The Impact of Anti-Patterns to Testing Strategies – A Case Study

Matthias Girkinger
Mobile Computing Master, Department SAIL
FH Oberösterreich Campus Hagenberg
Hagenberg, Österreich
mgirkinger95@gmail.com

***Abstract*—This paper's goal is to determine and evaluate the impact on testing strategies in projects written in a clean and structured way compared to unstructured ones containing a number of anti-patterns. In the first part, the most important technologies for this paper are introduced. The project for this case study features a system with two similar services, one written in clean-code style and one written unstructured with a number of anti-patterns. These systems are the base for the comparison and evaluation.**

I. Introduction

A. Motivation

Structuring a system or program and taking care of certain rules and guidelines is undoubtedly a good decision. Testing and maintaining said system or program are also very important tasks. That brings up the question, what are the downsides when implementing and testing a system or program using certain anti-patterns? Which drawbacks to testing strategies do badly structured or implemented systems or programs have? This paper aims to answer these questions.

B. Introduction to Anti-Patterns

In the project backing this paper there will be many so called anti-patterns used. These anti-patterns will not be discussed in detail, but for the sake of completeness they will be introduced in short. Some of the most important ones we will be taking a look at will be the following.

1) God Class: The God Class, also known as *The Blob* is a software development anti-pattern that usually impacts an entire application. It can generally be characterized by a single controller which is surrounded by a few simple data classes. These classes usually lack cohesiveness of attributes and operations and also often have a general absence of object oriented programming. [1]

2) Functional Decomposition: Functional Decomposition describes the usage of non object oriented usage of classes. It can be, detected by classes being named much like functions, for example `CalculateFoo` or `DisplayBar`. Classes like this often have only private members which are used within a single function. Class models usually make little to no sense in these cases. [1]

3) Spaghetti Code: Programs or systems with little to no software structure appear as *Spaghetti Code*. It is characterized by methods with large implementations surrounded by only a few classes which invoke a single process flow with multiple stages. Systems like these are hard to maintain and difficult to extend as well. [1]

C. Introduction to Testing Strategies

The testing strategies that will be discussed in this paper are split into three abstractions, unit testing, integration testing and acceptance testing. Each of these testing strategies is briefly discussed in this section.

1) Unit Testing: Unit tests are a very large topic with a lot of in-depth theory. This paper only gives a small introduction to unit tests. A unit test is best described as a test of a small code piece, also referred to as a *unit*, which is quick and done in an isolated manner. Measuring perfect execution speed or test scope is highly subjective and can hardly be defined.
There are two definitions for isolation, the *London school* and the *classical school* of unit testing. The London school isolates in units and one unit is always one class in this context. The classical school approach isolates in unit tests which is either one class or a set of classes. [2]

2) Integration Testing: Integration tests verify how your system works in integration with out-of-process dependencies. Out-of-process dependencies can be split into managed and unmanaged dependencies. Unmanaged dependencies are those which come from, for example, an external system and are not controllable from within the system being tested. For these unmanaged dependencies, it is usually best to test them using mocks of the external system because the provided data of an external system could change at any point, potentially leading to failing tests even though the logic is working correctly. Managed dependencies, on the other hand, are dependencies which are controlled from within the tested system. For this type of dependencies, real instances are used. The difference comes from the fact that unmanaged dependencies must be backwards compatible, which mocks are perfect for, as they can be used to mock a given version and new tests must be written when the version changes. This does not apply to managed dependencies, as they are only used within the program and when they are changed, the respective tests are adapted as well. [2] Figure 1 is a visualization of the difference between unit testing and integration testing and how they are determined.

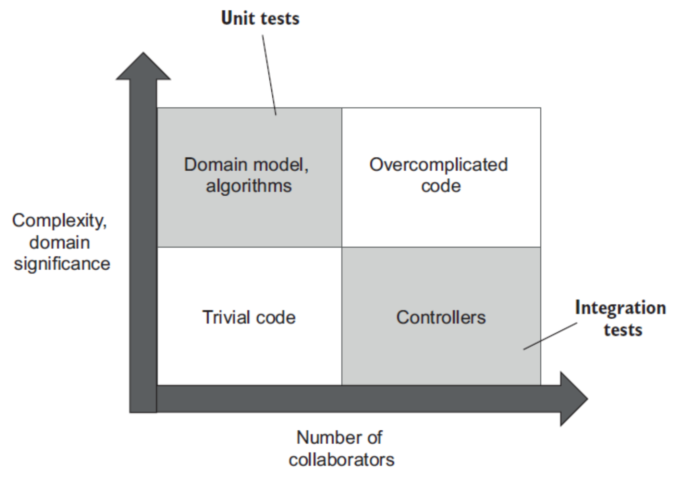

Fig. 1. Integration tests cover controllers, unit tests cover algorithms and the domain model. [2, p. 186]

