Achieving Quality on Software Design Through Test-Driven Development

Eduardo Guerra and Maurício Aniche

Abstract Test-driven development (TDD) is a technique for developing and designing software where tests are created before production code in short cycles. There is some discussion in the software engineering community on whether TDD can really be used to achieve software quality. Some experiments were conducted in the last years comparing development by using TDD with one creating tests after the production code. However, these experiments always have some threats to validity that prevent researchers from reaching a final answer about its effects. This chapter, instead of trying to prove that TDD is more effective than creating tests after, investigates projects where TDD was successfully used, and presents recurrent and common practices applied to its context. A common mistake is to believe that just by creating tests before production code will make the application design ”just happens”. As with any other technique, TDD is not a silver bullet, and while it certainly helps to achieve some desirable characteristics in software, such as decoupling and separation of concerns, other practices should complement its usage, especially for architecture and design coherence. In this chapter, we dive deep in TDD practice and how to perform it to achieve quality in software design. We also present techniques that should be used to setup a foundation to start TDD in project and to refine the design after it is applied.

1 Introduction

Test-driven development (TDD) has become popular among developers as it is one of the most important practices in any agile method, such as Extreme Programming

Eduardo Guerra
National Institute of Space Research (INPE), Brazil, e-mail: guerra@guerra.com

Maurício Aniche
Caelum Ensino e Inovação, Brazil e-mail: mauricio.aniche@caelum.com.br

[Beck(2004)]. The practice suggests developers to write a unit test before the production code in short cycles [Beck(2002), Astels(2003)]. In practice, a developer first introduces a new failing test, and then writes the smallest piece of code possible in the production code in order to make the test suite to execute successfully. As a final step, the code is refactored, if necessary, to provide a better structure to the current solution.

There is some discussion in software engineering community about whether TDD can be used to achieve software quality. Although the practice contains the word *"test"* on its name, a common speech among practitioners is that TDD drives developers towards a better code and class design. The idea is reinforced by many well-known book authors, such as Kent Beck [Beck(2002)], Robert Martin [Martin(2002)], Steve Freeman [e Nat Pryce(2009)], and Dave Astels [Astels(2003)].

Many different studies and controlled experiments were conducted in the last years evaluating the effects of TDD through many different point of views, such as internal and external quality, productivity, learning difficulties, etc. Just to mention a few, Janzen [Janzen(2005)] shows that TDD practitioners (from now on, called TDDers) developed code that contained 50% less bugs than the code developed by non-TDD practitioners. George and Williams [George and Williams(2003)] show that, altough TDDers were less productive, their code had better external quality than non-TDDers. They also present that those TDDers believed to spend less time debugging their code. Janzen again [Janzen and Saiedian(2006)] shows that his students, when practicing TDD, used more object-oriented concepts than the ones that did not practice TDD. Li [Li(2009)] shows, by means of a qualitative study with professional developers, that TDD leads developers to a simpler class design, a consequence of the constant refactoring.

However, as with any scientific work, there are always some threats to their validity, so that it prevents researchers to reach a final answer about its effects. In practice, getting all these benefits is not simple as it looks like. Some researchers have noticed that the effects of the practice (such as improving class design) are not natural or clear as expected [Aniche et al(2011)Aniche, Ferreira, and Gerosa, Siniaalto and Abrahamsson(2008)].

TDD is not a silver bullet. In this chapter, instead of trying to prove that TDD is more effective than traditional development, we will focus on discussing how developers can improve their code and design through the use of TDD. We will dive into the practice and present how it should be used in order to achieve quality in software design, such as high cohesion, low coupling, and less complexity. The end of each section presents an objective list of recomendations that summarize what is being said. We present other design techniques that can complement TDD in fields that are out of its scope. We expect that, after reading this chapter, a developer will be able to practice TDD and deliver high quality software, in a very productive pace.

2 Evidences on the Influence of TDD on Software Quality

Most studies that investigated the effects of unit testing and TDD on production code focused on external quality (e.g., bug proneness). In this section, we present studies that focus on the quality of the design and other aspects of internal quality. It is worth to notice that many empirical studies that evaluated the effects of unit testing in class design relied on the TDD technique, as it highlights the writing of unit tests and their use to reflect upon the design.

Janzen [Janzen(2005)] showed that the code complexity was much smaller and the code coverage was higher in code written using TDD. Another study by Janzen [Janzen and Saiedian(2006)], with three different groups of students, showed that the code produced using TDD made better use of object-oriented concepts, and responsibilities were better distributed into different classes, while other teams produced a more procedural code. In addition, tested classes were 104% less coupled than non-tested classes, and methods were 43%, on average, less complex than the non-tested ones.

George and Williams [George and Williams(2003)] showed that, although TDD can initially reduce the productivity of inexperienced developers, 92% of the developers in a qualitative analysis thought that TDD helped to improve code quality. 79% believed that the practice promotes a simpler class design.

A study by Erdogmus *et al.* [Erdogmus et al(2005)Erdogmus, Morisio, and Torchiano] with 24 undergraduate students showed that TDD increased productivity. However, no difference in code quality was found.

Langr [Langr(2001)] showed that TDD increased code quality, facilitated maintenance, and helped to produce 33% more tests when compared to traditional approaches.

Dogsa and Batic [Dogsa and Batic(2011)] also found an improvement in class design when using TDD. According to the authors, the improvement was a consequence of the simplicity that TDD adds to the process. They also affirmed that the test suites created during the practice favors constant code refactoring.

