

How Test-Driven Development Influences Class Design: A Practitioner’s Point of View

Mauricio Finavaro Aniche, Marco Aurélio Gerosa
Department of Computer Science
University of São Paulo (USP)
São Paulo - Brazil
{*aniche, gerosa*}@ime.usp.br

Abstract—Test-Driven Development (TDD) is the practice in which programmers write the test before the implementation. Although being usually addressed as a testing practice, TDD is actually a design technique, as it supports developers in the evaluation of the design quality. The practice leads programmers to design classes that can be easily tested, which is highly related to good design. This paper discusses the effects of the practice on the class coupling and cohesion, through the point of view of a TDD practitioner.

Keywords-Test-Driven Development, Software Design, Object-Oriented Systems, Agile Practices, Test Feedback on Design.

I. INTRODUCTION

Good OO programming states that **classes should be low coupled and highly cohesive**. However, design classes or modules that follow this principle is not easy. Consequently, after some time, the design commonly loses quality and the maintenance becomes hard and expensive.

To avoid this problem, developers constantly validate the quality of their design by means of several different practices, such as code revision, pair programming, code metrics, etc. Many developers also believe that writing unit tests is useful to validate the design.

Test-Driven Development (TDD) is one of the agile practices that focus on feedback. In a more formal definition, TDD is the craft of producing automated tests for production code, and using that process to drive design and programming. For every tiny bit of functionality in the production code, programmers first develop a test that specifies and validate what the code will do. Programmers, then, produce exactly as much code as will enable that test to pass. Then they refactor (simplify and clarify) both the production and test code [1].

TDD encourages developers to write easily testable code, which leads to several interesting characteristics, such as the focus on what a class should do rather than how it does, high-level of cohesion, and better management of dependencies that a class may have – the same characteristics a developer expects from a good design. According to Feathers [5], there is a synergy between testability and a good design. When looking for testability, developers end up with a good

design; when looking for a good design, they end up with a testable class.

Robert Martin [4] relates TDD and professionalism. In his opinion, a professional developer delivers clear and flexible code that works, on time. TDD supports developers reaching these goals. Academic experiments that evaluate the effects of TDD in internal quality, such Janzen et al. [7] and Williams et al. [8], also found that TDD helps developers creating simpler and more maintainable designs.

Traditional approaches have testing only after the feature is completely implemented and, consequently, they do not have the tests feedback during the classes initial design. Figure 1 illustrates two programmers using different approaches, writing small pieces of code for a feature. TDDers constantly validate the design through tests, what does not happen in traditional approaches where many small pieces of code are written before writing the tests.

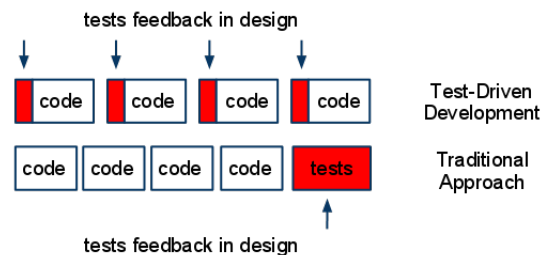


Figure 1. The difference from test feedback when doing TDD or not.

Most literature affirm that TDD is a design technique, but do not deeply discuss how tests or the testability effects influence on the final result. This paper presents how TDD, unit tests, and testability stimulates the development of a maintainable and evolvable code base.

The given examples during this paper were based on a real experience of a TDD practitioner on a team of 5 developers within a company with around 100 developers. All team members have good experience in agile, object oriented design, and TDD.

II. TDD AND CLASS DESIGN

Most of the tests feedback on design is based on the fact that **if it is hard to test a class in an isolated way, then it may have a problem with its design**. The harder to write a unit test, the bigger the possibility to find a design smell.

Writing a unit test first makes developers anticipate concerns. A simple unit test, as the one presented in Listing 1, makes developers think about the expected behavior from the class. Developers decide about its public interface, name, return types, or even exceptions thrown, before writing the actual class.

```
@Test
public void
    shouldCalculateTaxesInAnInvoice () {
    InvoiceFactory factory = new
        InvoiceFactory ();
    Invoice invoice = factory .build (
        customer );
    assertEquals (customer .getTotalAmount ()
        * 0.2, invoice .getTaxesAmount ());
}
```

Listing 1. A unit test for an InvoiceFactory

It may look like simple decisions, but the same happens in higher levels of abstractions; developers think better about the abstractions they are about to create and how these classes should communicate to each other.

