Test-Driven Development Step Patterns For Designing Objects Dependencies

EDUARDO GUERRA, National Institute for Space Research, Brazil
JOSEPH YODER, Refactory Inc., USA
MAURÍCIO FINAVARO ANICHE, University of São Paulo, Brazil
MARCO AURÉLIO GEROSA, University of São Paulo, Brazil

Test-driven development (TDD) is a development technique often used to design classes in a software system by creating tests before their actual code. The dependency management and class APIs decisions, that emerge during the practice of TDD, does not "just happen": the way that the tests are created should be used in this process to make decisions and drive the design in the desired direction. This paper introduces four patterns that document the kinds of TDD cycles that can be performed to guide the design in the desired direction. These patterns are part of a pattern language that intends to present recurrent solutions that are used in a TDD process.

Categories and Subject Descriptors: **D.1.5 [Programming Techniques]**: Object-oriented Programming; **D.2.11 [Software Architectures]**: Patterns

General Terms: Test driven development

Additional Key Words and Phrases: TDD, software design, patterns

ACM Reference Format:

Guerra, E., Yoder, J., Aniche, M. and Gerosa, M.. 2013. Test-Driven Development Step Patterns For Designing Objects Dependencies. Proceedings of the 20th Conference on Pattern Languages of Programs (PLoP). October 2013, 15 pages.

1. INTRODUCTION

Test-driven development (TDD) is a technique in which the tests are written before the production code (Beck 2002). By using it, the development occurs in cycles, comprised of the creation of an automated test, an update on the developed software to make the test pass, and a code refactoring to improve the solution. TDD can be used for many different goals: as a testing technique, in which developers expect an improvement in the external quality; or as a design technique, in which developers expect to improve class design (Beck, 2002; Martin, 2006; Astels, 2003; Freeman, 2009). **Patterns documented in this paper consider the usage of TDD as a design technique.**

The terminology used by the TDD community uses the metaphor “baby steps”. It refers to the fact that by using this technique the development advances continuously in small steps. These patterns borrow the word “steps”, referring to actions that make the system development and design to move forward. The goal is to take small steps towards a better design. In most of the patterns, a “step” refers to a TDD cycle, however it is not true in all cases. For instance, the patterns **Dive Deep** and **Pause for Housekeeping** are steps that should happen between TDD cycles.

One of the great challenges when using TDD as a design technique is modeling class dependencies, including their contracts and expected behavior. A testing technique that is often used to handle dependencies is **Mock objects** (Freeman et al. 2004), which is also documented as a pattern (Brown and Tapolcsanyi 2003; Mezsaros 2007). **Mock objects** are fake objects used to emulate dependencies' behaviors and verify the expected behavior of a tested class based on its method calls. The main goal of a **Mock Object** is to enable the isolation of the behavior of the tested class from its dependencies for testing purposes. There are several frameworks that can be used to define a **Mock Object** behavior and expectation in the test method itself. However, the mock object can be a simple class that has the same abstraction of the class dependency. To do that, the mock should implement the same interface or extend the main class. It is well known that **Mock Objects** are a broader concept than test stubs (Fowler 2007), and because of that it will be used on this paper

to refer all techniques that replaces dependencies by test doubles. In the examples, we use the JMock framework (Freeman and Pryce 2006).

This paper is part of a study that aims to identify recurrent TDD steps and how they can be used in a TDD session to drive the developed class design in the desired direction. It focuses on TDD step patterns that involve handling and designing class dependencies and relationships. Next section describes briefly the current state of this pattern language and the further sections present the patterns.

2. TDD STEPS PATTERN LANGUAGE

The goal of this pattern language is to document the steps that the developer can take to move forward in a TDD session. Some developers face TDD only as a testing technique, in which the functionality is created piece by piece by creating the tests first. A developer, who is not used to TDD, does not see naturally how these tests can be used as a tool to drive the design in the desired direction. This pattern language aims to explicitly present the steps that can be chosen to move forward classes development.

The target audience of this pattern language is software developers that use TDD to design and develop software. The patterns names form a terminology to reference the alternative steps that developers can perform. It also can present to beginners the mechanics of this design technique and, to practitioners, recurrent solutions that they can employ. The discussions presented in each pattern intend to clarify the consequences of each choice of step.

Instead of being inflexible about the dynamic of a TDD process, this pattern language prefers to present the different existing options, discussing their respective consequences. Some practices documented by these patterns may look like as anti-patterns in the first impression. However, if developers are aware of the consequences and of the other choices, they can be valid paths. Future evolutions and additions for this pattern language may reveal other possible steps that can complement the traditional TDD process.