Li [Li(2009)] conducted a case study in which she collected the perceptions of TDD practitioners about the benefits of the practice. She interviewed five developers from software companies in New Zealand. The results of the interviews were analyzed and discussed in terms of code quality, software quality, and programmer productivity. Regarding code quality, Li concluded that TDD guides developers to simpler and better-designed classes. In addition, the main factors that contribute to these benefits are the confidence to refactor and modify the code, a higher code coverage, a deeper understanding of the requirements, a code easier to understand, and the elevated satisfaction of the developers.

TDD practitioners usually make use of other agile practices, such as pair programming. This makes the evaluation of the practice more difficult. Madeyski [Madeyski(2006)] observed the results of groups that practiced TDD, groups that practiced pair programming, and the combination of them, and he was not able to show a significant difference between teams that used TDD and teams that used pair programming in terms of class package dependency management. However, when

combining the results, he found that TDD helps to manage dependencies at class level.

Muller and Hagner [Muller and Hagner(2002)] showed that TDD does not result in better quality or productivity. Steinberg [Steinberg(2001)] showed that code produced with TDD is more cohesive and less coupled. Participants also reported that defects were easier to fix.

3 TDD as a Design Technique

The first references about TDD state that it is a technique that can be used for software design and development [Beck(2002), Astels(2003)]. Based on them, the software API is designed throughout the tests, defining how a client of that code, which can be any other module, should interact with it. The internal design is performed through refactorings, which should be considered in every cycle.

Of course, there are many other ways to use TDD. It also can be used only for the implementation. In this approach, the software is designed in a previous step, perhaps by using diagrams, and the classes APIs are already defined when the TDD is performed. The internal class design is the only one that is performed during TDD in this scenario. By having a previous design, the TDD is used only as an implementation technique that helps the development to move forward more safely and to achieve a good test coverage on the software classes. Another varion of the practice is what the industry calls *Acceptance TDD (ATDD)*. Practitioners develop acceptance tests using TDD mentaility. By using this approach the tests consider the software as a whole and does not focus on the definition of its internal design.

It is important to state that this chapter considers TDD as a *design technique*, as a tool that can be used in the context of software development to achieve internal quality. By using TDD for design purposes, the test code is used to express the desired design that will be further implemented on production code. In this context, an analogy can be drawn between tests and UML diagrams, since they are different approaches to represent the software design. As in a diagram, the developer should know the design that s/he wants to express in order to represent it. The design does not ”just happens” because of the tool that is being used to express it.

However, as a design tool, TDD tries to guide the developer in the direction of some good principles of object-oriented design. While it is wrong to say that by using only TDD all the design problems will be addressed, it is correct to say that it encourages the usage of some important design values. The next paragraphs highlight these values, explaining why TDD mechanics can drive the developer towards their application.

Some design techniques, specially the ones that use diagrams upfront, encourage the software designer to think further of the current need of the application. Following this approach, the designer tries to predict future extension points and classes, adding them early into the class structure. TDD states that the developed code should be the simplest possible to make the current test suit to pass. This practice ensures

that the design focus is on the current functionality, and does not on assumptions or predictions that can lead to an overdesigned structure. So, **simplicity** and **focus on current functionality** are two design characteristics that are achieved by this mechanic.

There are different styles of TDD that use different kinds of test for the development. Despite the tests can be functional or focused on the component, unit tests are the most common type of test on TDD, since the original books about this technique encourage its usage [Beck(2002), Astels(2003)]. To unit test a class, it should be decoupled from its dependencies, since the test should verify its behavior apart from them. The technique that is usually applied to achieve this kind of test is the usage of test doubles, such as stubs or mock objects. Since the tests are created before, driving the class design, the capacity to decouple the dependences to enable their substitution for test doubles is naturally included on the developed classes. So, **low coupling** is an important design value that can be achieved by performing TDD with unit testing.

Since all the design is driven by tests, it is expected that **testability** is an attribute that happen naturally by using TDD. Since the test code is written first, developers intuitively define an API that can be easily tested. This also happen with the functionality, which is divided in small portions that make viable the test of them separately. The usage of TDD encourages this characteristic because it is easier to test small pieces of functionalities than to create tests for logic that mix different application concerns. Looking from this point of view, TDD can help the classes to have a **higher cohesion**, since it is easier to group the small amount of functionality that makes sense together. Despite TDD does not help in the definition and division of modules, it can be state that TDD helps to achieve **modularity** by encouraging the software division in smaller modules.

Another characteristic of TDD from the design point of view is that the software application design is considered a continuous task that is performed through the source code implementation. The presence of a refactoring phase on the small TDD cycles reinforces that developers should always look at the code and think if there is another solution or structure that can be implemented, better than the first one. Because of that, it can be affirmed that TDD considers the **design evolution** every time that a new functionality is added into the system. This is an important property for a design method to be used in agile methodologies, in which the requirements can change and the software is delivered in small iterations.

Based on that, it is possible to observe that TDD has several characteristics on its mechanics that reinforce the introduction of several design qualities on software. However, these qualities are not enough to guarantee that a suitable design is used in for the problems of the developed application. Some important points of an application design, such as the domain model and the architecture, are out of TDD scope. The next sections present why TDD does not help in such design domains and what other practices and techniques can be used to complement it.

Summary of recommendations:

- Define in the beginning of the project if TDD will have the role of a software design technique.
- If TDD is going to be used for design, face tests as a tool to define the class design, making its code maintenable.
- Define the tests considering only current requirements, and develop the current application considering only those.
- Define the test approach that will be used to test each kind of component in the architecture.
- Create test suites for different components by grouping tests related with the same concern, to achieve cohesion and modularity.
- Do not try to find the definitive solution on the first version, using the tests to support the code refactoring safely.