3) Acceptance Testing: Acceptance Testing, also known as User Acceptance Testing (UAT), is used to refer to end-user testing of software carried out before the system is released. User Acceptance Tests are defined by their implicit use-cases. [3]
For the system implemented in the scope of this paper, it is crucial to determine who the end-users are. As this is a back-end service, the end-users will be the developers integrating the system into a dedicated front-end. In this context, it can be said that what we need for our system to work properly is that we test the data transmission, serialization, deserialization, and computation. An example of this would be to test the transformation from Java Object to JavaScript Object Notation (JSON) format and vice versa.

D. Introduction to Technologies

To create the application for this project, following technologies were made use of:

1) Spring Boot: Spring Boot is a framework used to create stand-alone applications with a minimal amount of configuration necessary. It directly embeds Tomcat, Jetty or Undertow which removes the necessity to deploy WAR archives manually. [4]

2) TestNG: TestNG is a framework inspired by JUnit and NUnit. It provides a lot of annotations and functionality to improve and simplify testing, for instance, the abstract class `AbstractTestNGSpringContextTests` which enables the possibility to create spring context tests, making it a good fit in combination with the Spring Boot framework. [5]

3) Lombok: Project Lombok is a library for Java that provides numerous annotations to simplify Java code, especially data classes. Annotations, such as `@AllArgsConstructor`, `@NoArgsConstructor`, `@Builder`, `@EqualsAndHashCode`, `@Getter`, and `@Setter`, save a lot of boilerplate code (code that is repeated in many places with little to no changes) by generating this code at build time. [6] It is helpful in the context of providing better structure and overview in data classes.

E. SOLID Principles

SOLID is an acronym for 5 important principles concerning object oriented programming. Their goal is to make object oriented programming more understandable, maintainable and reusable. [7]

1) Single-Responsibility Principle: A class should have only one reason to change. [7, p. 95]
Even though the Single-Responsibility Principle (SRP) is one of the simplest principles in theory, it can be difficult to implement correctly. Additionally, it is one of the most important principles as separating responsibilities is a crucial factor in terms of structure and understandability in a program or system. Issues that arise in the upcoming principles have a high risk of stemming from unclear or incorrect separation. [7]

2) Open-Closed Principle: Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. [7, p. 99]
The meaning of the Open-Closed Principles (OCP) is that a software component should be easily extendable to add new functionality without modifying its existing code. This principle encourages the use of abstraction and polymorphism to separate the interface of a component from its implementation. By adhering to the open-closed principle, software is more maintainable, testable, and less prone to errors. [7]

3) Liskov Substitution Principle: Subtypes must be substitutable for their base types. [7, p. 111]
The Liskov Substitution Principle (LSP) is based on the idea of subtyping, where a subclass should be a subtype of its superclass. This means that any methods or properties that are defined in the superclass should also be available in the subclass, and they should have the same behavior. Adhering to this principle promotes code reusability and maintainability by allowing for more flexibility in class hierarchies. [7]

4) Interface Segregation Principle: Classes should not be forced to depend on methods that they do not use. [7, p. 137]
The Interface Segregation Principle (ISP) states that interfaces should be split into smaller, more specific interfaces, so that classes only need to implement the methods that are relevant to them. This principle promotes code reusability, flexibility and maintainability, by breaking down a large interface into smaller ones that are more easily understood and used. [7]

5) Dependency-Inversion Principle: a. High-level modules should not depend on low-level modules. Both should depend on abstraction. b. Abstractions should not depend on details. Details should depend on abstractions. [7, p. 127]
The Dependency-Inversion Principle (DIP) says that a class should not depend on a specific implementation of a module, but rather on an interface or abstraction of that module. This allows for more flexibility and ease of change in the code, as the implementation can be swapped out without affecting

the rest of the code. This principle encourages the use of dependency injection and inversion of control to decouple dependencies and promote loosely coupled code. [7]

