INF2102

Informatics Department

PUC-Rio

Programming Conclusion Project

Author:
Rodrigo Laigner

Instructor:
Dr. Marcos Kalinowski

March 29, 2021

Pontifícia Universidade Católica
do Rio de Janeiro

Contents

List of Figures

List of Tables

1 Introduction

Dependency Injection (DI) is a programming mechanism that "builds on the decoupling given by isolating components behind interfaces, and focuses on delegating the responsibility for component [or module] creation and binding to a DI container" [1]. According to Laigner et al. [2], "DI anti-pattern [is] a recurring DI usage pattern in source code that degrades aspects that DI is supposed to improve, such as coupling, or other quality aspects, such as performance."

Based on a proposed catalog of DI anti-patterns [2], this work aims at describing the development of a software system called DIAnalyzer that automatically identifies each anti-pattern proposed. A set of diagrams are presented, such as Class and Use Case, the objectives are stated, altogether with the system architecture and libraries used.

The remainder of this document is organized as follows. In Section 2 we provide the specification of DIAnalyzer. In Section 3, we describe the project, architecture, and diagrams. Next, test scripts are introduced in Section 4. Lastly, an evaluation of the occurrence of the cataloged DI anti-patterns by DIAnalyzer is presented on Section 5.

2 Specification

This section presents the specification of the system developed in this work. Thus, the goals, requirements and use cases are provided.

2.1 Goals

The main goal of the system is to automatically identify each proposed DI anti-pattern [2] on source code. A critical success factor is the ability to support different software projects. Thus, this work aims at providing support for the identification of anti-patterns in projects with different frameworks, such as Spring[1] and Guice[2].

Next, in order to measure the critical success factor, the number of software projects supported must be sufficiently high. Finally, as a risk, the

[1] https://spring.io

[2] https://github.com/google/guice

presence of bugs undermine the usage of the tool. It is necessary to avoid false positives and false negatives in the system output.

2.2 Requirements

This subsection presents the functional and non-functional requirements of the project.

2.2.1 Functional Requirements

- The user can provide the project to be analyzed
- The user can provide the output path of the analysis results provided by the tool
- The tool must provide as output an spreadsheet with the following information: For each DI anti-pattern detected, the class, element, and DI anti-pattern must be shown in each line
- The first line of the output spreadsheet provided by the tool must contain the header, with the column names Class, Element, and DI Anti-Pattern, respectively
- The user must be able to download open source projects from GitHub given an query provided as input

2.2.2 Non Functional Requirements

- An analysis cannot take more than 10 minutes

Portability

- The system must be able to execute on Windows, MacOS and Linux distributions

Usability

- The user must be able to execute an analysis in less than 1 minute

2.3 Use Cases

This subsection depicts the use cases diagram and describes each use case in the program.

2.3.1 Use Cases Diagram

The use cases diagram is shown in Figure 1. The use interacts with two use cases, Execute analysis and Submit GitHub query.

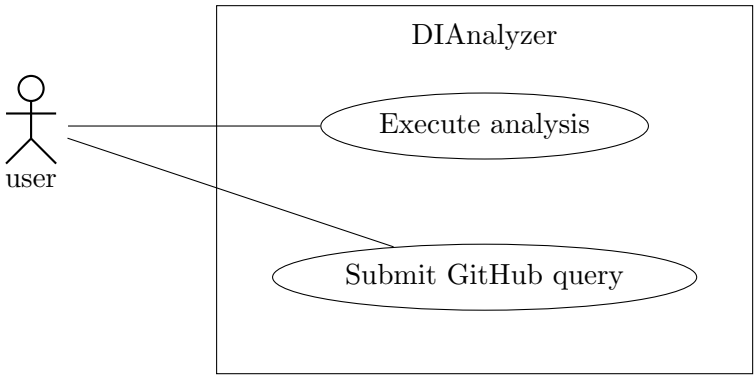


Figure 1: Use Cases Diagram

2.3.2 Use Cases Description

This sub-subsection further describes each use case found in Figure 1.

