DefaultMutationTestRun.cs`, `Faultify.TestRunner/TestRun/IMutationTestRun.cs`, `Faultify.MutationSessionScheduler`	Move code for test run scheduling to its own separate module
JM	09/01/2022	`Faultify.MutationSessionScheduler`	Change interface of mutation scheduling code, and move `MutationBucket` to a separate file.

JM	09/01/2022	Faultify.AssemblyDissection, Faultify.MutationCollector, Faultify.MutationSessionScheduler, Faultify.MutationSessionScheduler/ CoverageMapper.cs	Store entity handle information on mutation objects. Add the CoverageMapper class which uses these entity handles to map methods to coverage information.
AD	09/01/2022	Faultify.CoverageCollector	Finished up the Faultify.CoverageCollector module based on Jesse's comments.
JM	10/01/2022	Faultify.ProjectDuplicator/ TestProjectDuplication.cs Faultify.MutationApplier	Create MutationApplier module and class, based on code from the TestProjectDuplication class.
JM	10/01/2022	Faultify.CodeDecompiler/ SourceCollector.cs	Add class for calling the code decompiler on a list of mutations. (in stead of one at a time)
AD	11/01/2022	Faultify.TestHostRunner	Updated the result classes of Faultify.TestHostRunner to the new representations of mutation data.
AD	13/01/2022	Faultify.CoverageCollector, Faultify.Pipeline	Moved part of the Faultify.CoverageCollector module to the pipeline. Also implemented the initial setup of the pipeline.
CW	13/01/2022	Faultify.Cli, Faultify.Report	Created factory for Faultify.Report and removed function from Program.cs. Moved JsonReporter.cs to Faultify.Report namespace. Created ReporterType.cs enum to be used by Settings.cs in Faultify.Cli
JM	14/01/2022	Faultify.AssemblyDissection, Faultify.MutationCollector	Implement the GetEquivalentMutation method, which computes copies of mutations which apply mutations in a different project duplication.
AD	14/01/2022	Faultify.CoverageCollector, Faultify.Pipeline	Improved Faultify.CoverageCollector and Faultify.Pipeline based on Jesse's comments.
AD	15/01/2022	Faultify.MutationSessionRunner	Moved and adapted code from Faultify.TestRunner to its own module Faultify.MutationSessionRunner. Also did a few merge requests.
CW	15/01/2022	Faultify.Report, Faultify.Cli	Moved input error handling from Faultify.Report to Faultify.Cli. Fixed dependency errors in Faultify.Cli. Fixed output of new enum to console in ConsoleMessage.cs
CW	16/01/2022	Faultify.Cli, Faultify.CodeDecompiler	Fixed more dependency errors in Faultify.Cli. Corrected .NET core version on Faultify.CodeDecompiler module
AD	17/01/2022	Faultify.Pipeline	Implemented calls to the Faultify.MutationCollector and Faultify.MutationSessionScheduler from Faultify.Pipeline.

CW	17/01/2022	`Faultify.Pipeline`, `Faultify.Cli`	Moved `GenerateReport()` function from `Faultify.Cli` to `Faultify.Pipeline`. Cleaned up unused code left in `Faultify.Cli`
AD	18/01/2022	`Faultify.Pipeline`	Completed the pipeline, except the reporting part.
JM	19/01/2022	`Faultify.Pipeline/Pipeline.cs`	Create copies of mutations for the current project copy in the pipeline
AD	19/01/2022	`Faultify.Pipeline`, `Faultify.Report`, `Faultify.Cli`	Finalized the `Faultify.Report` module and implemented calls to it in `Faultify.Pipeline`. Also finalized the `Faultify.Cli` and `Faultify.Pipeline` setup to work together.
CW	19/01/2022	`Faultify.Tests`, `Faultify.Cli`, `Faultify.MutationApplier`, `Faultify.MutationCollector`, `Faultify.MutationSessionProgressTracker`, `Faultify.ProjectDuplicator`, `Faultify.Report`, `Faultify.TestHostRunner`	Refactored all tests to work on the new project structure. Updated NLog and Buildalyzer packages on multiple modules for project to build successfully.
AD	20/01/2022	Multiple modules	Started fixing build errors.
JM, AD	21/01/2022	`Faultify.Pipeline/Pipeline.cs`	Fix runtime errors and bugs resulting in different output when compared to old version of Faultify.
JM	22/01/2022	`Faultify.Tests/UnitTests/AssemblyMutatorTests`	Fix a faulty tests.
AD	24/01/2022	`Faultify.Cli`, `Faultify.MutationSessionProgressTracker`	Removed unneeded passing around of the `Faultify.MutationSessionProgressTracker` and mostly fixed console logging bug.
CW, JM, AD	24/01/2022	`Faultify.Cli`	Added `await Task` on final logging call to make it more likely for final log message to appear before program halts.
JM	25/01/2022	`Faultify.MutationSessionRunner/MutationSessionRunner.cs`	Remove unneeded dependency on the `TestProjectDuplication` class.
AD	25/01/2022	Multiple modules	Removed unused or unneeded dependencies from some modules

