

## Architecture Document

# FANLLIFA

Client:

RDW

Authors:

M. Bartelsman,  
C. Dadashov-Khandan,  
H. Monhemius,  
J. Faber

Teaching Assistant:

Marten Struijk

## Introduction

The product, Faultify, is a mutation-based testing tool. Its goal is to provide a quick simple way to realize mutation testing of .NET Core projects at the byte-code level. It deliberately introduces errors to the code and then runs the unit tests of the target codebase; if a unit test still succeeds, it is highly likely that the unit test is error-prone.

Faultify is an already existing project which was created for RDW by a previous student group. Given that a lot of the codebase has already been written, most design choices have already been made. Our main tasks are then to improve and expand the current proof of concept, along with tailoring the application to RDW's needs.

## Technology stack

### Languages

The codebase of Faultify is written in C#. This includes anything from making the mutations, to building the report on the mutation testing. The actual report is built up from HTML.

Faultify applies mutations on the Common Intermediate Language (CIL) that is generated when a .NET application is built. This is done for ease of operation and performance concerns.

### Frameworks

Since Faultify needs to be as versatile as possible, there are several different testing frameworks that are being used for the testing of the methods that have been mutated.

Faultify has functional interoperability with the dotNET and the NUnit test hosts; these are both accessed by the TestRunner module of Faultify. The dotNET testing framework lets Faultify run tests of any testing framework.

### Libraries

There is a heavy reliance on the Mono.Cecil library, which is used for modifying assemblies at runtime. This library is vital for compilation and also for modifying CIL opcodes in order to perform mutations. It is accessed by the Injection, Core, TestRunner, and Analyze modules.

In addition, Faultify uses external libraries to generate reports in formats specified by the User, namely WkHtmlToPdfDotNet, System.Text.Json, and RazorLight.

Besides this, the vast majority of Faultify's functionality relies only on System libraries.

How and where these libraries are used will be expanded upon in the next section.

## Architectural overview

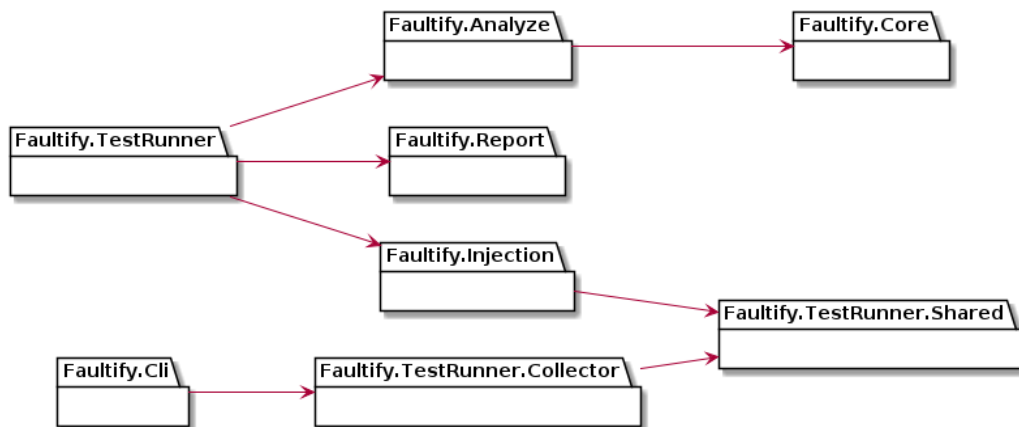
This architecture overview provides information about the general structure and the public facing members and classes of Faultify. Private members and inner class structure are not shown.

## Components

The main application is structured into 11 packages, plus a unit test and some benchmark packages for testing purposes. These packages are:

- Faultify.Analyze
- Faultify.Cli
- Faultify.Core
- Faultify.Injection
- Faultify.Report
- Faultify.TestRunner
- Faultify.TestRunner.Collector
- Faultify.TestRunner.Shared
- *Faultify.Tests*
- *Faultify.Benchmark*

External dependencies will be expanded on in each respective section. As for internal dependencies, the package dependency graph can be seen below:



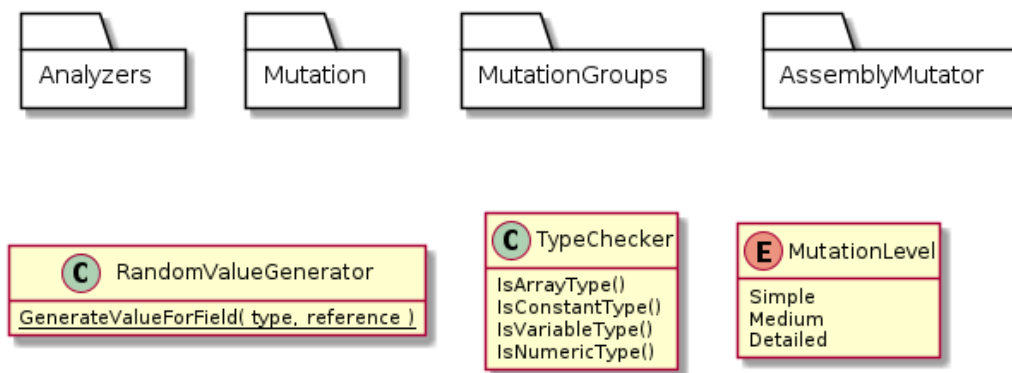
## Faultify.Analyze

External dependencies:

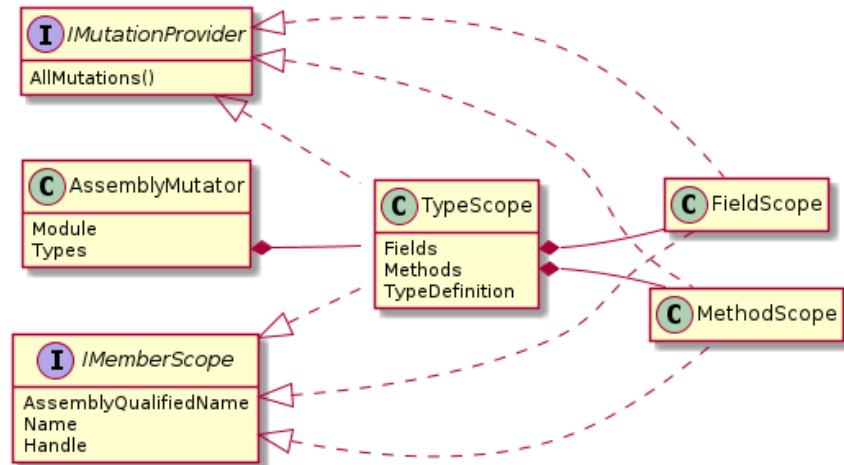
- *Mono.Cecil*
  - *Used to analyze the assembly code of the target project*
- *MonoMod.Utils*
  - *Used to create a new assembly instruction for array mutations*
- *NLog*
  - *The logging framework*

Faultify.Analyze handles the exploration of the files compiled into CIL, exploring the possible mutations and making them available.

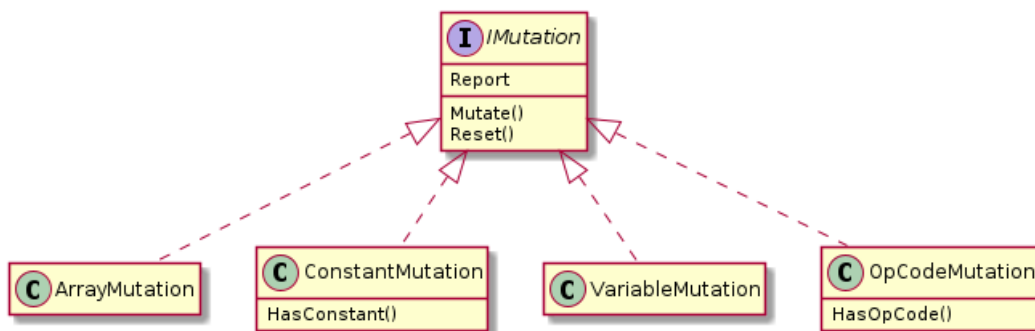
The package itself contains a few utility classes, functional classes are classified into 4 subfolders. The top-level structure is seen below:



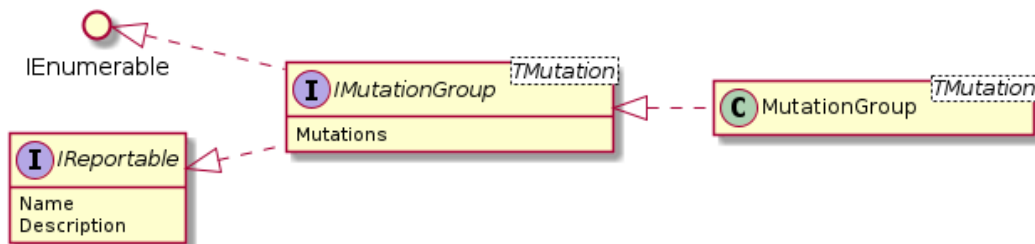
## AssemblyMutator



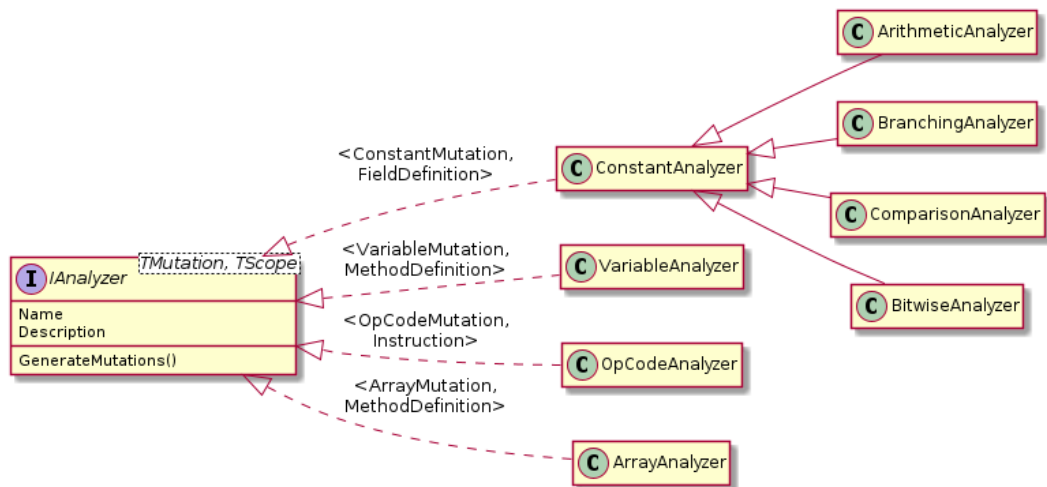
## Mutation



## MutationGroups



## Analizers

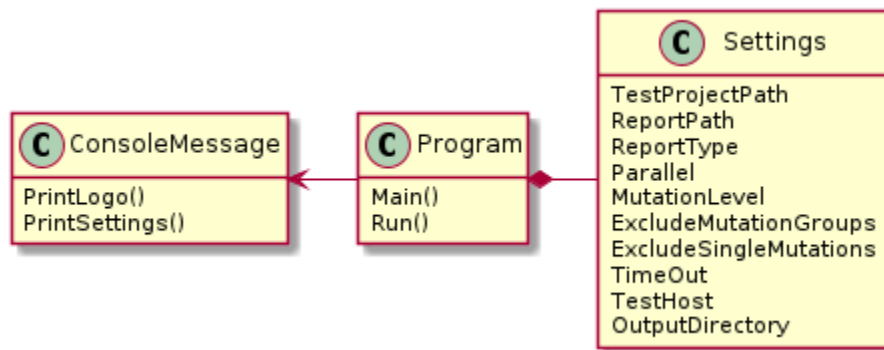


## Faultify.Cli

External dependencies:

- *CommandLineParser*
  - *Parsing CLI arguments into the Settings class.*
- *Microsoft.Extensions.DependencyInjection*
  - *Used to run Faultify with Dependency Injection*
- *Microsoft.Extensions.Configuration*
  - *Used as part of the Dependency Injection setup*
- *Microsoft.Extensions.Logging*
  - *Used as part of the Dependency Injection setup*
- *NLog*
  - *The logging framework*

Faultify.Cli is the entry point for Faultify, it also handles program configuration via the Settings class and a portion of the console messages printed during each run.