Developers are encouraged by TDD, not only to create low coupled classes, but to couple classes to more stable abstractions. Moreover, design tends to be simple, as programmers write only code that they are going to use. On cohesion, instead of writing big classes with many lines of code, TDD encourages developers to write simple and small classes, and use the power of object orientation mechanisms to put all the classes together.

The following sub-sections discuss the effects of TDD in both coupling and cohesion.

A. Effects on Coupling

It is easy to write coupled classes. Developers sometimes do not even notice that they are writing them, as the problem takes some time to appear. Indeed, highly coupled classes tend to be hard to maintain.

In order to test, TDD encourages developers to follow some good object orientation principles: classes explicit their dependencies, and therefore become opened for extensions; coupling usually tend to be to high-level stable interfaces, reducing the risk of a modification caused by changes in any dependency.

1) *Expliciting dependencies*: The Open-Closed Principle (OCP) states that classes should be open for extension, but closed for modifications [2]. According to this principle, a module can have its behaviour modified without the need

of altering its source code. This is an important principle to follow because, when evolving a system, a modification should not be propagated to many classes.

When writing a unit test for a class that collaborates with another, the class must be opened for extension. If not, developers are not able to test that unit in an isolated way. One way to make the class opened for extensions is to create a constructor that receives all dependencies, rather than to make the class instantiate all of them by itself.

TDDers write classes that receive all dependencies, as it enables the unit test to pass an implementation that mocks the behavior of the expected dependency, and therefore test the class isolated from the rest. The simple fact that classes have their dependencies injected also enables classes to be easily evolved. Different implementations can be passed to the class, changing its final behavior without actually modifying it.

2) *Focus on “what” rather than “how”*: **High-level modules should not depend upon low-level modules; both should depend upon abstractions. Abstractions should not depend upon details; details should depend upon abstractions.** This is called the Dependency Inversion Principle (DIP) [2].

TDD encourages developers to follow this principle. When writing a test for a new class, developers are focused only on the expected behavior of that class. If this behavior is complex, the class will make use of other classes. However, as unit tests isolate the class under test from the rest, TDDers are not interested on how other classes implement their behavior at that moment. They only create an abstraction that represents the dependency. This abstraction is usually implemented through an interface.

As an example, take a class named *CheckPaidInvoicesJob*, that is responsible to get all the invoices, check their status in the billing system, and mark them as paid if the payment is confirmed. The implementation of all process involves several low-level details, such as communication with databases or web services. However, when writing the test, a TDDer is not concerned with these details, but only with the final expected behavior. Because of that, when all invoices are required during the algorithm, a TDDer creates an interface which is only responsible for it. The same happens with the checking process in the billing system.

The emerged interfaces only describe what the class should do, rather than how it should be done. The way a concrete class implements the interface is not important at that moment, as the main class depends only upon the abstractions. The programmer will implement it later on. Program to an interface and not to an implementation is also another good design principle [6]. Figure 2 shows a diagram that represents the class and both emerged interfaces.

3) *Avoiding God Classes*: Highly coupled classes, also called God Classes, tend to be hard to maintain. As discussed before, TDDers tend to emerge many interfaces during

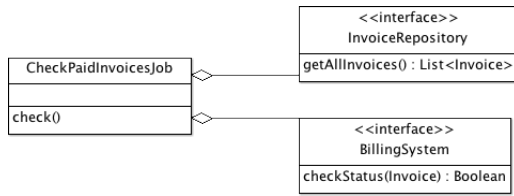


Figure 2. The class CheckPaidInvoicesJob and its dependencies.

the programming session, which may suggest that a TDD practitioner may end up with classes that contain several dependencies.

God Classes do not tend to appear during a TDD practice. When writing a test for a class like this, developers face a complex situation as they have to create one mock for each dependency, and set the expected behavior on each one. Classes with many dependencies also tend to have a lot of responsibility, which increases the number of scenarios to be tested, making it even harder to test.

The focus on what a class should do is the key point to avoid this problem. As an example, take a class A that depends upon different classes B, C, D, and E. In order to execute the behavior, class A should invoke B and pass its output to C, which generates the result that class A expects. The same happens with class D and E.

To unit test this class, a developer only needs the result of these interactions. Because of that, TDDers usually separate these concerns in different interfaces. As shown, if the interaction between B and C is the one that produces the expected output to class A, it may indicate that B and C can be extracted to a new interface, which is only responsible to do that.