Table 9: Changelog with individual member contributions

Code Artifact	**Action**	**Maintenance Activity**	**Team Member**
`Faulify.CLI.Program`	Removed reporting	Decoupling	CW
`Faulify.CLI.Program`	Fixed console logging	Bug fix	AD
`Faulify.CLI.Program`	Cli and pipeline work together	Restructuring	AD, CW
`Faulify.CLI.Settings`	Moved to `Faultify.Pipeline.Settings`	Decoupling	AD

Faultify.Analyze. ArrayMutationStrategy. ArrayMutationStrategy	Removed	Other	JM
Faultify.Analyze. ArrayMutationStrategy. DynamicArrayRandomizerStrategy	Moved to Faultify.MutationCollector. ArrayMutationStrategy. DynamicArrayRandomizerStrategy	Restructuring	JM
Faultify.Analyze. ArrayMutationStrategy. IArrayMutationStrategy	Moved to Faultify.MutationCollector. ArrayMutationStrategy. IArrayMutationStrategy	Restructuring	JM
Faultify.Analyze. ArrayMutationStrategy.ArrayBuilder	Moved to Faultify.MutationCollector. ArrayMutationStrategy.ArrayBuilder	Restructuring	JM
Faultify.Analyze. RandomValueGenerator	Moved to Faultify.MutationCollector. RandomValueGenerator	Restructuring	JM
Faultify.Analyze.Analyzers. ArithmeticAnalyzer	Moved to Faultify.MutationCollector. Analyzer.ArithmeticAnalyzer	Restructuring	JM
Faultify.Analyze.Analyzers. ArrayAnalyzer	Moved to Faultify.MutationCollector. Analyzer.ArrayAnalyzer	Restructuring	JM
Faultify.Analyze.Analyzers. BitwiseAnalyzer	Moved to Faultify.MutationCollector. Analyzer.BitwiseAnalyzer	Restructuring	JM
Faultify.Analyze.Analyzers. BranchingAnalyzer	Moved to Faultify.MutationCollector. Analyzer.BranchingAnalyzer	Restructuring	JM
Faultify.Analyze.Analyzers. ComparisonAnalyzer	Moved to Faultify.MutationCollector. Analyzer.ComparisonAnalyzer	Restructuring	JM
Faultify.Analyze.Analyzers. ConstantAnalyzer	Moved to Faultify.MutationCollector. Analyzer.ConstantAnalyzer	Restructuring	JM
Faultify.Analyze.Analyzers. LogicalAnalyzer	Removed	Other	JM
Faultify.Analyze.Analyzers. OpCodeAnalyzer	Moved to Faultify.MutationCollector. Analyzer.OpCodeAnalyzer	Restructuring	JM
Faultify.Analyze.Analyzers. VariableAnalyzer	Moved to Faultify.MutationCollector. Analyzer.VariableAnalyzer	Restructuring	JM
Faultify.Analyze.AssemblyMutator. AssemblyMutator	Moved to Faultify.AssemblyDissection. AssemblyAnalyzer	Restructuring	JM
Faultify.Analyze.AssemblyMutator. AssemblyMutator	Renamed to Faultify.AssemblyDissection. AssemblyAnalyzer	Other	JM
Faultify.Analyze.AssemblyMutator. FieldScope	Moved to Faultify.AssemblyDissection. FieldScope	Restructuring	JM
Faultify.Analyze.AssemblyMutator. IMemberScope	Moved to Faultify.AssemblyDissection. IMemberScope	Restructuring	JM
Faultify.Analyze.AssemblyMutator. IMutationProvider	Moved to Faultify.AssemblyDissection. IMutationProvider	Restructuring	JM
Faultify.Analyze.AssemblyMutator. MethodScope	Moved to Faultify.AssemblyDissection. MethodScope	Restructuring	JM