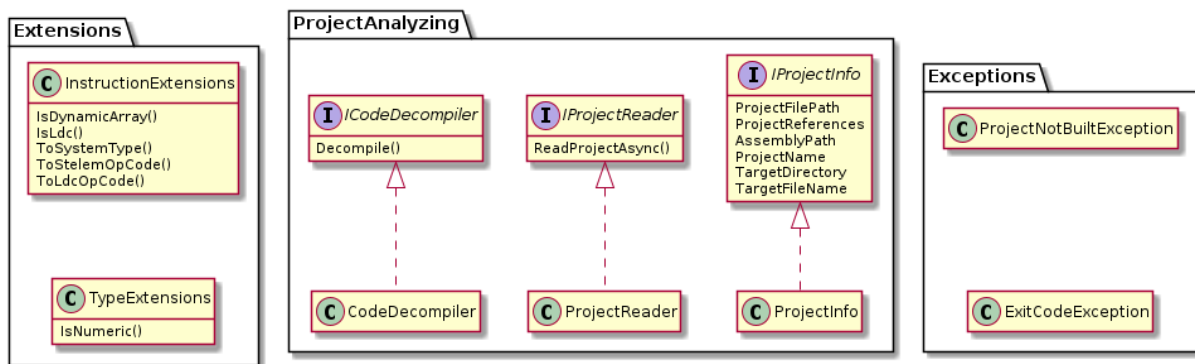


## Faultify.Core

External dependencies:

- Buildalyzer
  - Building the target project
- ICSharpCode.Decompiler
  - Decompiling the target codebase
- Mono.Cecil
  - Storing project information in “definition” classes

Faultify.Core takes care of compilation and decompilation functionality across the project, as well as hosting various extension methods.

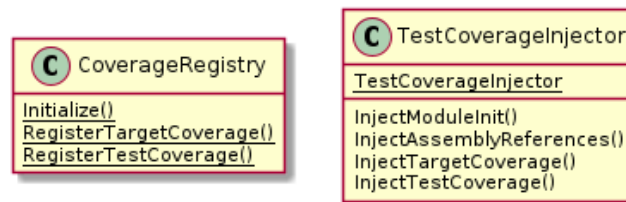


## Faultify.Injection

External dependencies:

- Mono.Cecil
  - Used for injecting code into the target codebase and resolving references
- NLog
  - The logging framework

This package manages the injection of foreign code into the assembly files that are being mutated, for purposes of registering test coverage.

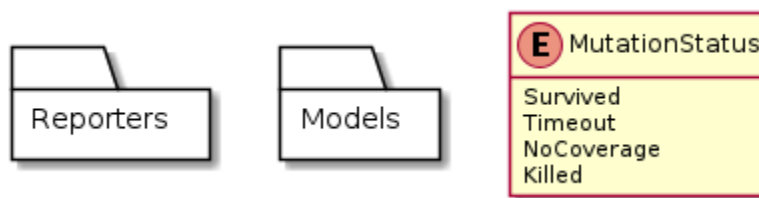


## Faultify.Report

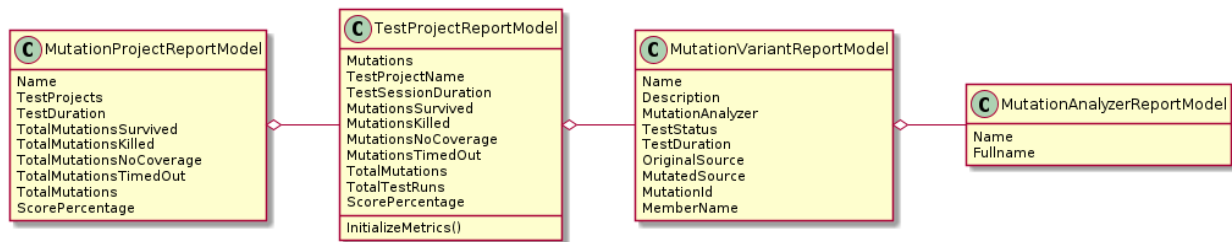
External dependencies:

- RazorLight
  - Generating HTML reports
- WkHtmlToPdfDotNet
  - Converting an HTML report to PDF
- NLog
  - The logging framework

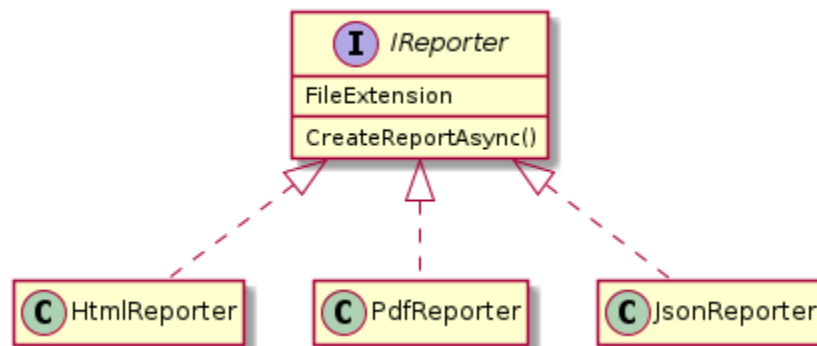
This package handles report building and rendering after each run of Faultify.