4 Modeling Relations with TDD

On object oriented design, the dependence between classes and objects is an important point. The division of responsibilities among the system classes is important to achieve cohesion on a single class and decoupling between classes that collaborate. On classical upfront design methodologies, a diagram is usually applied to represent the relation between the classes and interfaces, such as composition, aggregation, dependence, inheritance and abstraction. In TDD all these relationships can be modeled, but in a different way.

The goal of this section is to present how modeling of class relations are done when TDD is used as the design technique. In order to do that, the first subsection presents the main concepts about mock object, followed by a subsection that explain how they are used to model dependencies. Further, the final subsection presents how hierarchy and abstractions rise in a software developed by TDD.

4.1 Mock Objects

An important tool to model dependencies in TDD, such as aggregation and composition, is *mock objects*. A mock object is a test double that can replace dependencies for test purposes [Mackinnon T.(2001), Freeman et al(2004)Freeman, Mackinnon, Pryce, and Walnes]. As the mock is able to replace the dependency without the class awareness, it should share an abstraction, such as a superclass or an interface, with the actual dependency. The mock object also can have other two responsibilities: to emulate the behavior needed for the test scenario and to allow verifications to be performed on the dependence invocations.

The behavior emulation is used to create different test scenarios for the class that is being developed. This should be used when the behavior of the tested class depends on the return of a dependency invocation. The effect is not necessarily a method return, but can also be an exception or a modification on a parameter variable.

The mock object should also be able to record the invocations that it receives in order to perform verifications. These verifications have the goal to check if the developed class is performing the expected invocations in its dependencies. The verifications can focus on which method is invoked, on the number of invocations, on the method parameter values and on the method invocation order. Which one should be used will depend on the class requirements about how it should interact with its dependencies based on each ones responsibilities and collaborations.

The mock objects can be created as helper classes used for test. These classes should implement the dependency abstraction and provide internal logic to record the information needed from method calls and to allow the configuration of the expected behavior to simulate different test scenarios. Additionally, this class should also provide additional methods in order to allow the verifications to be invoked on the test method when needed.

However, the most common approach for creating mock object is by using mock frameworks. By following this approach, the mock characteristics, such as its behavior and expectations, are defined on the test method itself and generated dynamically by the framework. For more complicated mocks, the usage of mock frameworks makes the creation of it less painful. Another advantage is that all the testing logic is defined in the test method, reducing the indirection.

Summary of recommendations:

- Use mock objects to simulate a dependency of the tested class in a unit test.
- The mock object should be used to both verify the expected calls of the tested class and simulate the behavior of teh dependency for the test scenario.
- Mock frameworks can help in the definition of complex mocks, and keep all the definitions in the test method.

4.2 Designing Dependencies

The modeling of dependencies by using TDD happens by the definition of mock objects. When creating a unit test for a system class, the developer should be aware of the class responsibilities and its possible collaborations. This definition does not need to be formally defined in a diagram or in a document. However, it should be based on the design solution that the developer wants to apply. It can be based on the separation of concerns from reference architecture or on the participants of a design pattern that is being implemented.

When the test is being created, the definition of a mock can be used to model the dependency API and the division of responsibilities between the developed class and the dependency [Guerra et al(2013)Guerra, Yoder, Aniche, and Gerosa]. The first step is to create the mock and add it to the developed class. How this object is added to the developed class is an important design decision, because it defines how the dependency is bound to its lifecycle. For instance, if the dependency is added in the constructor, it is usually a mandatory dependency that will follow the class on its lifecycle. However, if it is added by using a setter method, it can be changed during the tested object existence. At last, the dependence can be a parameter in a method, which means that it is used only locally and it is not attached to the target class. In other words, how the dependence is inserted in the tested class reveals how both object's lifecycle relates.

The next step is to define which methods of the dependency, the tested class on a test scenario, should call. When a developer defines in a test that a mock method should be invoked, s/he is actually defining what kind of collaboration the developed class needs from its dependency in that scenario. This is the test driven approach to express test the role of each class in this collaboration. It clearly defines the contract between them to enable them to cooperate.

It is important to highlight that the mock is usually defined based on an abstraction, such as an abstract class or an interface. So, all the methods defined for the mock object, except the ones used for verification, are actually defined in this abstraction. So, following this approach, the contract between the class and the dependence is defined, and the actual dependence implementation can be performed later on a further TDD session.

Although that the dependency modeling can be done by using mock objects, not all dependencies should be mocked. A bad consequence of mocking is that the dependency's contract become coupled to the test code, which makes it harder to be modified. Because of that, a mock should be used to design more stable contracts. The ones that are important for the application's internal API or to the division of roles according to the adopted architecture. Internal dependencies that do not have or should not have impact on the external class API, can be hidden from the test, making it easier to be refactored in the future.

In this sense, to create or not a mock object for dependency can be considered a design decision on TDD modeling. When a mock is created and defined on test, there must be a meaningful relation in the system context, and it should be more stable. When it is decided to not create a mock, there is a local collaboration which is not important for the system as a whole, but only to achieve the class responsibility. It allows the adopted approach to be refactored in the future without a huge impact on other parts of the system.

When the dependency of the developed class already exists and has a well-defined API, there is no need to design the dependency. When that happens, the TDD can be performed by using a mock object in order to isolate the behavior of the developed class. But it can also be done considering the whole component, including the target class and the dependency on the test [Fowler(2007)].