Figure 1 presents a pattern map with the ones already identified for this pattern language. The idea of this map is to show how to navigate through the patterns according to the scenario faced in the TDD session. The patterns in grey have already been documented in previous studies (Guerra 2013). The ones in black are documented in this paper and the remaining ones, in white, have not been documented yet. The arrows in this diagram represent the paths that you can follow to choose a pattern aiming to move forward in a TDD session.

The following describes briefly each pattern in the language:

- **API Definition**: When you need to introduce a new programming element, such as a class or a method, create a test with the simplest scenario that involves it.
- **Differential Test**: When you want to move forward in the TDD session, add a test that increments a little the functionality verified by the previous tests.
- **Exceptional Limit**: When you have a scenario where the class functionality does not work properly, create a test with that scenario verifying if the class is behaving accordingly to these scenarios.
- **Everything Working Together**: When you have features in the same class that are tested separately, create a more complex test scenario where these features should work together.
- **Bug Locator**: When a bug is found, create a new test that fails because of it. By doing that the developer will be able to detect the location of that bug. Then, the developer should fix the code in order to make this new test to pass.
- **Diving Deep**: When the complexity of an implementation demands the creation of small auxiliary methods or classes, ignore temporarily the current test and start an embedded TDD session to develop this auxiliary code.
- **Pause for Housekeeping**: When the application class needs a huge change to make the current test to pass, ignore temporarily the current test and refactor the production code considering the previous tests.
- **Mock Complexity**: When a test is complicated to create because it depend on an external resource, define an interface that encapsulates the resource interaction and mock it in the test.

- **Dependency Exposure**: When you need to define an API from an explicit dependency of the application class, create a test that creates a Mock Object and define the expected calls to the dependency API.
- **Hide Internal Solution**: When there is no need to change an internal dependency implementation and it has a simple and well-defined role in the class functionality, encapsulate the implementation within the developed class and do not expose the solution to the test class.