## Models



## Reporters

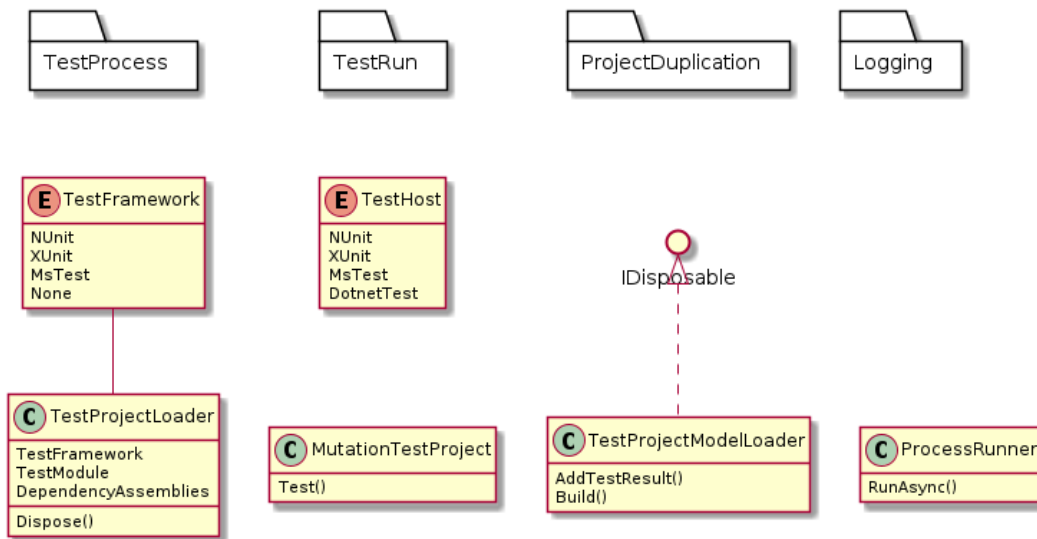


## Faultify.TestRunner

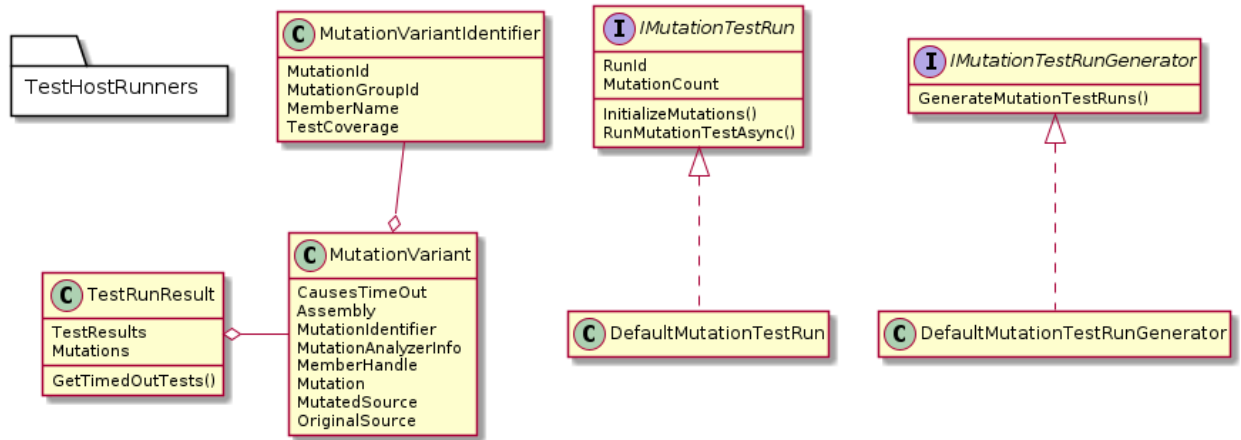
External dependencies:

- *MemoryTest*
  - *External Faultify package for running dedicated NUnit and XUnit test runner implementations*
- *Microsoft.NET.Test.Sdk*
  - *Indirectly required to run some test-related processes*
- *Mono.Cecil*
  - *Used to access target project assembly structure and definitions*
- *Newtonsoft.Json*
  - *Used to read configuration settings for mutation exclusion*
- *NLog*
  - *The logging framework*

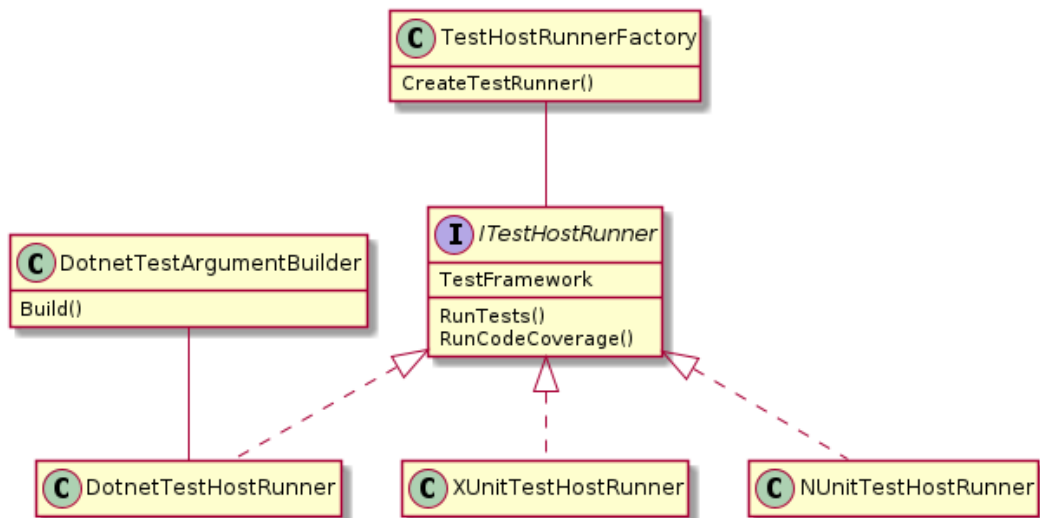
The TestRunner package does the majority of the heavy lifting in Faultify. It manages the project assembly files, the mutation process, the coverage process, the test runs, the assemblage of the report data and printing progress messages to the command line.