Summary of recommendations:

- Define how the dependency is related to the tested class lifecycle by how the mock is introduced on the object for test.
- Create methods on the dependency interface based on the tested class needs for external class collaboration.
- Use mocks to design meaningful system dependencies and avoid creating mocks for internal depencies that are encapsulated inside the tested class.
- Use mock creation as a way to express design decisions.

4.3 Hierarchy and Abstractions

In order to create the mock object of a dependency, its abstraction should be defined. Based on this approach, TDD is a design technique that encourages the modeling of the interface apart from the behavior. This is how the initial abstractions emerge in the TDD development. The abstraction can be used to develop classes with different behaviors, and also to develop proxies and decorators that add functionality on different classes from the same abstraction.

By having an abstraction, several implementations that follow the abstraction can be develop to fulfill different behaviors for a given component. When a common behavior is detected on these different components, a refactoring towards the usage of inheritance can be performed. Following this approach, common methods and attributes can be pulled up in order to capture on the superclass the common behavior of that kind of class.

The inheritance can also emerge when a single class has several conditionals that reflect different behaviors that it can have based on an attribute value. The refactoring to replace conditionals by polymorphism can be used to transform that single class into a class hierarchy in which the subclass and not an attribute determines the behavior.

When practicing TDD, the abstraction emerges from the need to decouple dependencies for testing purposes. On the other hand, the inheritance usually appears through refactoring, both when classes of the same abstraction share some behavior or when a single class with several behaviors can be split in several subclasses. This approach helps to avoid the inheritance over engineering, which can be detected when you find a superclass with only one subclass. Following this, the inheritance will only arise when the superclass really represent an abstraction of different concepts in the application domain.

Summary of recommendations:

- Define dependencies operations based on abstractions.

- Refactor towards inheritance when several implementations of an abstraction share the same behavior.
- Extract new hierarchies by refactoring conditionals with polimorphism where it is appropriate.
- Do not think of using inheritance before it is needed.

5 Large Refactorings

The implementation of the simplest solution for a given test suite is a very important practice on TDD. The main goal of this practice, as stated before in this chapter, is to encourage the simplicity of the solutions and the focus on the current requirements. However, this approach has a drawback, which is that it can generate situations where the current solution is not suitable for the next requirements.

One can argument that an upfront design can prevent this situation by considering future and potential requirements on the design, based on a detailed requirements elicitation method. On the other hand, most of these predicted requirements may never happen, or, when they happen, they should be probably different from foreseen ones. TDD believes that the time to refactor the software when needed should demand less effort than the implementation of all foreseen requirements. Another impact from this approach is that keeping the code simple, makes it is easier to maintain and evolve.

A large refactoring is often seen as a consequence of a mistake or a wrong design. However, it is normal to happen when practicing TDD as a design technique. It is a direct consequence of the focus on the simplest solution. This refactoring may be not performed on the same TDD session. They may be necessary on future iterations, due to the evolution or change on the requirements. When the large refactoring is needed, it is not precise to say that the design was wrong, but that ”it was suitable for the previous requirements” and ”unsuitable for the current ones”.

In a design created with TDD, classes are usually decoupled, which usually makes changes isolated inside specific classes. However, there are some situations in which the refactoring is large, usually involving things that affect several parts of the source code. Inside a class, the largest changes often involve modification on the data structure, since several parts of methods execution may depend on it. When more classes are involved, changes are usually on the classes API that are used by other classes.

In a recent paper [Guerra(2014)], the author reported a big refactoring that happened on the development of a framework by using TDD. In this refactoring, due to a change in requirements, a sequence of calls that are processed as soon as they were received, now needed to be stored and processed when the sequence is finished. In this situation, it was necessary a change in the data structure and on how it was processed by the class. However, it was restricted to that class.

As recommended on the literature [Fowler(1999), Kerievsky(2004)], refactorings should be performed step by step, executing the tests after each one is performed.

Finding a refactoring path where the behavior is preserved between the steps can be hard. In order to achieve that, sometimes it is necessary to keep both old solution and new solution working at the same time, to remove the old structure only at the end.

Summary of recommendations:

- You can not avoid large refactorings when you give way to the simplest solution.
- Try to isolate possible refactoring points by modularizing it from the rest of the software.
- Perform large refactorings in small steps, running the tests after each one.

6 Combining TDD with Other Design Techniques

Despite TDD can be a consistent technique for software design, it does not target all existing design domains needed for an application. In other words, TDD is not enough as a design technique for all the needs of a software project. This section highlights the context where TDD is applicable and presents design domains that are out of TDD scope. It also presents how TDD can be combined with other design techniques and patterns to achieve other application modeling goals.

Based on what was presented on the previous sections, it is possible to conclude that the main focus of TDD is on API design. By creating a test that uses a class in order to execute some of its behavior, the main design effort is to define an API that is appropriate for that class and for that test scenario. The refactoring, which is a step on its basic cycle, and the incremental development also contributes to the internal class design, which is continuously refined during the process.

It is correct to state that TDD is a technique whose focus is on a single class or in a small group of classes. Despite TDD can be applied in the development of a entire component, the definition of the design through the test happens mainly on the class that is the test target, which is usually the facade to the component functionality. Even when mock objects are used to define the dependences API, the main focus is on the collaborations needed by the class being tested, and not on the dependency itself.

Since TDD is a design technique that focuses on a small group of classes, it is not suitable for design domains where the set of classes and their relationships are more important. The next subsections presents some design domains out of TDD scope and how TDD can be combined with other technique to reach a suitable design for a software application.