UC1 - Execute analysis

Actor: User

Overview: The user opens up the command line tool of its operational system and executes the following command: java -jar dianalyzer.jar analysis <project path> <output path>. The user must ensure that a Java Virtual Machine is properly installed on operational system. After the submission of the command, the user must wait for the program termination. The report of the analysis will be available at the specified output path provided by the user as a parameter.

Trigger: There is no trigger

Pre-condition: The project under analysis and the output path must exist in user's file system and must be properly informed as a parameter

Post-condition: The report is available to the user.

Typical sequence of events

Actor Action	System's response
1. The user execute DIAnalyzer in command line tool	
	2. The program parses the parameters received, checking for correctness
	3. The program starts the analysis of the intended software project
	4. The program finishes its execution with the output of the analysis report

Table 1: UC1 Typical Sequence of Events

Alternative sequences

2.a: The programs outputs an error to the user, indicating that a parameter submitted is wrong

UC2 - Submit GitHub query

Actor: User

Overview: The user opens up the command line tool of its operational system and executes the following command: java -jar dianalyzer.jar mining <output path> <query string 1> <query string 2> ... <query string n>. The user must ensure that a Java Virtual Machine is properly installed on operational system. After the submission of the command, the user must wait for the program termination. The projects found by the search on GitHub repository will be available at the specified output path provided by the user as a parameter.

Trigger: There is no trigger

Pre-condition: The output path must exist in user's file system and must be properly informed as a parameter

Post-condition: The projects found are available to the user

Typical sequence of events

Alternative sequences

2.a: The programs outputs an error to the user, indicating that a parameter submitted is wrong

Actor Action	System's response
1. The user execute DIAnalyzer in command line tool	
	2. The program parses the parameters received, checking for correctness
	3. The program starts the search based on the query strings provided
	4. The program finishes its execution, making the projects that were found by the search available at the specified path

Table 2: UC2 Typical Sequence of Events

3 Project

3.1 Architecture

The system architecture is decomposed in a set of subsystems. The subsystems are: repository extractor, data model extractor, analysis, and report. The description of each subsystem is given as follows.

3.1.1 Repository Extractor Subsystem

The Repository Extractor subsystem is responsible for submitting a request (in form of a query) to GitHub API in order to clone a copy of a open source project into user's file system.

3.1.2 Data Model Extractor Subsystem

To support the identification of the DI anti-patterns, the JavaParser framework [3] was used. The Data Model Extractor subsystem converts each file of the project under analysis into a model that can be manipulated by the system. The Data Model Extractor Subsystem is built as a layer above the JavaParser framework, abstracting its internals in order to ease reuse and diminish coupling to JavaParser from other subsystems of the architecture.

[3]https://github.com/javaparser

JavaParser relies on constructing an AST for a given compilation unit (i.e. Java class) and represents object oriented elements, such as methods, attributes, and classes in form of a vertex in a tree.

3.1.3 Analysis Subsystem

The Analysis subsystem is the core part of the architecture, since it contains the logic that verifies if an anti-pattern is applied in the context of a class. Basically, each anti-pattern is modeled as a class in the Analysis subsystem. An anti-pattern class is composed by a set of rules. Each rule is also modeled as a class, being responsible for identification of injected elements that obey the characteristics of given rule. Some rules also have as dependence a data source, which are classes that provided additional class information on run time. In addition, an anti-pattern class can also depend on business classes. Business classes provide additional features, such as identification of injected elements. A comprehensive example is given in subsection 3.2.

3.1.4 Report Subsystem

The Report subsystem is responsible for handling requests related to convert the results of the Analysis Subsystem into a report.

3.2 Diagrams

This subsection provided diagrams referring to the system developed for this work. As the system works in main-memory, in other words, no databases are necessary to accomplish its goal, no ER diagram is provided in this work.

Figure 2 depicts the system decomposition. The first element, GitHubRepository represents the source of open-source projects extracted by means of **UC2** described in last section. Below, the four subsystems of the architecture are represented. For last, A spreadsheet represents the output the Report Subsystem provides.