Faultify.Analyze.AssemblyMutator.TypeScope	Moved to Faultify.AssemblyDissection.TypeScope	Restructuring	JM
Faultify.Analyze.IAnalyzer	Moved to Faultify.MutationCollector.Analyzer.IAnalyzer	Restructuring	JM
Faultify.Analyze.IReportable	Removed	Decoupling & Simplification	JM
Faultify.Analyze.Mutation.ArrayMutation	Moved to Faultify.MutationCollector.Mutation.ArrayMutation	Restructuring	JM
Faultify.Analyze.Mutation.ConstantMutation	Moved to Faultify.MutationCollector.Mutation.ConstantMutation	Restructuring	JM
Faultify.Analyze.Mutation.OpCodeMutation	Moved to Faultify.MutationCollector.Mutation.OpCodeMutation	Restructuring	JM
Faultify.Analyze.Mutation.VariableMutation	Moved to Faultify.MutationCollector.Mutation.VariableMutation	Restructuring	JM
Faultify.Analyze.Mutation.IMutation	Moved to Faultify.MutationCollector.Mutation.IMutation	Restructuring	JM
Faultify.Analyze.MutationGroups.IMutationGroup	Removed	Decoupling & Simplification	JM
Faultify.Analyze.MutationGroups.MutationGroup	Removed	Decoupling & Simplification	JM
Faultify.Analyze.MutationLevel	Moved to Faultify.MutationCollector.MutationLevel	Restructuring	JM
Faultify.Analyze.TypeChecker	Moved to Faultify.MutationCollector.TypeChecker	Restructuring	JM
Faultify.AssemblyDissection.AssemblyAnalyzer	Add metadata (assembly name) to mutations	Decoupling	JM
Faultify.AssemblyDissection.AssemblyAnalyzer	Add GetEquilvalentMutation method	Bug fix	JM
Faultify.AssemblyDissection.TypeScope	Add metadata (assembly name, type name) to mutations	Decoupling	JM
Faultify.AssemblyDissection.TypeScope	Add GetEquivalentMutation method	Bug fix	JM
Faultify.AssemblyDissection.FieldScope	Add metadata (assembly name, type name, field name, entity handle) to mutations	Decoupling	JM
Faultify.AssemblyDissection.FieldScope	Add GetEquivalentMutation method	Bug fix	JM
Faultify.AssemblyDissection.MethodScope	Add metadata (assembly name, type name, field name, method name, entity handle) to mutations	Decoupling	JM
Faultify.AssemblyDissection.MethodScope	Add GetEquivalentMutation method	Bug fix	JM

Faultify.AssemblyDissection. IMutationProvider	Add GetEquivalentMutation method	Bug fix	JM
Faultify.CodeDecompiler. SourceCollector	Added	Other	JM
Faultify.CodeDecompiler. SourceCollector	Use parentMethodEntityHandle for mapping coverage to mutations	Bug fix	AD
Faultify.Core.Exceptions. ProjectNotBuiltException	Moved to Faultify.ProjectBuilder. ProjectNotBuiltException	Restructuring, Decoupling	AD
Faultify.Core.Extensions. InstructionExtensions	Moved to Faultify.MutationCollector. Extensions.InstructionExtensions	Restructuring, Decoupling	JM
Faultify.Core.Extensions. TypeExtensions	Moved to Faultify.MutationCollector. Extensions.TypeExtensions	Restructuring, Decoupling	JM
Faultify.Core.ProjectAnalyzing. CodeDecompiler	Moved to Faultify.CodeDecompiler. CodeDecompiler	Restructuring, Decoupling	AD
Faultify.Core.ProjectAnalyzing. ICodeDecompiler	Moved to Faultify.CodeDecompiler. ICodeDecompiler	Restructuring, Decoupling	AD
Faultify.Core.ProjectAnalyzing. IProjectInfo	Moved to Faultify.ProjectBuilder. IProjectInfo	Restructuring, Decoupling	JM
Faultify.Core.ProjectAnalyzing. IProjectReader	Moved to Faultify.ProjectBuilder. IProjectReader	Restructuring, Decoupling	JM
Faultify.Core.ProjectAnalyzing. NullDecompiler	Moved to Faultify.CodeDecompiler. NullDecompiler	Restructuring, Decoupling	AD
Faultify.Core.ProjectAnalyzing. ProjectInfo	Moved to Faultify.ProjectBuilder.ProjectInfo	Restructuring, Decoupling	JM
Faultify.Core.ProjectAnalyzing. ProjectReader	Moved to Faultify.ProjectBuilder. ProjectReader	Restructuring, Decoupling	JM
Faultify.CoverageCollector. CoverageCollector	Added	Decoupling	AD
Faultify.Injection. TestCoverageInjector	Moved to Faultify.CoverageCollector. TestCoverageInjector	Restructuring, Decoupling	AD
Faultify.Injection.CoverageRegistry	Moved to Faultify.CoverageCollector. CoverageRegistry	Restructuring, Decoupling	AD
Faultify.Injection.CoverageRegistry	Changed RegisteredCoverage to a Tuple	Decoupling, simplification	AD
Faultify.MutationApplier. MutationApplier	Added	Restructuring	JM
Faultify.MutationCollector. AssemblyAnalyzers.ArithmeticAnalyzer	Add metadata (assembly name, type name, field name, method name, entity handle, analyzer name, analyzer description) to mutations	Decoupling	JM
Faultify.MutationCollector. AssemblyAnalyzers.ArrayAnalyzer	Add metadata (assembly name, type name, field name, method name, entity handle, analyzer name, analyzer description) to mutations	Decoupling	JM