Summary of recommendations:

- Understand the role of TDD for design in the context of the project that is being used.
- Combine TDD with other design techniques with different scopes.
- Do not rely on TDD for all aspects of application design.

6.1 Architectural Design

Software architecture cares about the general structure of the application, its layers and its constraints. For the architecture, it is also important to provide a structure that achieves the suitable quality attributes, such as performance and capacity, in order to fulfill the application requirements. The consistency on how the functionalities of software are implemented is an important characteristic of software architecture.

This general design of an application is out of TDD scope, since it does not care about the whole, but the focus is on a single class. Designing the application with TDD, but without architecture as a start point, can lead to bad consequences in a long term. Despite the classes can be decoupled by using mock objects; the functionalities can be implemented with different approaches and by using distinct structures, hurting the design consistency. Following this approach, the application structure can be lead to chaos, made by a net of decoupled but inconsistent components.

There can be reports of software projects that used TDD successfully without an explicit architectural design. However, these applications usually follow well-defined reference architecture, such as Java EE or Ruby on Rails, in which the main component types, their roles and their relationships are already defined and clear to all developers [Fairbanks(2010)]. By following reference architecture, it is possible to keep the design consistency by test-driven developing the architectural components following their roles and constraints.

On the other hand, after an initial architecture definition, the component details, their APIs and their relationships can be designing individually by TDD. While the architectural design cares about how all the components fit together to achieve the application requirements, TDD can pick a single component in this context and work with more detail in its individual design.

In this context, the architectural information about the class that is being developed by using TDD is crucial for its design and development. The class role in the architecture is important in order to verify on the tests only the responsibilities that are assigned to it [Guerra et al(2014)Guerra, Aniche, Gerosa, and Yoder]. If the architecture defines that a type of component should collaborate with other in order to achieve some functionality, a mock object can be used on the test to explicitly define this separation of concerns. In other words, the class development and design will be driven by the tests. However, the tests will be created based on the existing constraints for its role on the architecture.

Summary of recommendations:

- Use references for an initial architectural design.
- Consider architectural constraints of a class on TDD considering its role on the architecture.
- Consider using mocks to define low level details on the interaction of two explicit architectural components.
- Evolve the architectural design through the project.

6.2 Domain Modeling

The domain modeling is a very important part of a software design. This model captures the knowledge from the domain, including the main concepts and their relationships. It is also part of the domain model the division of responsibilities among the domain classes, and how to collaborate on business scenarios to achieve the desired functionality. This kind of modeling translates the business concepts and processes into the software, representing on it the portion of the real world that is relevant for the application functionality.

With the arise of strong reference architectures, such as Java 2 EE [Alur et al(2001)Alur, Malks, and Crupi], with the respective best practices and patterns [Fowler(2002)], the domain modeling was set aside at the expense of an emphasis on architectural modeling. This kind of practice, especially common on enterprise architectures, had bad consequences on applications that end up with a poor domain model, where domain classes are used only as value objects, in other words, to transport data between the database and the graphical interface.

With a poor domain model, the business rules are usually implemented on service classes or, worst, on controllers. That practice causes an overload of functionality on the service classes, duplication of functionality, and feature envy, since the logic that handles the data are defined in a separated class.

In this scenario, the Domain Driven Design (DDD) [Evans(2003)] was a design technique that brought back the focus to the domain modeling, highlighting its importance in the context of a software project. This technique, that is not the only one that can be used for domain modeling, presents patterns on how to identify the domain entities and how this structure should interact with the rest of the application. The usage of DDD are popular on agile teams, even being combined with TDD [Landre et al(2007)Landre, Wesenberg, and Olmheim].

The domain modeling is also out of the scope of TDD as a design technique. According to the first TDD references [Beck(2002), Astels(2003)] and patterns documented about it [Guerra et al(2014)Guerra, Aniche, Gerosa, and Yoder], a list of test scenarios should be used as a reference to start the TDD section. Based on that, a reference from the application domain and class responsibilities is important to build this functionality list. In other words, it assumes the existence of a previous domain analysis.

The eXtreme Programming [Beck(2004)] was the first agile methodology to propose Test Driven Development as part of its set of practices. However, it also proposes the usage of CRC cards in order to enable a collaborative discussion about the application domain among the developers and the stakeholders. XP proposes the combination of TDD and CRC for the application design. In this context, CRC is used for a more abstract modeling about the classes, their collaborations and responsibilities, and after that definition TDD is used to develop each class modeling its API and refining it continuously.

TDD works fine with other design techniques for domain modeling. The domain modeling usually comes first by identifying the domain classes, their characteristics, collaborations and responsibilities. Than, when TDD is used to develop that class, or a subset of its functionality, it uses the domain model as a reference. For instance, collaboration can be modeled defining a mock object and delegating part of the behavior for it.

Summary of recommendations:

- Complement TDD with a technique for domain design, to have a broader view of the application domain.
- Create tests on a TDD session based on the domain class responsibilities.
- Create mock objects based on expected domain class colaborations.

6.3 Design Patterns

Patterns can be considered a recurrent solution used to solve a problem in a given context [ale(1977)]. The design patterns [Gamma et al(1995)Gamma, Helm, Johnson, and Vlissides] document solutions in the domain of object-oriented software design, providing several solutions that can be used to solve design problems. Each pattern contains forces and consequences that present the tradeoffs of adopting such solution.

Design patterns are an important design knowledge base that can be used independent of the design technique adopted. If the design is performed with diagrams, the pattern solution can be expressed on them. Similarly, the same pattern can also be expressed by using the test, by defining an API suitable with the pattern implementation and mock objects consistent with the pattern collaborations. While on TDD the test is used to drive the implementation and design, the patterns can be used to define the direction of that guidance.