Figure 3 shows the Class Diagram of DIAnalyzer, depicting an excerpt of the system. There are four classes (*BadPracticeTwo*, *InjectionBusiness*, *ReferenceOnConcreteClass*, and *HashMapDataSource*) and one interface (*IDataSource*) depicted.

BadPracticeTwo represents the anti-pattern two (Concrete class injection [2]). As mentioned in subsection 3.1, an anti-pattern class refers to a set of

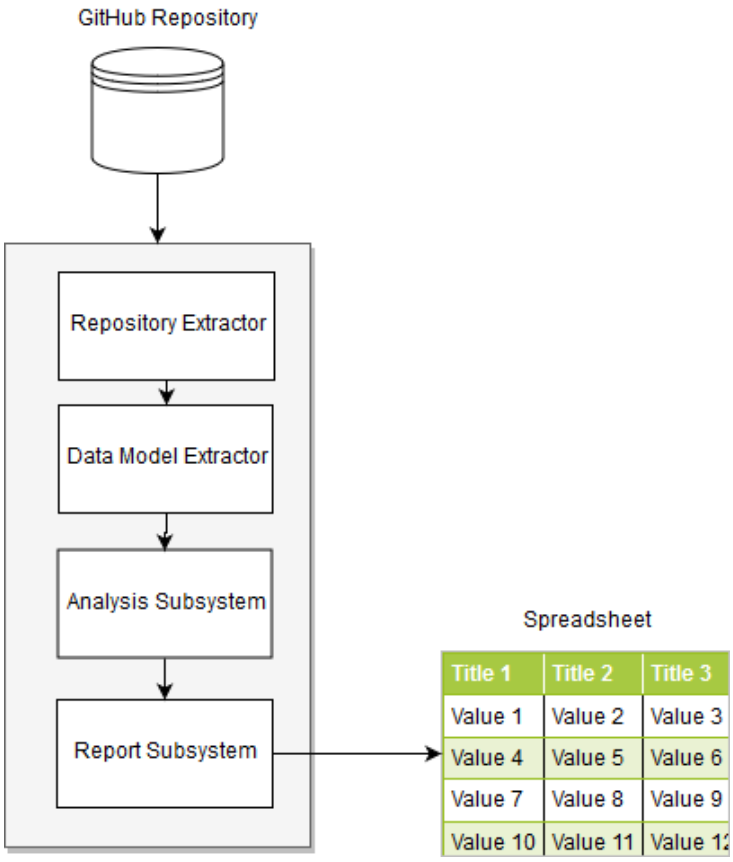


Figure 2: Schematic Overview of DIAnalyzer

rule classes in order to define if the anti-pattern is applied. In this context, *BadPracticeTwo* rely on *InjectionBusiness* in order to obtain all attributes that will receive injected instances as values in construction time. For each such attribute, *BadPracticeTwo* requests *ReferenceOnConcreteClass* to discover if the given injected element is a concrete class injection. *ReferenceOnConcreteClass*, in turn, relies on the interface *IDataSource* in order to hold an instance of *HashMapDataSource*. *HashMapDataSource* aims at retrieving additional information regarding a searched class, such as the type of the class (interface or not).

The design of the project followed an orientation to abstraction, as suggested by Martin [3] in his work on Dependency Inversion Principle (check