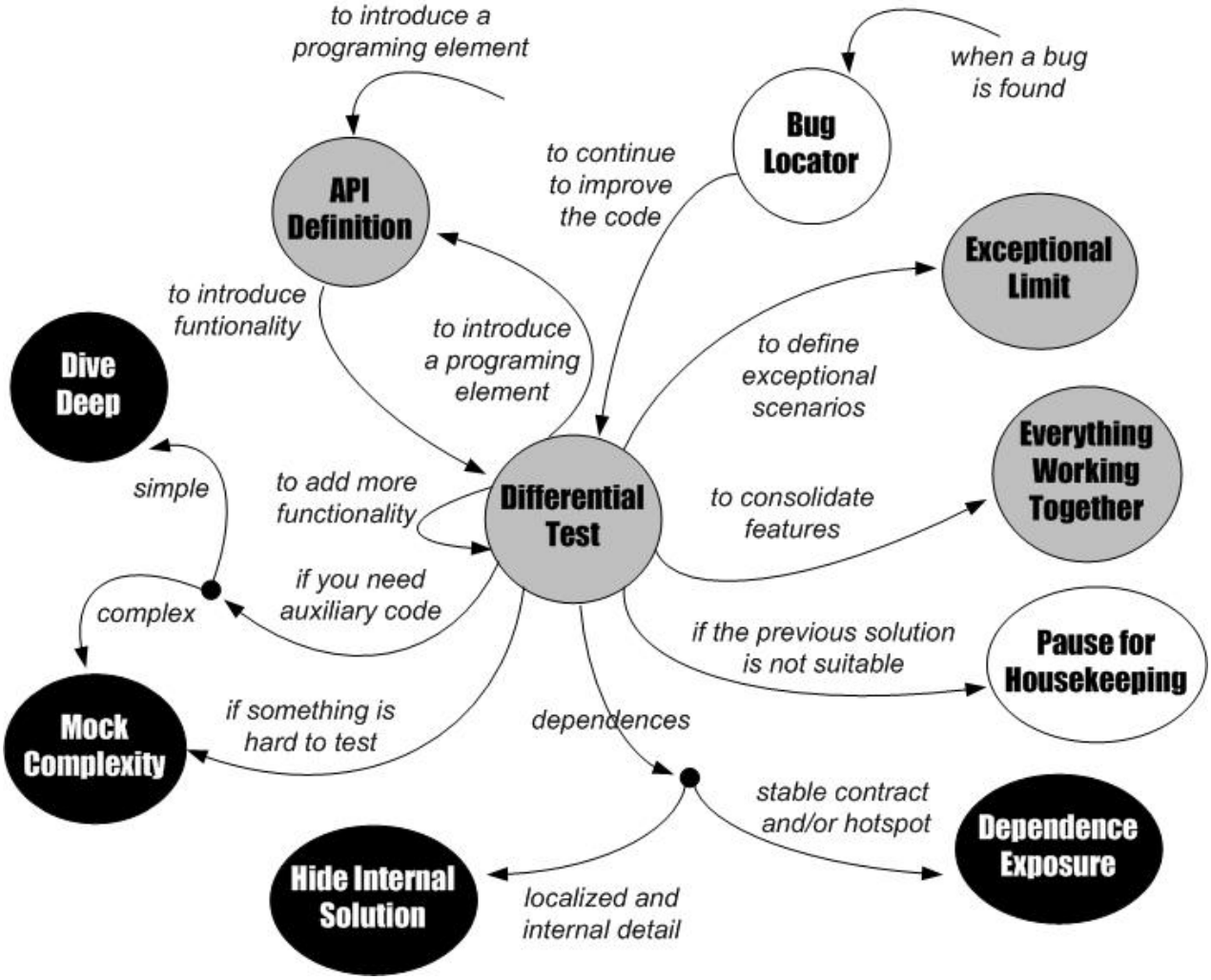


Fig. 1. *TDD Step Patterns Map.*

3. DIVE DEEP

Also known as: Pause-Create-Resume, Embedded TDD Session.

Sometimes you need to dive deeper than the current class in the TDD session to create auxiliary code.

Sometimes when the developer is creating some functionality, he needs auxiliary methods, or even a helper class, to perform a specific and focused task. This situation usually happens when there is a small piece of logic, which is necessary to make the test to pass, but it is not directly related to the responsibilities of the focus of the current TDD session. This auxiliary logic could not be big or complex enough to justify wait for its implementation in a future TDD session. Additionally, this secondary functionality may be useful in the development of other functionalities.

♦ ♦ ♦

What should be the next step in the TDD session when you need helper code to continue the development of the current class?

Developers, sometimes, write tests not only for the main responsibility of the class, but to other small concerns of it. That may be considered as a problem because, when doing that, developers tend to lose focus from the main behavior. As an example, it often happens when developers need to test private helper methods.

If the developed functionality is included in the class being developed, one can extract a method and move it to another class to enable its reuse in other contexts. However, the tests that verify this functionality will remain in the original class test suite. That can make hard to evolve this extracted method using TDD, since its tests will be mixed with other class tests. To solve this issue, it would be necessary to also refactor the test code.

Examples of this kind of helper code often include parsing strings, formatting dates, calculating values, and performing non-trivial tasks in existing APIs. Considering the current class responsibilities, this kind of logic usually does not belong there. The inclusion of this functionality in the current class can break its cohesion, and affect the current TDD session taking out the focus of the main class responsibilities.

One of the principles of TDD is “simplicity”, in a way that when you create a class using TDD, it drives you to a cohesive solution because it is easy to focus on a single concern. When you find a scenario that you need to create tests that focus on concerns that are not class responsibilities, such as if your class is making a calculation correctly or if a string is parsed correctly, it is easy to lose this simplicity.

Therefore:

Skip temporarily the last test added in the current TDD session, and starts a new TDD session to develop the auxiliary class or method. When this development is finished, return to the original class and invoke the developed code in the solution.

When you pause the current TDD session and start a new one, the original class responsibilities do not need to be considered in that moment. In this new TDD session, the developer can focus on its specific requirements and make tests that consider all possibilities. The tests should include as well possible Exceptional Limits that may exist. Then, as soon as the embedded session finishes, the developer should return to the first session.

When this embedded TDD session is finished, and consequently the auxiliary code is tested and developed, the developer can come back to the original session and work to make the last test in the suite, which was temporarily ignored, to pass. At this point, he can use the helper code trusting that its functionality is working correctly. As a consequence, it is not necessary to introduce exhaustive test scenarios that focus on all possibilities of the auxiliary code. The further tests on the original TDD session should now focus on concerns relative to its main responsibilities.

A negative consequence of this pattern is that the developer will lose his focus on the original TDD session, however that can be tolerable if it is a small piece of functionality. Because of that, this practice is not advisable when the auxiliary code that needs to be developed is too complex or too big. However it may not always be possible to accurately foresee it and the developer should use his experience to better determine if he should go for Mock Complexity and continue the TDD session or to apply this pattern.

This pattern can also be applied when the developer does not know how to implement part of the solution. The development of this auxiliary code can be used to search for a spike solution. The tests in the embedded TDD session will help to validate a viable solution. Despite it takes out the focus of the original TDD session, in some cases it is better to explore the solutions away from a more complex class context. If the final code was too small, an inline refactoring can be performed to include it in the original class.