A recent article reported the usage of TDD to perform the design of a persistence framework [Guerra(2014)]. In this framework, the pattern Visitor was adopted as a design solution to decouple the query definition from the query generation for a specific database. According to the paper, the Visitor implementation was driven by the tests, which pushed the code in the direction of the pattern. However, the author also states that the pattern knowledge was crucial to its adoption, and hardly the same solution would be adopted if it were not known.

On the context of TDD, another approach for implementing patterns is through refactoring [Kerievsky(2004)]. One of the big advantages on applying patterns by refactoring is that you do not need to add it until it is really necessary, which can help to avoid over engineering. Usually, the patterns that are applied following this approach are usually to accommodate a crescent software structure, enabling the management of a large number of classes and behaviors.

Summary of recommendations:

- Consider design patterns in the definition of the class interface in the test.
- Use patterns as targets to refactoring to solve design problems.
- Use the expected pattern relation to define mock behavior of a dependence.

7 Preparing for TDD in a Software Project

An important message from this chapter is that TDD is not enough to design an entire application. It is not viable to pick a set of requirements or user stories and jump straight to creating tests, without having a notion of how the architecture should be and how the domain entities relates to each other [Aniche and Gerosa(2012)]. Sometimes this process is not formal or explicit, but that does not mean it does not exist. This section presents some activities that should be performed before the practice of TDD in order to improve the software quality as a whole.

In order to start a TDD session, the developer should know the scope of that session. In other words, he needs to know what should be developed. To define it, the developer should be aware of the functional requirements s/he needs to implement, what classes should be included in the context of this development, and the constraints based on the class role in the architecture. It is important to define these things with the team, either formally defining them in a document or by an informal discussion. Leaving these decisions for each developer can generate a lack of standardization in similar classes or layers, leading the production and test code to an unorganized structure.

To be able to understand the constraints of the class that is being developed, an envisioning of the architecture is necessary. The reference architectures play an important role on this issue, because they often can be used as this starting point for the architecture [Fairbanks(2010)]. Based on them, the developers can start the development with a set of component types with their roles and constraints, without a formal architectural document. With more challenging requirements or when working to an application domain that is not well established in industry, an initial project that present solution proposals for the main architectural problems is advisable. By using this architectural definition as an initial reference, the team can refine and evolve it through the iterations.

To exemplify how the architectural constraints are considered in the TDD development, consider the development of a class from the business layer on a typical

web-based information system. As constraints, the architectural definition states that it cannot access the database directly and that it should not retrieve information from the user session. Based on that, to develop this class by using TDD, the test needs to consider that this class needs collaboration from a class to persist an information. The API of the class being developed should also consider that if it needs information present on user session, it should be received somehow (for instance, as a method or constructor parameter).

Besides the architectural constraints, the TDD and testing approach that should be used to develop the target class should also be defined. Although the initial TDD sessions can be used to explore the possibilities, in long term it is important to use the same approach to develop similar classes. For instance, it is desirable that classes from the same layer are developed using the same testing technique.

On a TDD session there are two scopes that should be defined: the test scope, and the development scope. The test scope includes which classes are verified by the tests and the development scope is the classes that are developed on the TDD session. They can be the same, for instance when the development uses unit tests, however existing classes may be involved on the tests but are not the target of that development session.

The standardization of how TDD should be used on each type of component of the application architecture is important for the success of TDD as a design technique in the context of the entire software. This process can help to define the scope of tests and TDD development for the application layers and the tools and frameworks that are going to be used in each one. Even more specific things, such as helper test classes [Meszaros(2006)] and test superclasses [Guerra and Kinoshita(2012)], can be developed to help on test code development. This practice is complimentary to the architecture definition, defining how each architectural component should be tested.

Even though TDD can be used as a personal development practice, for it to be effective on the context of a project as a design technique, the whole team should be aware of the application architecture and the TDD and test approach on its context. By having this preparation for TDD on the project, the team member can share the same understanding on how the software should be designed, enabling a synergy on the system evolution.

For instance, to use different tools and testing approaches to develop two components of the same layer, it would be like on a more traditional design approach to use different types of diagrams to model the same kind of behavior. This team shared understanding to talk the same design language when performing TDD enables them to work together on the software evolution, even when working on separated artifacts.

Summary of recommendations:

- Define the scope of the tests and of the TDD session before its begining.

- Consider class constraints and known collaborations based on architecture design or domain model.
- Use the same test approach on TDD for the same kind of component.
- Encorage the reuse of common test routines to make test easier on further TDD sessions.

8 Continous Inspection

Besides TDD have refactoring as part of its cycle, that fact does not guarantee that the developers perceive all the problems in source code. Especially when the sessions use unit tests to develop a single class, it is hard to perceive a design problem with a broader scope. Because of that, the practice of continuous inspection [Merson et al(2013)Merson, Yoder, Guerra, and Aguiar] is important to provide a continuous feedback on the application design, allowing the detection of problems untimely, drawing the attention of the developers to that part of the source code.

The practice of continuous inspection uses static and dynamic analysis tools to retrieve information about important quality attributes from the source code, such as test coverage, complexity and decoupling. SonarQube and Code Climate are examples of such tools. These tools are frequently executed on each source code version and the result of its execution is reported to the developers. Based on the information provided, the developers can draw their attention to a piece of code with a potential problem and consider its refactoring. Additionally, in the context of several project iterations, it is possible to perceive how the code is evolving, making possible to plan larger refactorings to the next iterations.