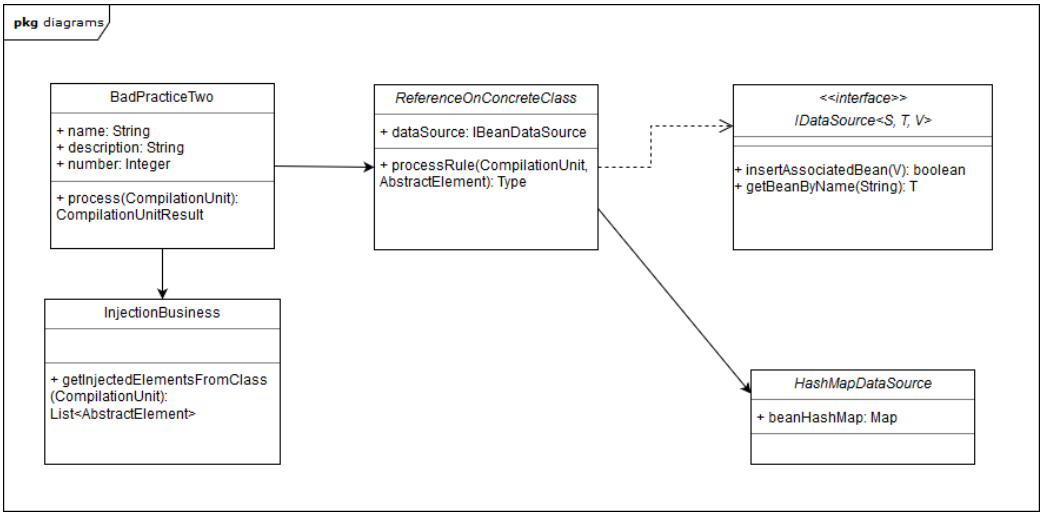


Figure 3: DIAnalyzer Class Diagram

Figure 4 (extracted from [3]) for visual representation of the principle). This way, although concrete classes are depicted in Figure 3, they are detailed implementation of abstract classes the system provides.

Thus, the system contains a set of abstract classes and interfaces in order to enable DIP. Table X presents each abstraction and its goal.

4 Test

In order to mitigate the threat regarding the presence of bugs in source code, unit tests were developed. The framework, and tests criteria employed are addressed in this section.

4.1 Framework

In order to take advantage of Java platform, a test framework was employed in this work. JUnit [4] is a test framework for Java applications that support the development of test cases in a seamless way, such as providing primitives for stating pre and post conditions on each test case developed.

[4]https://junit.org

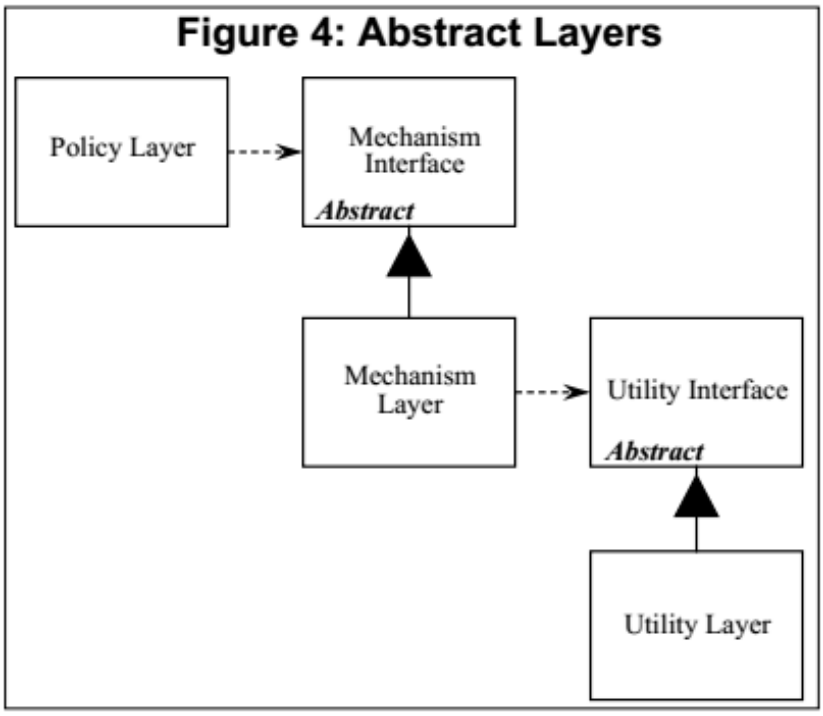


Figure 4: Dependency Inversion Principle

4.2 Implementation

An abstract test class was developed in order to enable reuse of common behavior for each test class. For each anti-pattern, a test class was created. Figure 5 shows the abstract test class implemented for this work. The annotation *@RunWith* before class definition defines that the given class is a class with test methods. The annotation *@Test* on a method defines that a unit test must be performed according to the body of the method. Next, annotation *@Parameters* defines that parameters (class files to be analyzed) must be passed to this method by instances of inherited concrete classes.