Listing 1. Test of CarOffer where a licence plate is invalid

```
@Test(expected=InvalidCarLicencePlateException.class)
public void carWithABadLicencePlate(){
   Offer o = new CarOffer();
   Car c = new Car("Ferrari",2005,"AAAAA");
   o.addProduct(c);
}
```

When the developer was creating the solution, he realized that several scenarios about the licence plate validity should be considered. This logic does not fit on the CarOffer responsibilities Based on that, he decided to Dive Deep in the implementation of a class responsible to validate licence plates. Listing 2 presents the first test of this embedded TDD session.

Listing 2. A test of LicencePlateValidator

```
@Test
public void fiveLettersLicencePlate(){
   LicencePlateValidator v = new LicencePlateValidator()
   assertFalse(v.validate("AAAAA"));
}
```

This new TDD session focusses only on this logic, and can explore diverse scenarios without other concerns. When this session is finished, the original session can be resumed and the solution can use the class LicencePlateValidator. Since this class was already tested in its own test class, the original TDD session needs only to verify a scenario when it returns true and another when it returns false.

♦ ♦ ♦

This pattern is often necessary during a **Differential Test** that is introduced in a main TDD session. The first tests on the embedded TDD session should start with an **API Definition** that considers the necessity of the class that is being developed in the current session. Additional parameters can be further introduced through

some refactoring. They can be motivated by new or changing requirements, in order to allow the reuse of this code in other contexts.

The change of focus can be used to explore greater number of scenarios in the auxiliary functionality, including its **Exceptional Limits**. The developer should **Hide Internal Solution** when using this pattern because, since the auxiliary class solves an internal and specific problem, it doesn't need to be exposed externally.

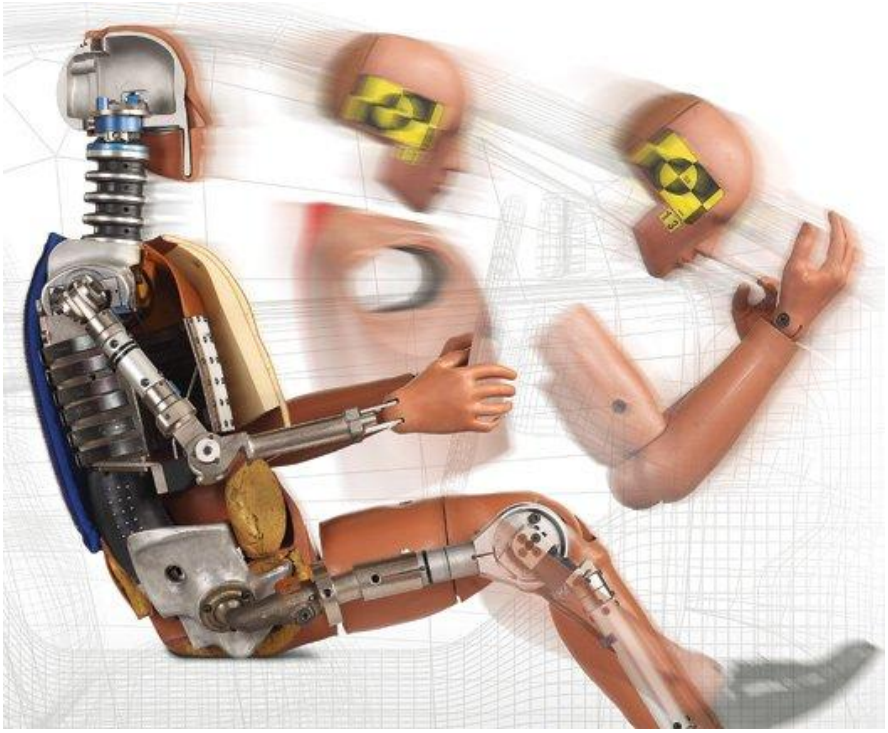
When the developer needs to create complex or long helper functionality and does not want to focus on the current TDD session at the moment, an alternative is to use a **Mock Complexity**. By using this pattern, a mock is created to simulate the complex auxiliary logic and the dependency can be developed further

On Esfinge QueryBuilder (Esfinge 2012), a framework for persistence, this pattern was often used to develop auxiliary methods to handle complex functionality about reflection and code annotations. Despite the complexity, the fact that these methods could also be used for other functionality also motivated the usage of this pattern. For instance, the framework considers that a property can be annotated on the attribute or in one of its accessor methods. When this requirement was necessary, the development ***Dove Deep*** *to create a static method that returns annotation wherever it was configured. After that, the development of the original class was resumed, and each test only considered one approach for annotation configuration.*

Alura (Caelum, 2011) is an online e-learning system. During the development of the payment process, we noticed that a Payment could be created for different reasons: because it was paid, it was given for free; it was given because the user has annual subscription, and so on. When we were developing this specific part, we created many static methods that worked as factory methods. We then used the ***Dive Deep*** *pattern to create them, as the implementation would be simple and would help the rest of development.*

4. MOCK COMPLEXITY

Also known as: Complexity Simulation.