A TDD practitioner creates new abstractions X and Y, which are responsible to generate the specific output that class A expects. Class A now depends only upon X and Y, reducing the coupling. Figures 3 and 4 exemplifies the solution.

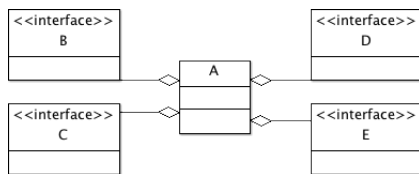


Figure 3. Class A depending upon B, C, D and E.

4) *Stable classes*: When a class depends upon another, a change inside the dependency may affect that class. If dependencies change often, it would possibly force all classes connected to them to change together. On the other hand, stable classes are those that are not constantly modified. As they do not tend to change, they do not propagate modifications to classes that are connected to them. Based on that, Robert Martin derived the Stable Dependencies

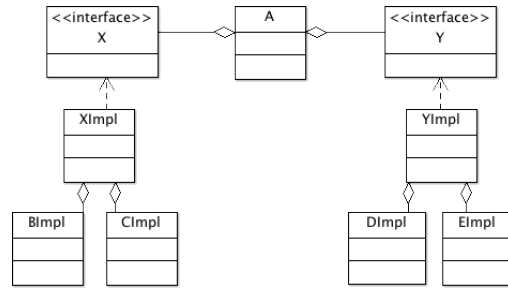


Figure 4. Class A depending upon X and Y.

Principle (SDP), which states that a **module should depend only upon modules that are more stable than it is** [2].

When practicing TDD, developers tend to follow the SDP. As an example, suppose a class that reads and writes some content. It depends upon *Sap* and *SqlServerDao*. These classes deal with low-level details, which may change and cause a series of modifications in classes that depends upon them. However, as mentioned in Section II-A2, TDDers create interfaces that represent the behavior they expect. In the example, when creating this class, a TDDer creates the interfaces *FinancialSystem* and *Repository*.

These two interfaces, exemplified in Figure 5, tend to be more stable than the first two classes. First, these interfaces depend upon nothing, which means that no changes would be caused by their dependencies. Second, these interfaces will have many classes that implement their contract, preventing programmers to change it – if they do, they would have to change all implementations.

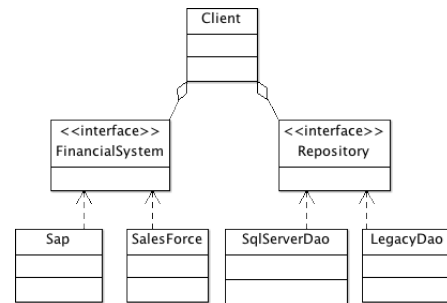


Figure 5. Interfaces FinancialSystem and Repository tend to be stable.

That may suggest that good dependencies are those stable; a bad dependency is the one unstable. Interfaces that constantly emerge from the practice tend to be stable as they only specify what, and not how. When practicing TDD, developers tend to couple their classes to abstractions that are stable, and therefore, making good dependencies more often.

B. Effects on Cohesion

Creating highly cohesive classes demands a huge effort by the developer. Simply adding more code inside of an existing

class is usually easier. However, the Single Responsibility Principle (SRP) states that classes should have only one responsibility, and therefore only one reason to change [2].

As low cohesive classes tend to be hard to test, TDDers constantly try to isolate each concern in a separate class, and group them together only afterwards, using class composition techniques.

1) *Small classes:* Tests are classes like any other. When a test is validating a production class, it only invokes methods that are allowed (the public ones). When classes grow, developers start to break the public method into a set of private methods, reducing the size of them. However, even these small private methods can grow. When facing this situation, developers sometimes change the method's access modifier in order to test it.

The desire to change the method's visibility for testing happens when the behavior encapsulated inside the public method is too big. Indeed, the class is possibly suffering from low cohesion. TDDers tend to notice that as testing low cohesive classes only through its public API requires too much effort to set up scenarios and to validate the outputs.

In order to make it easier tested, a TDD practitioner extracts responsibilities to specific classes, and creates the desired behavior by means of class composition. Figure 6 shows an example of it.

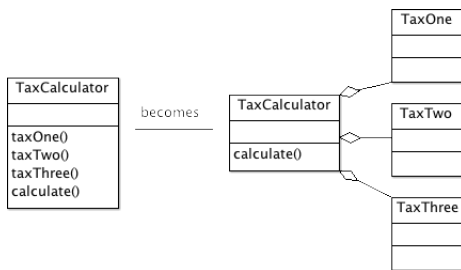


Figure 6. A class with many lines of code extracted into many classes.