`Faultify.MutationCollector.AssemblyAnalyzers.BitwiseAnalyzer`	Add metadata (assembly name, type name, field name, method name, entity handle, analyzer name, analyzer description) to mutations	Decoupling	JM
`Faultify.MutationCollector.AssemblyAnalyzers.BranchingAnalyzer`	Add metadata (assembly name, type name, field name, method name, entity handle, analyzer name, analyzer description) to mutations	Decoupling	JM
`Faultify.MutationCollector.AssemblyAnalyzers.ComparisonAnalyzer`	Add metadata (assembly name, type name, field name, method name, entity handle, analyzer name, analyzer description) to mutations	Decoupling	JM
`Faultify.MutationCollector.AssemblyAnalyzers.ConstantAnalyzer`	Add metadata (assembly name, type name, field name, method name, entity handle, analyzer name, analyzer description) to mutations	Decoupling	JM
`Faultify.MutationCollector.AssemblyAnalyzers.ConstantAnalyzer`	Use parentMethodEntityHandle for mapping coverage to mutations	Bug fix	AD
`Faultify.MutationCollector.AssemblyAnalyzer.ConstantAnalyzer`	Use parentMethodEntityHandle for mapping coverage to mutations	Bug fix	AD
`Faultify.MutationCollector.AssemblyAnalyzers.IAnalyzer`	Add metadata (assembly name, type name, field name, method name, entity handle, analyzer name, analyzer description) to mutations	Decoupling	JM
`Faultify.MutationCollector.AssemblyAnalyzers.IAnalyzer`	Use parentMethodEntityHandle for mapping coverage to mutations	Bug fix	AD
`Faultify.MutationCollector.AssemblyAnalyzers.OpCodeAnalyzer`	Add metadata (assembly name, type name, field name, method name, entity handle, analyzer name, analyzer description) to mutations	Decoupling	JM
`Faultify.MutationCollector.AssemblyAnalyzers.OpCodeAnalyzer`	Use parentMethodEntityHandle for mapping coverage to mutations	Bug fix	AD
`Faultify.MutationCollector.AssemblyAnalyzers.VariableAnalyzer`	Add metadata (assembly name, type name, field name, method name, entity handle, analyzer name, analyzer description) to mutations	Decoupling	JM
`Faultify.MutationCollector.AssemblyAnalyzers.VariableAnalyzer`	Use parentMethodEntityHandle for mapping coverage to mutations	Bug fix	AD
`Faultify.MutationCollector.Mutation.ArrayMutation`	Add metadata (assembly name, type name, field name, method name, entity handle, analyzer name, analyzer description) to mutations	Decoupling	JM
`Faultify.MutationCollector.Mutation.ArrayMutation`	Add `GetEquivalentMutation` method	Bug fix	JM
`Faultify.MutationCollector.Mutation.ArrayMutation`	Use parentMethodEntityHandle for mapping coverage to mutations	Bug fix	AD
`Faultify.MutationCollector.Mutation.ConstantMutation`	Add metadata (assembly name, type name, field name, method name, entity handle, analyzer name, analyzer description) to mutations	Decoupling	JM
`Faultify.MutationCollector.Mutation.ConstantMutation`	Add `GetEquivalentMutation` method	Bug fix	JM