II. Project Implementation

This section deals with the implementation of the project created in the course of this paper. The first part deals with the good practice service example and the second part presents the poorly set-up service.
In both instances, the service is a basic order processing service that receives input in the form of JSON via an HTTP POST request body. The implementation process began with the development of a clean code version of the service. As expected, this task proved to be the most challenging aspect of the project, as numerous decisions had to be made during this phase. The system requirements can be summarized as follows:

- The system must be able to calculate an order.
- An order can contain 1 to n items.
- The calculation can be done either in net or in gross mode.
- Each item must contain a VAT rate.
- Each item must contain a net or gross price, respective to the calculation mode.
- The calculated order must contain net amount, VAT amount and gross amount for the complete order as well as every single item.
- The calculated order must be returned in a way that it can be easily used by another system (as JSON).

In short, the system is designed to calculate the total cost of an order, including the net amount, VAT amount, and gross amount for the entire order, as well as for each individual item. It is important to keep in mind that both systems meet these requirements, but they differ in their respective implementations.

A. Clean Code Service

1) Structure: The order service in clean code style has the following project structure:

```
– cleancode
    – model
        – output
            * OrderCalculationResult.java
        • CalculationTypeEnum.java
        * DeliveryAddress.java
        * Item.java
        * Order.java
        * Price.java
    – util
        * BigDecimalUtils.java
        * CalculationResultTransformer.java
        * OrderCalculationUtils.java
    ¦ OrderService.java
    * OrderServiceImpl.java
```

– ... Package; • ... Enum; ¦ ... Interface; * ... Class;

2) System Description: The system uses a clean architecture approach, separating the different concerns into different layers and packages. The ”model” package contains the domain model classes such as `Order`, `Item`, `Price`, etc. which are used to represent the order and the items in it. The ”util” package contains utility classes such as `BigDecimalUtils` and `CalculationResultTransformer`, which are used to perform calculations and transform the order’s result into a `OrderCalculationResult`.
The ”model.output” package contains the `OrderCalculationResult` class, which is the result of the order calculation. The `CalculationTypeEnum` class is used to indicate whether the calculation should be done in net or gross mode, each item must contain a VAT rate and a net or gross price, respective to the calculation mode.
The `OrderService` and `OrderServiceImpl` classes in the root package are responsible for coordinating the different components and performing the calculation, using the classes from the ”model” and ”util” packages.
The project structure follows the Clean Architecture principles, which is a good approach to organize the code and separate the different concerns. The system also follows the SOLID principles, which, as mentioned earlier, is a key factor for a well-designed and maintainable code.

3) Code: The system’s code is generally well structured and follows common clean coding principles such as using clear, meaningful variable and method names, and separating concerns through the use of classes and interfaces. Additionally, the use of the `@AllArgsConstructor`, `@NoArgsConstructor` and `@Builder` annotations from the lombok library simplifies the process of instantiating objects and improves code readability.
The code also follows principles of clean architecture, such as separating the domain model from the application layer through the use of classes such as `Order`, `Item`, and `Price`. However, it could be beneficial to further separate the concerns through the use of interfaces and to create a clear boundary between the domain and application layers.

B. Anti-Pattern Service

1) Structure: The order service including anti-patterns has the following project structure:

```
– antipattern
    * CalculateGrossOrder.java
    * CalculateNetOrder.java
    * SuperItem.java
    * SuperOrder.java
    * SuperOrderService.java
```

– ... Package; * ... Class;

2) System Description: In theory, this system uses the same logic as the one written in a clean code style. However, the implementation is very different as the system lacks a clear package structure and contains all of it’s classes, regardless which use-cases they have, in the root package.

After a close look at the class names, the classes `CalculateGrossOrder` and `CalculateNetOrder` are able to calculate the gross and net values of an order. The `SuperItem` and `SuperOrder` classes represent the item and order objects themselves, and the `SuperOrderService` class handles the business logic for the service.

Now, by analyzing these classes, some potential issues that may arise include tight coupling between the classes and lack of separation of concerns. For example, the `CalculateGrossOrder` and `CalculateNetOrder` classes have direct access to the internal state of the `SuperItem` and `SuperOrder` classes, which makes it difficult to change or test the service.

Due to having only a single method in the classes which contain logic or data classes being overloaded with too many fields, all of these classes violate the Open-Closed Principle.