Dummies are used to substitute humans on crash tests because real persons can get hurt. They are built to simulate a real person and to measure the impact results.

To start the development of a class using TDD, it is necessary to create its automated tests. An obstacle found in the development of some classes comes from the fact that the creation of their tests is a complicated task.

These scenarios are usually related to external dependencies, such as hardware interaction or external systems access. Other examples are random functions, time-related logic, and even classes with a complex logic. The concept of what is complex to test may depend on the software architecture and the tools available to create such test.

♦ ♦ ♦

How to proceed on a TDD session when the class that is being developed depends on a resource that is hard to test?

When a developer implements a class using TDD, the tests should focus on the class main responsibilities. However, a class usually interacts with different other classes, or even with external resources, which makes it hard to create an automated test. This kind of scenario, which is not rare, can put an obstacle in the creation of automated tests, and consequently in the adoption of TDD as a design and development technique.

An example of an external resource access is hardware interaction. To automate such test, first of all it is necessary to configure and install the hardware on every machine that will run the tests. Furthermore, to perform the test, the tested class should access the hardware, and its state accessed and verified by the tests. That can be unfeasible depending on the hardware price and on the cost to actually access the hardware in every test.

Another example is external systems that should interact with the application. The environment configuration can be the first problem, because the external system should be executing before the class access. Sometimes the system is not easily accessible for testing. Tests that involve external systems should be able to verify the effects that the class invocation provoked on the other system, which is usually a hard task.

Even the use of application classes which implements complex logic can be hard to test. It can be hard to find the right parameters to generate the desired result to be used for tests. Another example are application classes whose behavior depends of the current time, or does not have a deterministic behavior.

When testing a single class, developers should focus on the main responsibility of it, and not on the behavior of its dependencies. So, it is desirable to test the main class behavior without dealing with the complexity of the rest of the system.

Therefore:

Create an interface that define methods that represent the services of the complicated resource and create a mock object that simulate its behavior. Then, introduce this mock object on the application class and use it in the current TDD session.

By creating a mock object to emulate the logic that is hard to test, the TDD session can proceed without that concern. The mock object should simulate different scenarios, by having different results of its execution, to allow the class testing. It also should be able to verify if the received calls are according to the requirements.

Using this solution, the developed class is decoupled from the external API used to handle the external resource. By making the design simpler for testing, the developer is also adding a desired property to the application design. As a consequence, the class can accept different implementations of the mocked services, becoming more flexible.

Even when the external API provides a set of interfaces to mock, it can be a good practice to create your application's own interface to abstract those services. Usually the existing API has a more general functionality, which is much more than your application needs. Because of that, it may be necessary to mock more classes and more methods, making the act of mocking a hard task to perform. Following this recommendation, the access to the original API would be encapsulated in a class that implements the new interface.

By using this pattern, the code that is hard to test is isolated and is not addressed in the current TDD session. However, the code to access the external resource should be developed sometime and the developer will face this problem again. There are some approaches that can be adopted to handle this code. If the team decides to automate the tests of the functionally mocked, at least it needs to be done only once and without being mixed with application business rules. Other alternatives include the manual test of this class and its test integrated with the entire application.

Consider that a class needs to notify an external application by sending a message to a message broker. More specifically, this class authenticates the users and sends a message in every failed attempt. To test it in the traditional way, the test should access the message server to verify if the message was sent correctly. Additionally, before starting the test, it should run the server and make sure that it has the appropriate configuration.

To avoid handling the server in the test, the service to send the message was abstracted in an interface. This interface is presented in Listing 3. Despite the messages API in Java (JMS) is based on interfaces, it is hard to be mocked since it demands the creation of several classes and the invocation of several methods to send a single message

Listing 3. Definition of an interface to send messages

```
public interface MessageSender{
   public void send(Object msg);
}
```

Listing 4 presents the test that verifies the scenario of a failed authentication attempt of the class named Authenticator. The test creates the Authenticator instance, creates a mock based on the interface using JMock framework and set this mock on the tested class. Furthermore, it introduces a mock expectation that defines that the method send() should be invoked once on it with the parameter "Login Failure: admin". To finish the test, it executes the method login() that is the one that actually exercise the functionality to be tested.