`Faultify.MutationCollector.Mutation.ConstantMutation`	Use parentMethodEntityHandle for mapping coverage to mutations	Bug fix	AD
`Faultify.MutationCollector.Mutation.IMutation`	Add metadata (assembly name, type name, field name, method name, entity handle, analyzer name, analyzer description) to mutations	Decoupling	JM
`Faultify.MutationCollector.Mutation.IMutation`	Add `GetEquivalentMutation` method	Bug fix	JM
`Faultify.MutationCollector.Mutation.IMutation`	Use parentMethodEntityHandle for mapping coverage to mutations	Bug fix	AD
`Faultify.MutationCollector.Mutation.OpCodeMutation`	Add metadata (assembly name, type name, field name, method name, entity handle, analyzer name, analyzer description) to mutations	Decoupling	JM
`Faultify.MutationCollector.Mutation.OpCodeMutation`	Add `GetEquivalentMutation` method	Bug fix	JM
`Faultify.MutationCollector.Mutation.OpCodeMutation`	Use parentMethodEntityHandle for mapping coverage to mutations	Bug fix	AD
`Faultify.MutationCollector.Mutation.VariableMutation`	Add metadata (assembly name, type name, field name, method name, entity handle, analyzer name, analyzer description) to mutations	Decoupling	JM
`Faultify.MutationCollector.Mutation.VariableMutation`	Add `GetEquivalentMutation` method	Bug fix	JM
`Faultify.MutationCollector.MutationFactory`	Added	Decoupling & Restructuring	JM
`Faultify.MutationCollector.MutationFactory`	Use parentMethodEntityHandle for mapping coverage to mutations	Bug fix	AD
`Faultify.MutationScheduler.CoverageMapper`	Added	Restructuring	JM
`Faultify.MutationCollector.CoverageMapper`	Use parentMethodEntityHandle for mapping coverage to mutations	Bug fix	AD
`Faultify.MutationScheduler.MutationBucket`	Added	Other	JM
`Faultify.MutationSessionRunner.IMutationSessionRunner`	Added	Decoupling & Restructuring	AD
`Faultify.MutationSessionRunner.IMutationSessionRunner`	Moved SessionProgressTracker to pipeline	Decoupling	AD
`Faultify.MutationSessionRunner.MutationSessionRunner`	Added	Restructuring	AD
`Faultify.MutationSessionRunner.MutationSessionRunner`	Moved SessionProgressTracker to pipeline	Decoupling	AD
`Faultify.Pipeline.Pipeline`	Added	Restructuring	AD
`Faultify.ProjectDuplicator.IFileDuplication`	Added	Decoupling	AD

Faultify.ProjectDuplicator.ITestProjectDuplication	Added	Decoupling	AD
Faultify.ProjectDuplicator.ITestProjectDuplicator	Added	Decoupling	AD
Faultify.Report.JsonReporter	Moved to Faultify.Report.Reporter.JsonReporter	Bug-fix	CW
Faultify.Report.ReportData	Added	Other	AD
Faultify.Report.ReporterFactory	Added	Decoupling	CW
Faultify.Report.ReporterType	Added	Decoupling	CW
Faultify.TestHostRunner.Results.ResultsUtils	Added	Other	AD
Faultify.TestRunner.MutationTestProject	Removed (spread out across many places)	Restructuring & Decoupling	JM, AD
Faultify.TestRunner.ProjectDuplication.TestProjectDuplicationPool	Removed	Other	AD
Faultify.TestRunner.ProjectDuplication.TestProjectDuplication	Moved to Faultify.ProjectDuplicator.TestProjectDuplication	Restructuring	AD
Faultify.TestRunner.ProjectDuplication.TestProjectDuplicator	Moved to Faultify.ProjectDuplicator.TestProjectDuplicator	Restructuring	AD
Faultify.TestRunner.ProjectDuplication.FileDuplication	Moved to Faultify.ProjectDuplicator.FileDuplication	Restructuring	AD
Faultify.TestRunner.TestProjectReportModelBuilder	Removed (parts re-used for Report module)	Restructuring	AD
Faultify.TestRunner.Collector.CoverageDataCollector	Moved to Faultify.TestHostRunner.Collector.CoverageDataCollector	Restructuring	AD
Faultify.TestRunner.Collector.TestDataCollector	Moved to Faultify.TestHostRunner.Collector.TestDataCollector	Restructuring	AD
Faultify.TestRunner.Logging.LogMessageType	Moved to Faultify.MutationSessionProgressTracker.LogMessageType	Restructuring	AD
Faultify.TestRunner.Logging.MutationRunProgress	Moved to Faultify.MutationSessionProgressTracker.MutationRunProgress	Restructuring	AD
Faultify.TestRunner.Logging.MutationSessionProgressTracker	Moved to Faultify.MutationSessionProgressTracker.MutationSessionProgressTracker	Restructuring	AD
Faultify.TestRunner.Shared.MutationCoverage	Removed (merged into ResultUtils)	Other	AD