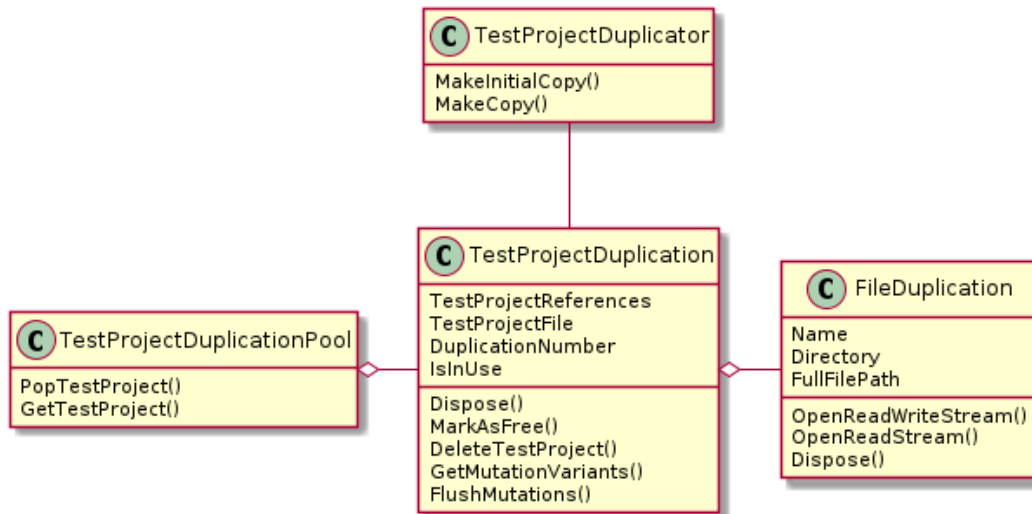
## TestRun



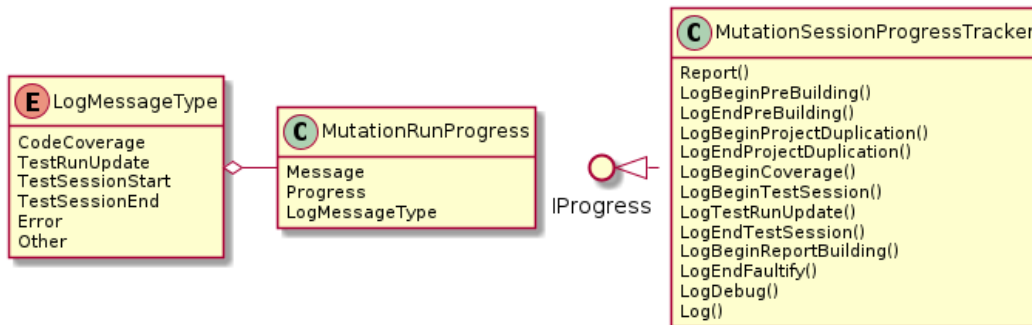
## TestHostRunners



## ProjectDuplication



## Logging

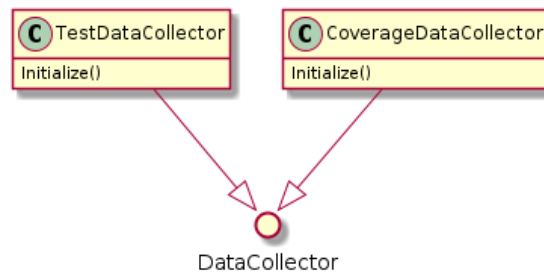


## Faultify.TestRunner.Collector

External dependencies:

- *Microsoft.TestPlatform.ObjectModel*
  - *Framework used for retrieving and collecting test diagnostics*

This package reads the files generated by test runs and coverage analyses and generates objects containing the file data for later use in the program.

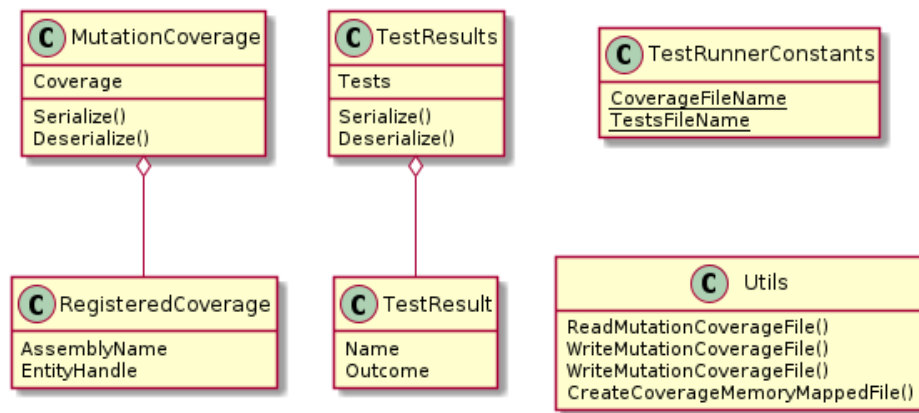


## Faultify.TestRunner.Shared

External dependencies:

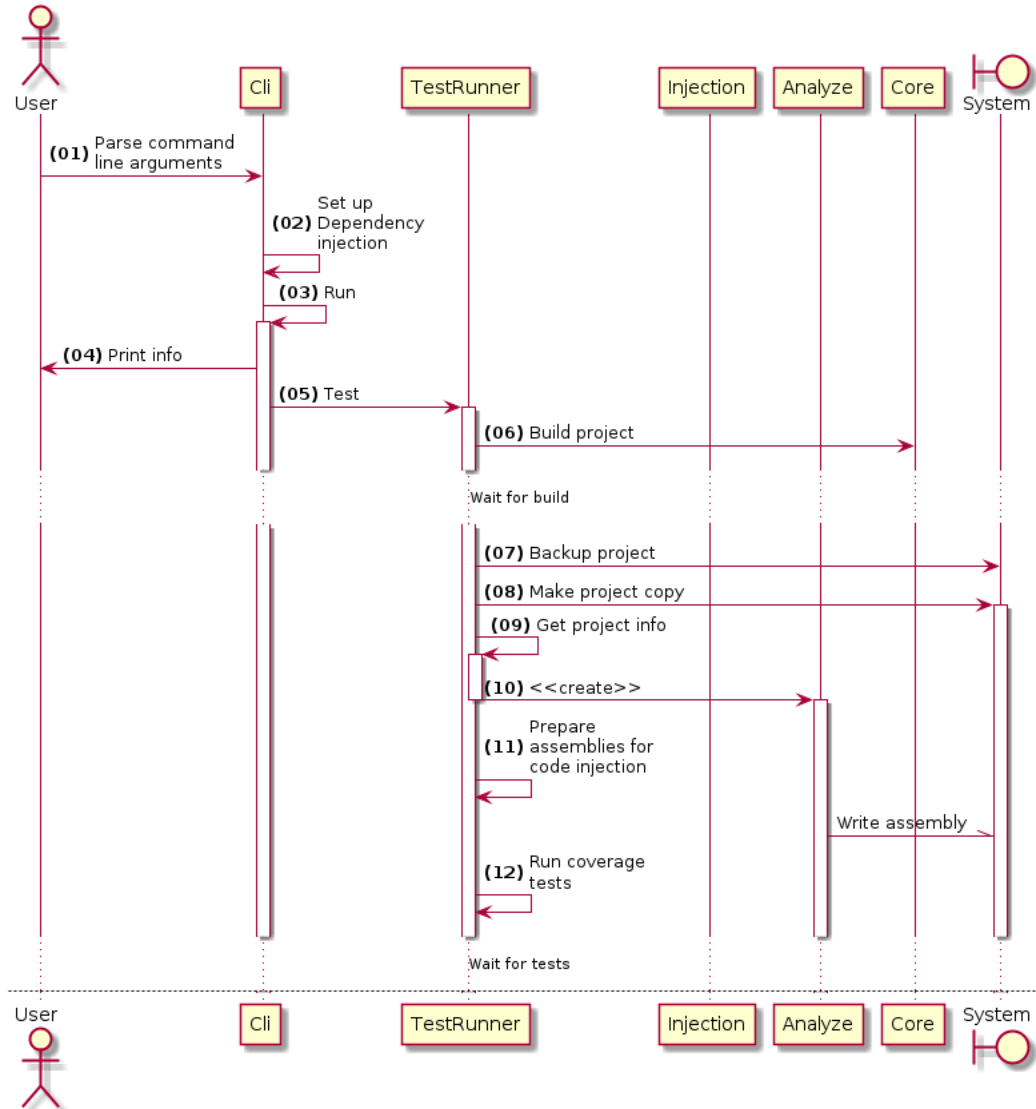
- *Microsoft.TestPlatform.ObjectModel*
  - *Framework used for retrieving and collecting test diagnostics*
- *NLog*
  - *The logging framework*

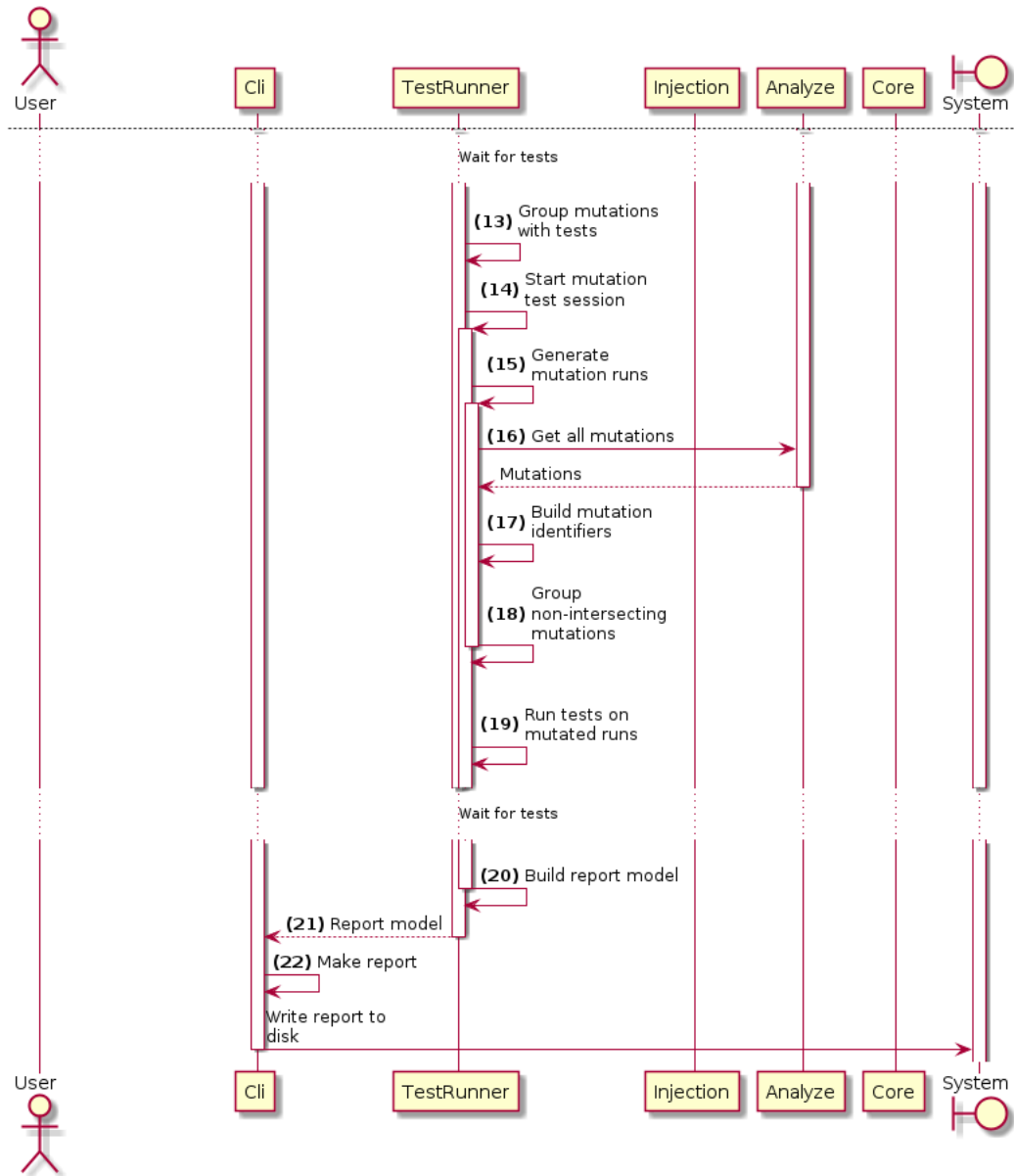
Provides common functionality for the various TestRunner packages. This includes result collection classes, coverage collection classes, constants, and I/O utility functions.