Listing 4. Creating a test that mocks the interface to the class Authenticator

```
public class AuthenticatorTests {

   @Rule public JUnitRuleMockery ctx = new JUnitRuleMockery();

   @Test
   public void loginFailureNotification(){
      Authenticator auth = new Authenticator();
      MessageSender mock = ctx.mock(MessageSender.class);
      auth.setSender(mock);

      ctx.checking(new Expectations() {{
         oneOf(mock).send("Login failure: admin");
      }});

      auth.login("admin", "wrong");
   }
}
```

By analyzing the example, it is possible to see that by creating the interface MessageSender initially for testing purposes, the class Authenticator was decoupled from the API used to actually send the message. As a consequence, new implementations of MessageSender can be created and other approaches for sending messages can be easily incorporated in the future.

♦ ♦ ♦

This pattern can be considered a kind of **API Definition**, however it is the API of the class being mocked that is being defined. This kind of step can happen in the middle of a sequence of Differential Tests, when a new feature demands an interaction with a class or resource that complicates the automation of its testing.

In the system SADE (Perillo et. al 2011), responsible for handling emergency calls on a brazilian state, the component responsible to send reverse AJAX notifications was developed using TDD, where this pattern was applied. An interface was created to abstract the service to send the notifications. In the class development, which focus on when and to whom it should send the notifications, this interface was mocked. Later, it was created a

class that implements this interface and is responsible to interact with DWR (2013) framework to actually send the notifications. This functionality was tested manually as part of the functional tests.

MetricMiner (Sokol et. al 2013) is a system that helps researchers on mining software data to extract information about the source code and team members. The tool consumes data from source code management tools (SCM), such as Git and SVN. When developing it using TDD, this pattern was applied. We created an interface with all the expected behavior of a SCM, such as "cloning a repository", "go to a specific revision", etc. Then, we mocked through all the system. Later, concrete classes were created to deal with Git and SVN repositories.

Restfulie.NET (Aniche 2010) is a plugin to Microsoft's Asp.Net MVC framework, which allows developers to create RESTful web services. When developing the tool, there were many places in which we needed to make use of the internal Asp.Net MVC's framework API. At that moment, we decided to create an interface that puts an abstraction over the API. When the implementation of the current class was finished, we worked on the concrete implementation of that interface.

5. DEPENDENCY EXPOSURE

Also known as: Mock Hotspots, Contract Modeling.

A Blu-ray player explicitly expose the interfaces that it can have with other devices.

There are some classes that, as a consequence of architecture or a design choice, have a requirement to interact with other classes. A TDD session should be used to design the contract and the division of responsibilities between the class being developed and its dependencies. These dependencies usually are not a fixed class, but can be one of several implementations that implement an abstraction.

♦ ♦ ♦

How to define the contract between the developed class and a dependency during a TDD session?

Define contracts between classes is an important task in an object-oriented design. And, as any experienced developer knows, it is not simple. Many different reasons make the class design to rot over time. Such reasons are frequent customer changes, lack of the right abstraction for that specific problem, and so on.

If TDD is being used as a design technique, it should be possible to define these contracts, through the creation of the tests. The defined contract should be explicit and exposed as part of the system API. Although developers expect that the internal class implementation evolves, its interface to other classes should be more stable.

An instance of this scenario happens when the architecture defines classes with different roles, that should interact to fulfill the system requirements. Each class represents an architectural component, and all of them

should be decoupled. As an example, system architecture can define that data access and business rules must be decoupled.

Another scenario happens when it is used a design pattern where a class is composed by another. For instance, if **Strategy** (Gamma et al. 1994) is being used, part of the processing is delegated to an object that composes the main class. Since this implementation can vary, it is not desirable to develop the class coupled with a specific one. Additionally, the chosen implementation may not provide all the possible behaviors for that dependency, such as an exceptional scenario.

This situation happens in the development of abstract designs or frameworks. Their architecture is composed by hotspots, which are points where new behavior can be inserted. Sometimes, these points are only implemented by an application that instantiate the framework. In this case, the class dependency are not defined yet when it was being developed.

Therefore:

Use the tests to define a mock object that reflects the requirements of the class interaction with its dependency and design the contract between the class and the dependency by using this mock definition.

The greatest benefit in using a mock object in the class development is to improve the decoupling between the developed class and the class being mocked. To allow the mock object to substitute a real implementation, it is necessary to be possible to configure this dependency somehow. Additionally, a contract, such as an interface, needs to be used to define the dependence between the classes, for the mock to be able to implement this abstraction and emulate the necessary behavior for the test. These actions that aim testability, also have a huge impact in decoupling.

By testing two or more classes together, it is necessary to consider all the possible scenarios for them. There will be sets of tests where behavior in one class is fixed and on the other it varies. Depending on the number of possibilities in each class, it can increase the necessary number of tests, comparing to an approach where each class is tested separately. When one class is simple, that is not an important issue.