The feedback of the continuous inspection tools can happen at three different timings. The fastest one is in the developer IDE, showing warnings and errors based on the source code analysis, allowing the developer to rethink his current decisions. The second moment is at compile time on the developers' machine, when a broader analysis can be performed considering the entire application and even other components. Finally, the third moment is in a continuous integration server, when more complex and long analysis can be performed, even by executing the application to retrieve dynamic metrics.

There are several types of analysis that can be performed in the continuous inspection process. The most important ones based on the requirements should be chosen by the team, and can also be evolved based on the retrospectives. The quality analysis can retrieve metrics from the source code and compare the numbers to industry standards based on statistical thresholds [Lanza et al(2005)Lanza, Marinescu, and Ducasse]. Additional information, such as bad smells and bug detection can also be executed. Architectural conformance [Merson(2013)] and security verifications are other types of static analysis that can be executed on the source code. Additionally, dynamic analysis can be performed to measure other quality attributes, such as test coverage and performance.

Besides tools can give an important feedback on quality attributes related to the source code, having the developers taking a critical looking at the project artifacts can also raise some important issues that were not detected by the tools. Because of that, it is important to have a time to stop and discuss the situation of the source code based on the tools report and the developers' point of view. The iteration retrospectives are a nice place to have this discussion, but the team can define when it should take place and the frequency that it should happen.

The practice of continuous inspection has the same value of evolutionary design as TDD. Since TDD introduces a simple design at first to evolve it through the iterations, the inspection reports can help to identify points where the design should evolve at the time that this need appears. In this sense, it complements TDD having a broader focus on the application design as a whole, covering that weak point of TDD to focus on the design of a small number of classes at a time.

Summary of recommendations:

- Use the tools that focus on important aspects needed by the team.
- Evolve the Continous Inspection process and verifications based on the team feedback and learning.
- Discuss on retrospectives actions based on the tools inspection.
- Encorage developers to give additional feedback.

9 Conclusions

TDD is a design and development technique that can have an important role in a software project. It brings important design values to the team, but its main focus is on the class' API and on its internal structure, which is far from being enough. The main weak point of TDD from the design perspective is that it does not consider the software as a whole and only looks at a small part of it. This chapter's goal was to present how TDD can be combined with other techniques to contribute to the software quality in a long term.

Initially, this chapter presents TDD as a design technique, raising evidence from existing works about its contribution to software quality. After that it explained what are the design values adopted on TDD and how they are applied in practice based on the technique dynamics. The following section presented the mock objects and how TDD design class and object relations; which is an important point on an object-oriented design. This presentation of TDD finishes with a section that presents why big refactorings are part of a normal TDD process if adopted in a software project.

After that, the chapter presented some design domains that are out of TDD scope, and how the use of other techniques can by combined with TDD. It was presented how a software project should be prepared for the TDD can be used as a team practice and to have a positive impact on software quality. Finally, continuous inspection

is presented as a complimentary technique to enable the team to always keep an eye on how the design is evolving through the iterations.

As a final conclusion, it is important to recognize that TDD has a limited scope and should be combined with other techniques to be used as a team practice in a long-term software project. The weight for the entire application design should not be responsibility of a single technique. If the support techniques for TDD are applied appropriately, it has a high potential to have a great impact on the software quality.

References

[ale(1977)] (1977) A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure), later printing edn. Oxford University Press, URL http://downlode.org/etext/patterns/

[Alur et al(2001)Alur, Malks, and Crupi] Alur D, Malks D, Crupi J (2001) Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall PTR, Upper Saddle River, NJ, USA

[Aniche et al(2011)Aniche, Ferreira, and Gerosa] Aniche M, Ferreira T, Gerosa M (2011) What concerns beginner test-driven development practitioners: A qualitative analysis of opinions in an agile conference. 2o Workshop Brasileiro de Métodos Ágeis (WBMA)

[Aniche and Gerosa(2012)] Aniche MF, Gerosa MA (2012) How the practice of tdd influences class design in object-oriented systems: Patterns of unit tests feedback. In: Software Engineering (SBES), 2012 26th Brazilian Symposium on, IEEE, pp 1–10

[Astels(2003)] Astels D (2003) Test-Driven Development: A Practical Guide, segunda edn. Prentice Hall

[Beck(2002)] Beck K (2002) Test-Driven Development By Example, 1st edn. Addison-Wesley Professional

[Beck(2004)] Beck K (2004) Extreme Programming Explained, 2nd edn. Addison-Wesley Professional

[Dogsa and Batic(2011)] Dogsa T, Batic D (2011) The effectiveness of test-driven development: an industrial case study. Software Quality Journal pp 1–19, 10.1007/s11219-011-9130-2

[Erdogmus et al(2005)Erdogmus, Morisio, and Torchiano] Erdogmus H, Morisio M, Torchiano M (2005) On the effectiveness of the test-first approach to programming. IEEE Transactions on Software Engineering 31:226–237, DOI http://doi.ieeecomputersociety.org/10.1109/TSE.2005.37

[Evans(2003)] Evans (2003) Domain-Driven Design: Tacking Complexity In the Heart of Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

[Fairbanks(2010)] Fairbanks G (2010) Just Enough Software Architecture: A Risk-Driven Approach. Marshall and Brainerd

[Fowler(1999)] Fowler M (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

[Fowler(2002)] Fowler M (2002) Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

[Fowler(2007)] Fowler M (2007) Mocks aren't stubs. http://martinfowler.com/articles/mocksArentStubs, Último acesso em 26/11/2014