In addition, a convention over configuration implementation was adopted. It means that concrete test classes must only define the anti-pattern class that must be tested. Figure 6 depicts a concrete class for anti-pattern *Concrete-ClassInjection*. It is possible to observe how straightforward is to implement a test unit for an anti-pattern, once it is only necessary to define the given class in the annotation *BadPracticeApplied*. For last, in order to accomplish

Abstraction	Goal
AbstractPractice	Every class that represents a given anti-pattern inherits from this class. The objective with this abstraction is to provide abstraction at the report subsystem, once the report subsystem do not need to acknowledge implementation details of each anti-pattern
AbstractElement	Representes an element of code related to dependency injection in source code. As abstraction on each anti-pattern is enforced, in order to enable same abstraction level on the use of rules in different anti-patterns, an AbstractElement provides the necessary abstraction to avoid implementation details on the presence of one rule in more than one anti-pattern, for example
IDataSource	Abstracts a data source. The current implementation of data source is in memory, through a HashMap. a possible extension is to include a data source as an in memory database.

Table 3: Abstractions implemented

convention over configuration, the classes to be parsed and tested must be placed in expected directory.

4.3 Test Criteria

For each DI anti-pattern, an unit test concerning the confirmation of the existence of its respective instance given a class that contain elements that characterize the anti-pattern was designed. In other words, the tests aimed at discovering false negatives and false positives.

These tests were designed throughout the development process of the tool due to intensive changing in rules for detecting elements of code that could yield instances of anti-patterns. This way, the presence of tests supported the process of avoiding the introduction of bugs in the program.

4.4 Automatized Test Cases

As mentioned, no human intervention is needed in order to test application logic. Figure 7 depicts a test being executed through the IDE IntelliJ IDEA

[5] and its generated logs.

5 Evaluation

A summary of the evaluation conducted by Laigner et al. [2] is presented in this section.

5.1 Recall

As there is no available dataset regarding the presence of DI anti-patterns, Laigner et al. [2] developed an oracle dataset containing instances of DI anti-patterns.

Then, a relative recall analysis in the manually generated oracle showed that DI Analyzer was able to retrieve 130 out of the 141 instances, including instances of all eight DI anti-patterns contained in the oracle, resulting in a relative recall of 92.19%.

5.2 Precision

In addition, for precision analysis, a random scope of classes were selected. The results shown that DIAnalyzer detected 835 instances of DI anti-patterns, with precision between 80 to 100% for AP1, AP2, AP4, AP5, AP6, AP8, AP9, AP10, AP11, and AP12.

We consider these precision results to be sufficient for our purpose of evaluating the occurrence of the DI anti-patterns in Java projects.