Another benefit of using mock objects in general is that it is easy to simulate all necessary behaviors to create the scenarios needed for test. When dealing with dependencies, it is common not to cover all possible outputs a dependency can generate. Exceptional scenarios, null returns, and so on, are examples. For instance, imagine a requirement of a class in which it needs to try again after receiving an exception on the first invocation. If you are not using a mock object it can be hard to create the behavior of throwing an exception on the first call and return a value on a second one.

This practice is recommended when you want to use the TDD process to design the contracts between the class and its dependencies. If this interface is already designed and you have an implementation that can provide every scenario necessary for test, it is a valid choice to implement the class integrated with a dependency, testing them together. That can happen when the dependency was developed first or when TDD is being used more for development activity than for design. However, when the implementation is not developed yet, this pattern needs to be applied.

This pattern is very useful when you have different teams working on different parts of the same application. A contract between the classes can be defined in the TDD session allowing the development of a class without the existence of its dependency. Furthermore, another team can use that interface to develop the dependency implementation.

As an example, consider that a developer is creating a class that represents a shopping cart in an e-commerce application. Every time that a product is added to the shopping cart, several functionalities of other subsystems need to be activated. For instance, the product category is added in the customer interests, a unit of the product should be reserved and an entry should be added on the product statistics. Facing these requirements, the developer decided to use the Observer (Gamma et al. 1994) pattern instead of notifying each subsystem individually.

In this scenario, involving a subsystem in the test makes it complicated, since it also depends on several different other behaviors. Additionally, the test can become coupled with several other functionalities, and will need to be modified if any of the subsystems change. Based on this, the developer may choose to use a dependency Exposure and create a mock object to design the Observer interface and how the shopping cart should interact with it.

Listing 5 presents an example of how a test using a mock object could be created to design the dependency interface. The interface ShoppingCartObserver is introduced in this test and added in the ShoppingCart class. By defining the mock expectations, that a method called productAdded() should be invoked when a product is added in the shopping cart, the developer iteratively defines the contract between the classes. Further tests could verify other scenarios, like when a product is removed and when one of the observers throw an exception.

Listing 5. Defining an Observer to a shopping cart

```
public class ShoppingCartTests {

   @Rule public JUnitRuleMockery ctx = new JUnitRuleMockery();

   @Test
   public void receiveProductAddition(){
      ShoppingCart cart = new ShoppingCart();
      ShoppingCartObserver mock = ctx.mock(ShoppingCartObserver.class);
      cart.addObserver(mock);
      final Product p = new Product("HDMI Cable", 10.5);

      ctx.checking(new Expectations() {{
         oneOf(mock).productAdded(p);
      }});

      cart.add(p);
   }
}
```

One can notice that the pattern does not completely remove the dependency. Still, it helps developers to reduce the problem that a bad dependency can cause. More of it can be read on Robert Martin's work (Martin, 2006) on class dependencies.

♦ ♦ ♦

This pattern shares the practice of mock creation with **Mock Complexity**, however the mocks are introduced for different reasons. In both cases mocks are applied to identify what the classes need from the dependency and to design its interface based on that. Because of that, this pattern can be considered a kind of **API Definition**.

When **Dependency Exposure** is used, you should not **Dive Deep** to develop the dependency. Since the mock is being used to simulate the dependency, you do not need the actual implementation to proceed with the current TDD session. Using this approach, the mock should be used to define the dependency interface, and them this interface must be implemented by the dependency to be developed in a further TDD session.

On Esfinge QueryBuilder (Esfinge 2013), the class MethodParser reads a method signature and identify which query elements it describes. To give this information to the class that generates a query to a type of database, it receives an instance of the class QueryVisitor and invokes methods on it. This solution was applied because the QueryVisitor is a hotspot and there are different implementations for several databases APIs. A ***Dependency Exposure*** *was used on the interface QueryVisitor to identify the methods that it needs and to verify if the invocation order performed by the class was correct.*

MetricMiner (Sokol et al, 2013) contains a set of implemented code metrics, which are executed on a source code and then persisted on the database. All these metrics are represented by the Metric interface. During the development of the first metric, we did not know what services the classes that implement this interface should provide to other classes. Then, we applied this pattern in the development of its client to discover the methods that it should provide to collect the metric, return the results, and save it in the database.

When developing Restfulie.NET's (Aniche 2010) configuration API, we had no idea on what methods would be needed. The combination of possible configurations was countless, as the user can configure supported media queries, remove default ones, and so on. We then applied the dependency Exposure pattern to identify and design the required methods to that API.

6. HIDE INTERNAL SOLUTION

Also known as: Dependency Encapsulation, Blackbox Test.

Not every component of a solution should be externally exposed, such as the wires that need to be plugged in to make a computer work.