[Freeman et al(2004)Freeman, Mackinnon, Pryce, and Walnes] Freeman S, Mackinnon T, Pryce N, Walnes J (2004) Mock roles, objects. In: Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, ACM, New York, NY, USA, OOPSLA '04, pp 236–246, DOI 10.1145/1028664.1028765, URL http://doi.acm.org/10.1145/1028664.1028765

[Gamma et al(1995)Gamma, Helm, Johnson, and Vlissides] Gamma E, Helm R, Johnson R, Vlissides J (1995) Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

[George and Williams(2003)] George B, Williams L (2003) An initial investigation of test driven development in industry. In: Proceedings of the 2003 ACM symposium on Applied computing, ACM, New York, NY, USA, SAC '03, pp 1135–1139, DOI http://doi.acm.org/10.1145/952532.952753, URL http://doi.acm.org/10.1145/952532.952753

[Guerra(2014)] Guerra E (2014) Designing a framework with test-driven development: A journey. Software, IEEE 31(1):9–14, DOI 10.1109/MS.2014.3

[Guerra and Kinoshita(2012)] Guerra EM, Kinoshita B (2012) Patterns for introducing a superclass for test classes. In: Proceedings of the 9th Latin American Conference on Pattern Languages of Programming, ACM, New York, NY, USA, SugarLoafPLoP '12

[Guerra et al(2013)Guerra, Yoder, Aniche, and Gerosa] Guerra EM, Yoder J, Aniche M, Gerosa MA (2013) Test-driven development step patterns for handling objects dependencies. In: Proceedings of the 20th Conference on Pattern Languages of Programs, ACM, New York, NY, USA, PLoP '13

[Guerra et al(2014)Guerra, Aniche, Gerosa, and Yoder] Guerra EM, Aniche M, Gerosa MA, Yoder J (2014) Patterns for preparing for a test driven development session. In: Proceedings of the 21th Conference on Pattern Languages of Programs, ACM, New York, NY, USA, PLoP '14

[Janzen and Saiedian(2006)] Janzen D, Saiedian H (2006) On the influence of test-driven development on software design. Proceedings of the 19th Conference on Software Engineering Education and Training (CSEET'06) pp 141–148

[Janzen(2005)] Janzen DS (2005) Software architecture improvement through test-driven development. In: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM, New York, NY, USA, OOPSLA '05, pp 240–241, DOI http://doi.acm.org/10.1145/1094855.1094954, URL http://doi.acm.org/10.1145/1094855.1094954

[Kerievsky(2004)] Kerievsky J (2004) Refactoring to Patterns. Pearson Higher Education

[Landre et al(2007)Landre, Wesenberg, and Olmheim] Landre E, Wesenberg H, Olmheim J (2007) Agile enterprise software development using domain-driven design and test first. In: Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion, ACM, New York, NY, USA, OOPSLA '07, pp 983–993, DOI 10.1145/1297846.1297967, URL http://doi.acm.org/10.1145/1297846.1297967

[Langr(2001)] Langr J (2001) Evolution of test and code via test-first design. http://www.objectmentor.com, Último acesso em 01/03/2011

[Lanza et al(2005)Lanza, Marinescu, and Ducasse] Lanza M, Marinescu R, Ducasse S (2005) Object-Oriented Metrics in Practice. Springer-Verlag New York, Inc., Secaucus, NJ, USA

[Li(2009)] Li AL (2009) Understanding the efficacy of test driven development. Master's thesis, Auckland University of Technology

[Mackinnon T.(2001)] Mackinnon T CP Freeman S (2001) Endotesting: unit testing with mock objects. In: Succi G, Marchesi M (eds) Extreme Programming Examined, Addison-Wesley Longman Publishing Co., pp 287–301

[Madeyski(2006)] Madeyski L (2006) The impact of pair programming and test-driven development on package dependencies in object-oriented design - an experiment. In: Munch J, Vierimaa M (eds) Product-Focused Software Process Improvement, Lecture Notes in Computer Science, vol 4034, Springer Berlin / Heidelberg, pp 278–289

[Martin(2002)] Martin RC (2002) Agile Software Development, Principles, Patterns, and Practices, primeira edn. Prentice Hall

[Merson(2013)] Merson P (2013) Ultimate architecture enforcement: custom checks enforced at code-commit time. In: Hosking AL, Eugster PT (eds) SPLASH (Companion Volume), ACM, pp 153–160, URL http://dblp.uni-trier.de/db/conf/oopsla/splash2013c.html#Merson13

[Merson et al(2013)Merson, Yoder, Guerra, and Aguiar] Merson P, Yoder J, Guerra E, Aguiar A (2013) Continuous inspection - a pattern for keeping your code healthy and aligned to the architecture. In: Proceedings of the 3rd Asian Conference on Pattern Languages of Programs, ACM, New York, NY, USA, AsianPLoP '13

[Meszaros(2006)] Meszaros G (2006) XUnit Test Patterns: Refactoring Test Code. Prentice Hall PTR, Upper Saddle River, NJ, USA

[Muller and Hagner(2002)] Muller M, Hagner O (2002) Experiment about test-first programming. Software, IEEE Proceedings - 149(5):131 – 136, DOI 10.1049/ip-sen:20020540

[e Nat Pryce(2009)] e Nat Pryce SF (2009) Growing Object-Oriented Software, Guided by Tests, 1st edn. Addison-Wesley Professional

[Siniaalto and Abrahamsson(2008)] Siniaalto M, Abrahamsson P (2008)

[Steinberg(2001)] Steinberg DH (2001) The effect of unit tests on entry points, coupling and cohesion in an introductory java programming course. XP Universe