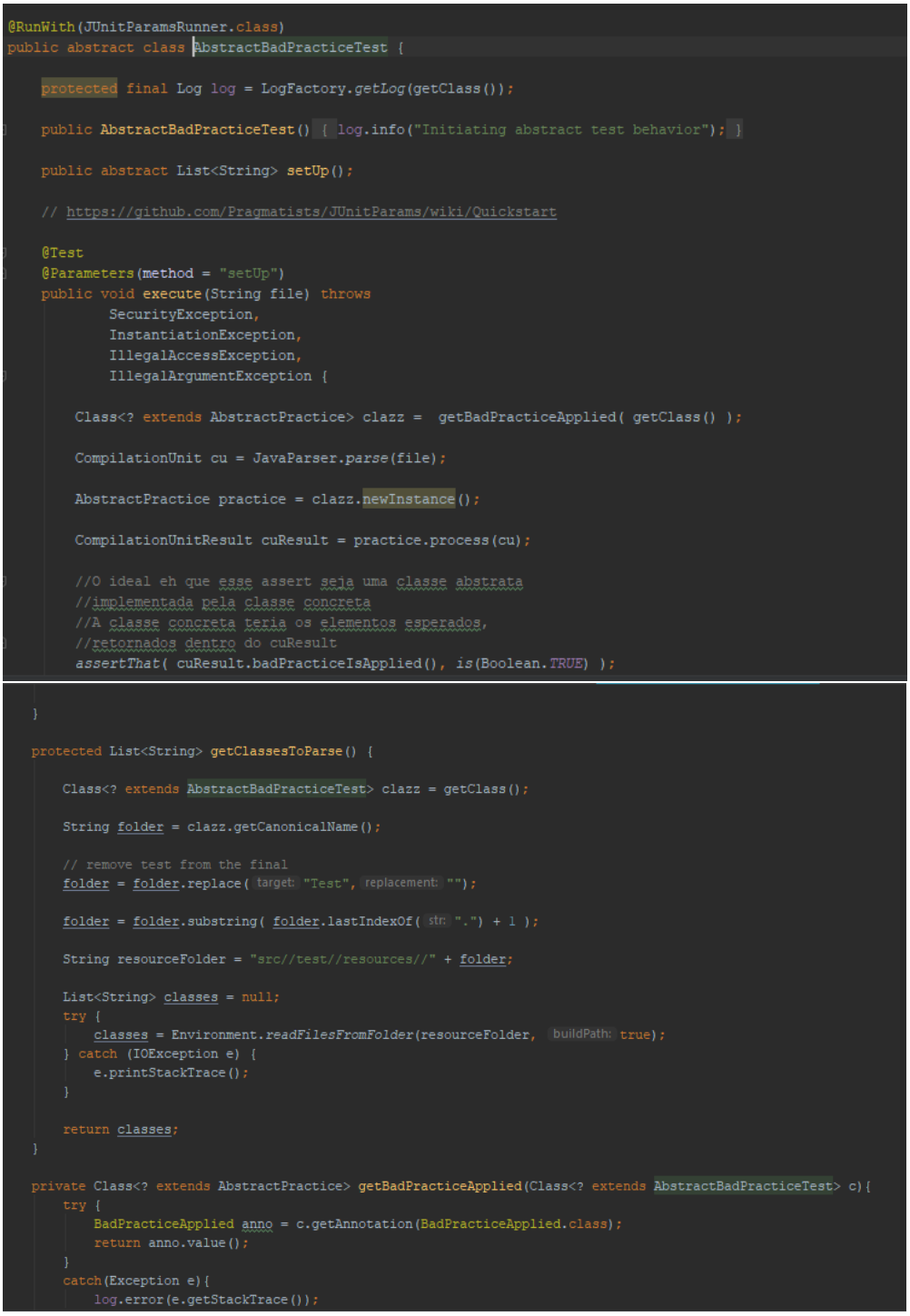
References

[1] M. Crasso, C. Mateos, A. Zunino, and M. Campo. Empirically assessing the impact of di on the development of web service applications. *Journal of Web Engineering*, 9:66–94, 2010.

[2] Rodrigo Laigner, Marcos Kalinowski, Luiz Carvalho, Diogo Mendonça, and Alessandro Garcia. Towards a catalog of java dependency injection anti-patterns. In *SBES 2019 - Research Track*, Salvador, BA, sep 2019.