Faultify.TestRunner.Shared.RegisteredCoverage	Removed	Decoupling & Simplification	AD
Faultify.TestRunner.Shared.TestResults	Removed	Decoupling & Simplification	AD
Faultify.TestRunner.Shared.TestResult	Removed	Decoupling & Simplification	AD
Faultify.TestRunner.Shared.TestRunnerConstants	Moved to Faultify.TestHostRunner.Results.TestRunnerConstants	Restructuring	AD
Faultify.TestRunner.Shared.Utils	Removed (merged into ResultUtils)	Other	AD
Faultify.TestRunner.TestFramework	Removed	Decoupling & Simplification	AD
Faultify.TestRunner.TestHost	Moved to Faultify.TestHostRunner.TestHost	Restructuring	AD
Faultify.TestRunner.TestProcess.ExitCodeException	Moved to Faultify.TestHostRunner.TestProcess.ExitCodeException	Restructuring	AD
Faultify.TestRunner.TestProcess.ProcessRunner	Moved to Faultify.TestHostRunner.TestProcess.ProcessRunner	Restructuring	AD
Faultify.TestRunner.TestProjectInfo	Removed	Decoupling	AD
Faultify.TestRunner.TestRun.DefaultMutationTestRunGenerator	Moved to Faultify.MutationScheduler.DefaultMutationTestRunGenerator	Restructuring	JM
Faultify.TestRunner.TestRun.DefaultMutationTestRun	Removed	Decoupling & Simplification	JM
Faultify.TestRunner.TestRun.IMutationTestRunGenerator	Moved to Faultify.MutationScheduler.IMutationTestRunGenerator	Restructuring	JM
Faultify.TestRunner.TestRun.IMutationTestRun	Removed	Decoupling & Simplification	JM
Faultify.TestRunner.TestRun.MutationAnalyzerInfo	Removed	Decoupling & Simplification	JM
Faultify.TestRunner.TestRun.MutationVariantIdentifier	Removed	Decoupling & Simplification	JM
Faultify.TestRunner.TestRun.MutationVariant	Removed	Decoupling & Simplification	JM
Faultify.TestRunner.TestRun.TestHostRunner.DotnetTestArgumentBuilder	Moved to Faultify.TestHostRunner.TestHostRunners.DotnetTestArgumentBuilder	Restructuring	AD

`Faultify.TestRunner.TestRun.TestHostRunner.TestHostRunnerFactory`	Moved to `Faultify.TestHostRunner.TestHostRunnerFactory`	Restructuring	AD
`Faultify.TestRunner.TestRun.TestHostRunner.DotnetTestHostRunner`	Moved to `Faultify.TestHostRunner.TestHostRunners.DotnetTestHostRunner`	Restructuring	AD
`Faultify.TestRunner.TestRun.TestHostRunner.ITestHostRunner`	Moved to `Faultify.TestHostRunner.TestHostRunners.ITestHostRunner`	Restructuring	AD
`Faultify.TestRunner.TestRun.TestHostRunner.NUnitTestHostRunner`	Moved to `Faultify.TestHostRunner.TestHostRunners.NUnitTestHostRunner`	Restructuring	AD
`Faultify.TestRunner.TestRun.TestHostRunner.XUnitTestHostRunner`	Moved to `Faultify.TestHostRunner.TestHostRunners.XUnitTestHostRunner`	Restructuring	AD
`Faultify.TestRunner.TestRun.TestRunResult`	Removed (in favour of simple tuples)	Decoupling & Simplification	AD