The `SuperOrderService` class is the God Class in this context. Since this project only consists of one small service, the impact of this pattern can not be shown on a full scale. Imagining that the business logic for this service is to be extended soon, all of the logic would go into this class, most likely even into it's main method, being an example for not only the God Class, but also for Spaghetti Code.

The Single-Responsibility Principle can be discussed for the classes `CalculateGrossOrder` and `CalculateNetOrder`, since, theoretically speaking, calculating an order is a single responsibility. However, it is not sufficient for a class to be responsible for only one calculation option, such as gross mode or net mode. This is a clear example for the Functional Decomposition anti-pattern.

III. Testing

Now that the system and it's functionality are clear, it is time to get into the core part of this case study – the testing.

A. Testing Clean Code Service

Right after the implementation of the order service in clean code was finished, the system and it's components were tested to ensure the requirements were met and the system behaves as expected. All mentioned classes contain a varying amount of `private` methods not mentioned in the following.

1) Unit Tests: To determine which entities of the system can be considered a unit, the project structure definitely is of help. All classes in the "model" package were not considered for unit testing, as these are merely data classes. Further, their functionality is provided by the lombok library and can be expected to work properly. For the classes in the "utils" package, this is a different story. Them being the classes `BigDecialUtils`, `CalculationResultTransformer` and `OrderCalculationUtils`, all representing a respective responsibility.

`BigDecialUtils` currently contains only one `public` method, `nullSafeAdd`, which is responsible for adding two nullable `BigDecimal` instances. To ensure correct behaviour, the test must contain at all possible scenarios, the first addend being `null`, the second added being `null` and both of them being `null`.

The class `CalculationResultTransformer` contains one `public` method, `transformOrder`, which, as it's name suggests, transforms a calculated order into an `OrderCalculationResult`. The reason behind this is to not overlap the input API with the output API. In this case it is necessary to test the conversion and assert that the result is transformed correctly and contains all required fields with their respective values.

Finally, `OrderCalculationUtils` is to be tested, containing two `public` methods, `calulateNetOrder` and `calculateGrossOrder`. They both take an order as input and calculate it in the given calculation mode, *net* or *gross*. The return value in both cases is an `OrderCalculationResult`. To validate the intended behaviour, the result must be checked for the total `Price` and each individual item's `Price` entity's correctness. A `Price` holds a net amount, a vat amount, a gross amount, a vat rate and a currency.

2) Integration Tests: Going back to the root package of the project, the actual implementation for the order service logic, `OrderServiceImpl`, is found here. It contains one method, `calculateOrder`. This method also takes an order as input, but before calculating it relying on the `OrderCalculationUtils`, it validates if an order is processable. In order to process an order, it must contain the same vat rate for every item, as well as net- or gross-prices respectively in every item. This can be tested processing invalid orders and asserting all possible outcomes to be correct.

B. Testing Anti-Pattern Service

Testing the anti-pattern system was the last part of the implementation process for the project. None of the mentioned classes contain any `private` methods.

1) Unit Tests: In regards to the fact that there is no package structure in the anti-pattern system's source code, determining which components are to be tested how is a bit more complex than in the cleanly implemented service. Since the project is small, this is, by all means, doable, but as the system grows, this process becomes more and more tedious, especially if the class names are not clear.

The classes `SuperItem` and `SuperOrder` can be identified as data classes, therefore, they can be excluded from the testing. Most likely, the classes `CalculateGrossOrder` and `CalculateNetOrder` could be identified as something like units, so it makes sense to write unit tests for them. Both classes contain only one `public` method, called `calculate` in both instances. This behavior violates the Open-Closed Principle. The class name implies that it is a method rather than a class, which limits the possibility of extending the class and requires modification within each method if the calculation logic or requirements change. Although the functionality can be tested and should produce correct results, any failure may require a complex and time-consuming adaptation.

2) Integration Tests: For an integration test, the only class remaining is the `SuperOrderService`. However, this class is largely ineffective as there are few components working together. It contains a single method, `calculateOrder`, which also includes validation for an order (in this context, an order refers to a `SuperOrder`). However, all of the validation is contained within this single method. Similar to the unit tests in the anti-pattern service, the tests for this class are less accurate and the class is more prone to errors, increasing the risk for lengthy adaptation sessions.

C. Comparison

If the projects were compared based on total lines of code, the anti-pattern system would be smaller. However, this comparison is not about the size of the program, but about the quality of the system. Specifically, it is about which system is more testable and maintainable, and the impact of the anti-patterns on the testing process.