[5] https://www.jetbrains.com/idea/

[3] R. C. Martin. The dependency inversion principle. *Report.*, 8(6):61–66, 1996.



```java
@RunWith(JUnitParamsRunner.class)
public abstract class AbstractBadPracticeTest {

    protected final Log log = LogFactory.getLog(getClass());

    public AbstractBadPracticeTest() { log.info("Initiating abstract test behavior"); }

    public abstract List<String> setUp();

    // https://github.com/Pragmatists/JUnitParams/wiki/Quickstart

    @Test
    @Parameters(method = "setUp")
    public void execute(String file) throws
            SecurityException,
            InstantiationException,
            IllegalAccessException,
            IllegalArgumentException {

        Class<? extends AbstractPractice> clazz =  getBadPracticeApplied( getClass() );

        CompilationUnit cu = JavaParser.parse(file);

        AbstractPractice practice = clazz.newInstance();

        CompilationUnitResult cuResult = practice.process(cu);

        //O ideal eh que esse assert seja uma classe abstrata
        //implementada pela classe concreta
        //A classe concreta teria os elementos esperados,
        //retornados dentro do cuResult
        assertThat( cuResult.badPracticeIsApplied(), is(Boolean.TRUE) );
    }

    protected List<String> getClassesToParse() {

        Class<? extends AbstractBadPracticeTest> clazz = getClass();

        String folder = clazz.getCanonicalName();

        // remove test from the final
        folder = folder.replace( target: "Test", replacement: "");

        folder = folder.substring( folder.lastIndexOf( str: ".") + 1 );

        String resourceFolder = "src//test//resources//" + folder;

        List<String> classes = null;
        try {
            classes = Environment.readFilesFromFolder(resourceFolder,  buildPath: true);
        } catch (IOException e) {
            e.printStackTrace();
        }

        return classes;
    }

    private Class<? extends AbstractPractice> getBadPracticeApplied(Class<? extends AbstractBadPracticeTest> c){
        try {
            BadPracticeApplied anno = c.getAnnotation(BadPracticeApplied.class);
            return anno.value();
        }
        catch(Exception e){
            log.error(e.getStackTrace());
```

Figure 5: Abstract Class Test

```java
package br.pucrio.inf.les.ese.dianalyzer.diast_test.practices;

import br.pucrio.inf.les.ese.dianalyzer.diast.practices.BadPracticeTwo;
import br.pucrio.inf.les.ese.dianalyzer.diast_test.annotation.BadPracticeApplied;

import java.util.List;

@BadPracticeApplied(BadPracticeTwo.class)
public class BadPracticeTwoTest extends AbstractBadPracticeTest {

    @Override
    public List<String> setUp() { return super.getClassesToParse(); }

}
```

Figure 6: Concrete Class Test

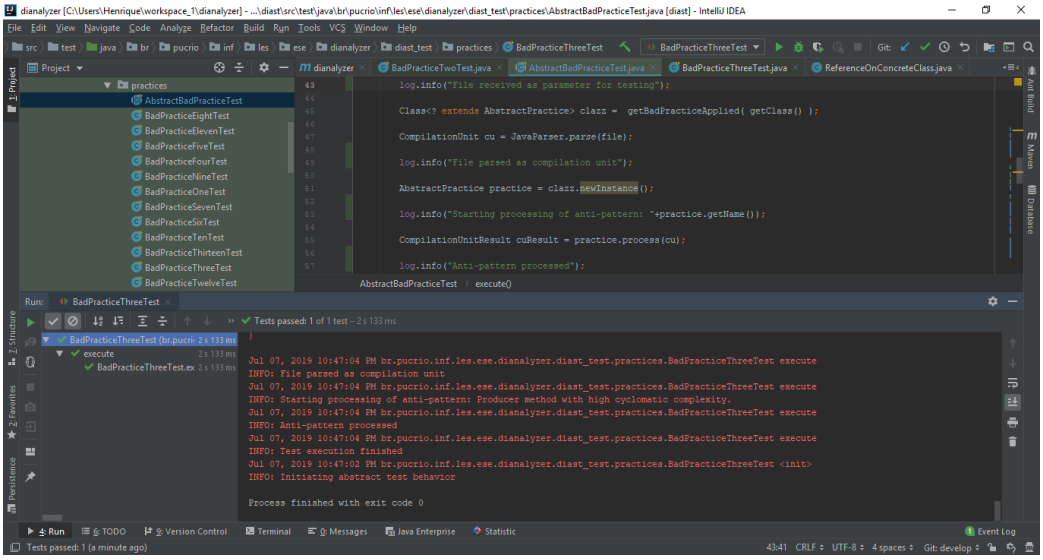


Figure 7: Logs from a test