2) *Composing Behaviors:* As said before, when classes have many responsibilities, testing is hard, and this situation may be avoided by creating well-specialized classes. However, when writing a class that represents a bigger behavior by making use of many of these small classes, a developer can create a low cohesive highly coupled class.

An example might be a class that is responsible for generating invoices and, after that, sends them to the customer's e-mail and to some ERP program. Although each responsibility is held in a specific class, a developer needs to put them all together. Besides the low cohesion, this main class may become a God Class if its responsibility grows even more. Figure 7 illustrates it.

In order to test this, developers need to make use of many mock objects, and setting all their expectations. It requires much effort by TDDers, which makes them rethink about the design. In the example, the main responsibility of the class

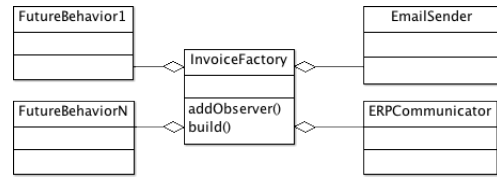


Figure 7. InvoiceFactory highly coupled to the behaviors.

is to generate invoices. The others are just a complement of this behavior, and the main class should enable them to be easily plugged.

Many design patterns suggest a way to solve this problem, composing different behaviors and keeping the coupling still low. Factories, Builders, Decorators and Observers are good candidates [6]. TDD encourages developers to use good OO practices, in order to put the all behavior together.

The example above can be improved if the *InvoiceFactory* is responsible only for the invoice generation, and the other behaviors implement some observer interface. The factory, after processing the invoice, notifies all observers. Figure 8 explains the solution.

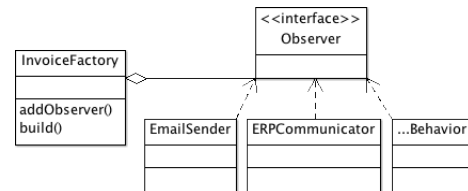


Figure 8. InvoiceFactory and the Observer Pattern.

This simple improvement has many positive effects on design. It reduces the *InvoiceFactory* coupling as it only depends on one interface, and enables the class to naturally evolve as new behaviors can be easily added. In addition, the *Observer* interface is a good candidate to be stable, and therefore a good dependency.

III. CONCLUSION

Developers validate the quality of their design constantly, with any practice they find useful. This paper advocates that TDD is one of them. The given information about both coupling and cohesion are useful to drive the design. Tests warn developers about possible design smells. Nevertheless, it is up to developers to notice the problem and fix it.

Interestingly, the same design feedback discussed on this paper can be used by developers that do not make use of TDD. Experience and knowledge in software development is clearly the most important factor when trying to write good object oriented systems, and TDD does not create maintainable designs by itself. However, TDD gives feedback about it constantly, which may help even the most experienced programmer.

Although many evidences are supported by academic studies, there are still space for deeper researches about the effects of TDD in software design. Meanwhile, developers may try the practice and see its effects in practice.

ACKNOWLEDGMENT

The first author would like to thank Caelum Learning and Innovation for the support and feedback provided during the writing.

REFERENCES

- [1] Agile Alliance. TDD. http://www.agilealliance.org/programs/roadmaps/Roadmap/tdd/tdd_index.htm, 2005.
- [2] Martin, Robert C.. Agile Software Development, Principles, Patterns, and Practices. Prentice Hall, first edition, 2002.
- [3] Beck, K. Test-Driven Development By Example. Addison-Wesley Professional, first edition, 2002.
- [4] Martin, R. Professionalism and Test-Driven Development. IEEE Software, volume 24, pages 32-36. IEEE Computer Society, Los Alamitos, CA, USA, 2007.
- [5] Feathers, M. The deep synergy between testability and good design. http://michaelfeathers.typepad.com/michael_feathers_blog/2007/09/the-deep-synerg.html, 2007. Last access in October, the 27th, 2010.
- [6] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. 1st edition, 1994.
- [7] Janzen, D., Saiedian, H. On the Influence of Test-Driven Development on Software Design. Proceedings of the 19th Conference on Software Engineering Education and Training (CSEET'06), 2006.
- [8] George, B., Williams, L. An Initial Investigation of Test Driven Development in Industry. Proceedings of the 2003 ACM symposium on Applied computing, ACM, 2003.