Compared to the clean code system's tests, the tests of the anti-pattern system are relatively superficial. They test the components for basic functionality, but they do not effectively identify errors or their source. In contrast, the clean code system's tests have well-defined objectives for what to test and how to test it, not just the `public` methods, but also the underlying `private` methods and their respective tasks. These tests provide a clear understanding of where errors occur and why.

This improved testing process leads to more efficient maintenance of the clean code system, as developers can quickly locate and fix errors. Additionally, the clean code system is more scalable, as new features can be added with confidence that the existing functionality will not be impacted. The anti-pattern system, on the contrary, is closed for extension and adding new functionality is a tedious process, potentially taking a massive amount of time to adapt the whole system.

1) Acceptance Testing: Acceptance testing was not mentioned in the earlier chapters on testing. This has a simple explanation. Both systems are able to meet the specified requirements and, in fact, they do. As a result, the testing process is the same for both systems.

The systems are tested using a tool called *OpenAPI* (previously known as *SwaggerUI*), which is an API documentation tool provided by *SmartBear Software*. [8] The testing process was conducted manually. To do this, a JSON file representing an `Order` and another JSON file representing a `SuperOrder` are created. These files can be easily edited due to the human-readable structure of JSON files.

The files were edited to include valid or invalid values, then processed through the Swagger user interface. A POST request is sent to the API containing the JSON file in the body, and the processed result is obtained in the form of another JSON file from the response body. In the case of the clean code service, an `OrderCalculationResult` is returned, while the anti-pattern service returns a `SuperOrder` in the same format as its input.

The result must be manually verified, which is not too different for both projects. However, the anti-pattern system faces the same problem as with its unit and integration tests: it is extremely difficult to identify the root cause of an error if one occurred.

2) Sonar Analysis: In order to further compare the systems, another tool called *Sonar* was used. This tool analyzes a projects code for potential code smells and errors. It can be combined with a tool called *Java Code Coverage (JaCoCo)*, in order to generate and show a detailed report on which lines of code are covered by tests. Naturally, the clean code project offers way better coverage than its anti-pattern counterpart.

In total, there are 14 unit tests, four of which are found in the anti-pattern system and ten being found in the clean code system. This may not seem much on a small scale, but the anti-pattern system contains less than half of the tests. Although only having one method to be tested, the `SuperOrderService` has the worst test coverage on all relevant classes, scoring only 71.1%.

For reference, this test report can be found in the projects root folder if further information is required.

IV. Conclusion

In conclusion, the clean code service follows good practices such as the clean architecture approach and SOLID principles, resulting in a well-structured and maintainable code. On the other hand, the anti-pattern service lacks clear package structure and has all its classes in the root package, making it less organized and harder to maintain. The impact of the anti-patterns on the service is significant, as it reduces the overall quality of the code and makes it more challenging to test. The tests for the anti-pattern system are way less speaking compared to the tests for the clean code system, resulting in less coverage of code in the systems components. Also, errors that are detected can not be easily adapted due to the general structure in classes as well as in code. In summary, the choice of using a clean code approach can greatly improve the quality and maintainability of a project, making it easier to test and debug in the future.

References

[1] A. Shvets, "Dive Into Design Patterns", Refactoring.Guru, 2018

[2] V. Khorikov, "Unit Testing Principles, Practices, and Patterns", Manning Publications Co., Shelter Island, NY, 2020

[3] B. Hambling and P. van Goethem, "User Acceptance Testing A step-by-step guide", BCS, The Chartered Institute for IT, Swindon, United Kingdom, 2013

[4] VMWare and Spring Authors and Contributors, The Official Spring Website, [https://spring.io/projects/spring-boot, Online, Accessed 15.01.2023]

[5] Apache and TestNG Authors and Contributors, The Official TestNG Website, [https://testng.org/doc/, Online, Accessed 15.01.2023]

[6] The Project Lombok Authors and Contributors, The Official Project Lombok Website, [https://projectlombok.org/, Online, Accessed 15.01.2023]

[7] R. C. Martin, "Agile Software Development Principles, Patterns, and Practices", Pearson Education Inc., Upper Saddle River, NJ, 2003

[8] SmartBear, OpenAPI and SwaggerUI Authors and Contributors, The Official Swagger Website, [https://swagger.io/tools/swagger-ui/, Online, Accessed 29.01.2023]