## Sequence of events





## 01. Parse command-line arguments

Faultify makes use of the CommandLine package to provide command-line arguments parsing. Arguments are parsed into a Settings object that is used to define core configuration options for running Faultify. Sensible defaults are provided where needed.

## 02. Set-up dependency injection

Faultify was designed to work with .NET's dependency injection (DI) facilities. The Settings object is bound to the DI framework and the main process is started via DI as a singleton service.

## 03. Run

This task handles a high-level overview of the program's execution. It runs the mutation testing suite and then generates a report based on the results of these tests.

## 04. Print information

Before the beginning of the testing process, this handles the printing of the program's welcome screen, as well as the general program settings, to confirm the options that have been provided by the user.

## 05. Test

Begins the suite of tasks needed to complete a single mutation test run. The general overview of the process is:

- Build the target project
- Backup the built assemblies
- Run a mutation coverage analysis
- Run mutations and unit tests on the mutated code

## 06. Build Project

Faultify uses a build framework called Buildalyzer in order to build the target project.

## 07. Backup project assemblies

For mutation testing to work, the program needs to be modified at some level (source code, byte code, machine code). Because this is a destructive process, it is first necessary to back up the original assemblies and instead work on a copy of these.

In the case of Faultify, it generates a TestDuplicator object, which contains a reference to the assemblies and interfaces with the filesystem to provide copies of the assemblies for mutation as needed.

## 08. Make initial copy

Produce a first copy of the project assemblies. This first copy will be the target for the coverage analysis. This process also stores all the information needed to make further copies without reading the original build multiple times.

## 09. Get project information

Generate a TestProjectInfo object. This object has references to the various modules present on the built project and provides streaming and filesystem I/O facilities.

## 10. Create Assembly Mutators

In order to have access to these facilities, a new AssemblyMutator object must be created for each assembly generated during the build process.

## 11. Prepare assemblies for code injection

Callbacks are injected into the target project assemblies. The purpose of these callbacks is to create a registry relating unit tests and methods in the target

project. This is done in order to minimize the number of test runs necessary to test all the generated mutations.

## 12. Run coverage tests

A test host instance is created and executed on the modified assemblies in order to populate the coverage registry

## 13. Group mutations with tests

Maps mutations to the tests that cover them

## 14. Start mutation test session

Handles the tests runs and output for the mutated codebase

## 15. Generate mutation runs

Assembles the various mutation groups that will be tested together

## 16. Get all mutations

Retrieves all generated mutations

## 17. Build mutation identifiers

Generates identifier objects for each mutation, these are used to keep track of every unique mutation

## 18. Group non-intersecting mutations

An algorithm is executed to make mutation groups such that the tests that cover them do not overlap and the number of groups is minimal. This is a form of the Set Cover problem, and the algorithm that is selected depends on the number of mutations in question.

## 18. Run tests on mutated code

Creates a new test host and executes the unit tests on the mutated code

## 19. Build report model

Using the results from the test runs, assembles a report model that can be used to produce a report by one of the IReporter classes

## 20. Make report

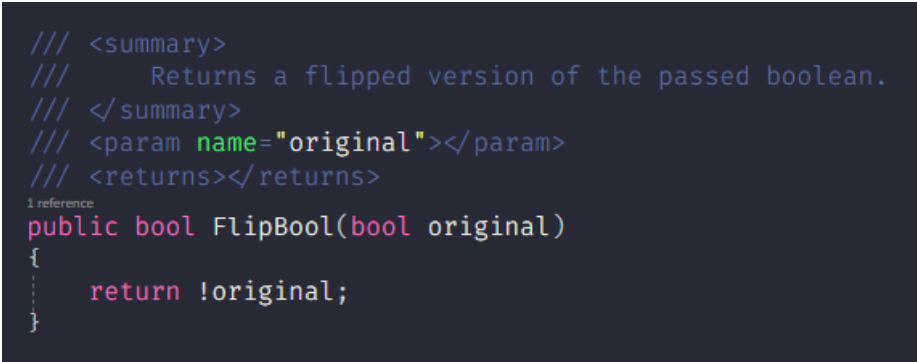
Build a report in the format that was requested

## Issues

Early on in the project, the Faultify codebase was determined to be problematic, with numerous design flaws, code smells and readability issues. While many of these issues have been addressed, we believe that Faultify, as it currently stands, is unscalable.

### Initially found and resolved issues

#### Code readability and bad practices



```
/// <summary>
///     Returns a flipped version of the passed boolean.
/// </summary>
/// <param name="original"></param>
/// <returns></returns>
1 reference
public bool FlipBool(bool original)
{
    return !original;
}
```

Use of shorthand notation, naming convention violations, confusing code organization and structure, empty classes, multiple classes per file, poor package organization, and various other issues made the code hard to understand, read and navigate, which negatively impacted our ability to work on the codebase.

#### Improper documentation

There were numerous cases where documentation was either missing, outdated, or copied over from other methods without being properly updated.

## Current issues

### Coupling and redundancy

One of the biggest problems that were faced during this development process was that high coupling between classes and modules made it extremely costly to introduce new features or modify existing ones. Small modifications would have repercussions throughout the whole project, resulting in weeks of debugging and troubleshooting after every minor change.

Time was set aside to refactor the project as best as possible but ultimately the time that could be allocated for this task was insufficient to significantly fix the underlying problems.

### Mutation strategies

The choice of mutations when analyzing code seems to be, generally speaking, arbitrary and unrefined. Certain mutation targets are mutated multiple times, proving redundant. Other targets are not mutated at all. In some cases, Faultify goes to great lengths to modify more complex code structures, while completely disregarding others.

The nature of the problem, however, is not merely one of choice, but one of architectural decisions. Faultify, with its current structure, is not scalable or extendable in terms of mutations, without making the codebase more fragile than it already is.

### Fragile implementations

Some of the core functionality of the code is structured in ways that are dependent on perfect conditions and inputs, which results in flaky handling of edge cases and code that is hard to extend. Some of these issues were fixed throughout our involvement in the project, but there are a few instances where these issues persist, mostly at the direct byte-code adjustment classes.



## Testing

Faultify's functionality has a complex nature that cannot be boiled down to a simple "Input -> Output" format that unit tests heavily rely on. For instance, to test Faultify's ability to run another codebase's tests, one must supply Faultify with a target test file and also use a lot of Faultify's packages in bulk without individually evaluating them.

As a result, Faultify is impossible to fully test without creating bloated dummy set-up classes or using mocking frameworks. Both of these options were too resource and time expensive for the scope of the project. Therefore no new unit tests were written for our implemented requirements. Only the existing unit tests were updated and corrected, as they already provided good coverage of all code that is testable.

## Known bugs and problems

- Faultify does not properly report timeouts when running tests with the dotnet framework. As a result, in detailed log files, the timeouts are indistinguishable from crashes.
- Due to issues with the NUnit and XUnit libraries, running test cases with the dedicated NUnit and XUnit libraries prevents the test duplications from being deleted at the end of the test run.
- Inability to resolve dependencies in certain projects. We have not been able to reproduce this bug, as such, the causes of this issue are still unknown.
- If Faultify is unable to find any unit tests in the target project, it will run without issues but will report 0 mutations and a negative time estimate.
- If a test project needs to modify two or more assemblies that have a dependency relation, one of those assemblies might be locked as read-only by the system, as it is being executed.
- The PDF report takes an inordinate amount of time to be rendered
- Running a test with the dedicated NUnit runner will have Faultify fail to output final details of its test runs, making it look like it terminated halfway through despite running successfully.
- On large codebases, Faultify might be unable to resolve test cases due to shared dependency assemblies being locked by the filesystem

- In some cases, the generated report will show the correct code pre-mutation but show completely unrelated code post-mutation. This is possibly due to the fact that the mutated method is kept track of with an EntityHandle, which is an unreliable identification.

## Recommendations for future development

Faultify in its current state is functional, however it is not maintainable and is not scalable. There are many fundamental problems that remain in core parts of the codebase. We recommend that future teams first rewrite core parts of Faultify before expanding on its functionality.

In addition, we recommend using [mocking frameworks](#) to better unit-test Faultify, to address the issues mentioned in the [Testing](#) section.

## Team organisation and Project management

Due to Faultify's interconnected nature, it was hard to divide the team into permanent roles. Throughout the project, we divided work based on requirements, and often worked in pairs with larger tasks. We followed standard team procedures otherwise, with weekly or bi-weekly pull requests into the main branch approved by up to two other group partners.

Early on in the project, a developer from the previous student team had an advisory role in the project, but communications ceased several weeks later.

The project was managed using Github. During the development of Faultify, we used a branch naming convention. It consists of an initial tag (feat, bug, cln), then the number of the sprint and a quick label describing what was being worked on. An example branch of this condition would be **feat-05-MSTestSupport**.

## Change log

Who	What	When
Miguel Bartelsman, Chingiz Dadashov-Khandan, Hessel Monhemius, Joran Faber	Creation of the document	30/03/21
Miguel Bartelsman, Chingiz Dadashov-Khandan, Hessel Monhemius,	Update document per feedback	23/04/21
Miguel Bartelsman	Wrote the Architectural Overview Components section	26/04/21
Miguel Bartelsman	Added class diagrams for Faultify.Analyze	28/04/21
Miguel Bartelsman	Added class diagrams for Cli, Core, Injection, and Report	04/05/21
Miguel Bartelsman	Added class diagrams for TestRunner, Collector, and Shared. Updated diagram for Core	21/05/21
Miguel Bartelsman	Began writing the Sequence of Events section	29/05/21
Chingiz Dadashov-Khandan, Joran Faber	Performed some updates to the document and added the issues session	30/05/21
Chingiz Dadashov-Khandan, Joran Faber, Miguel Bartelsman	Update diagrams and general polishing. Adding the Issues section	31/05/21

Chingiz Dadashov-Khandan, Joran Faber, Miguel Bartelsman, Hessel Monhemius	Finalize the architecture document	12/6/21
---	---------------------------------------	---------