Sometimes you need to use library classes or even create new classes to develop the solution needed to make the test suite to be executed successfully. Additionally, the class clients do not need to have access to these auxiliary classes used as part of the solution. When that happens, this functionality is usually used on specific points of the class logic. It is not desirable to couple other classes with this solution, because it may be refactored in the future.

♦ ♦ ♦

How to handle class dependencies in the tests when the dependency should not be exposed to class clients?

Differently from class contracts that are part of the system architecture and represent important relationships between components, there are simple class relationships that just aim to reuse existing functionality or to decouple functionality. The existence of this relationship does not concern the class clients and its part of internal class implementation.

When you create a mock object for an application interface, the test becomes coupled with that contract. Consequently, each change to the interface will demand changes to the tests. Because of that, it is desirable that the interfaces that are mocked on the tests should be stable, and do not change much over time. The idea is not to restrain the system from evolving, but to couple the tests only to interfaces that represent important contracts of the architecture, which change at a lower rate.

Therefore:

When the dependent class is part of the internal solution and its implementations does not need to be replaced, create the tests as if it does not exist. Encapsulate its use in the class and do not reference it on the tests.

When the developer decides that another class will be used as part of the solution, he should also decide if it is an explicit dependency in the architecture, or if it is a local decision and is part of the internal class solution. If the dependency is not exposed on the API and does not need to be replaced dynamically, it is a hint that this pattern should be applied.

If you apply this pattern, the TDD session will proceed normally as if there are no dependencies on the class. If the dependent class does not exist yet, you should develop this new class as part of the solution. If the new class is going to be reused in other contexts, the developer should **Dive Deep** and develops this new class before, and then proceeds with the development of the main class.

By using this pattern, the dependency might become coupled with the developed class, since this is not enforced by the creation of a mock object. At least, there is nothing on the tests that drives the design in the direction of their decoupling. However, as a tradeoff, the class can be refactored freely without affecting the test code.

Consider a class that represents a payment in an e-commerce system. As a requirement, this class should be able to confirm the information contained on it and verify if the data is the same later. The goal is to make it possible to detect frauds that change the values directly on the database. Listing 6 presents an example of how the test of this functionality was created.

Listing 6. The test that verifies the consistency of the payment data

```
public class PaymentTest {

   @Test
   public void verifyConsistency() {
      Payment p = new Payment();
      p.setCustomerName("John Jones");
      p.setCreditcard("1111 2222 3333 4444");
      p.setValue(250.0);
      p.setDate(new Date());
      p.confirmData();
      assertTrue(p.verifyData());

      p.setValue(200.0);
      assertFalse(p.verifyData());
   }
}
```

Since there are several approaches to make this verification, the developer makes a decision to keep the solution completely encapsulated inside the class. Listing 7 presents a solution that uses a hashing algorithm accessed by the interface MessageDigest to perform this verification. Furthermore, if a completely different solution needs to be adopted, like to store the same information on another location, the class can be refactored without affecting the tests.

Listing 7. Implementation that uses a MessageDigest to confirm data based on hashing

```
public class Payment {

   private String customerName;
   private String creditcard;
   private Double value;
   private Date date;
   private String hash;

   //getters and setters
```

```
    public void confirmData() {
        hash = generateHash();
    }

    public boolean verifyData(){
        String currentHash = generateHash();
        return hash.equals(currentHash);
    }

    private String generateHash(){
        try {
            String str = customerName+creditcard+
                      value.toString()+date.toString();
            MessageDigest md = MessageDigest.getInstance("MD5");
            md.update(str.getBytes());
            return new String(md.digest());
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException();
        }
    }
}
```

In the example, the usage of the MessageDigest class was simple and localized. An alternative and similar solution would be not to put this business logic here, but in another class (let's say HashGenerator), and make Payment to depend upon HashGenerator, however in a way that the external world does not know about it. For example, making it as a private attribute which is instantiated in Payment's constructor. Although the behavior is encapsulated in a different class now, the external world does not know it, and the solution is still internally hidden. So, this is another way to apply the pattern **Hide Internal Solution**, to keep the solution as simple as possible, avoiding unnecessary layers and abstractions.

♦ ♦ ♦

Hide Internal Solution and **Dependency Exposure** are concurrent patterns. When there is a dependency you need to choose which approach you choose. This choice is actually a design decision, because you can encapsulate the dependency from the rest of application or you can make it replaceable explicitly. How you choose to make your tests will have influence on the class design.

On Esfinge QueryBuilder (Esfinge 2013) this pattern was applied in the data structure used to store query parameters on the class MethodParser. There are classes to represent simple parameters and Composites to join parameters using "and" or "or". These classes were all developed in the context of the main class and were not exposed as part of the API. They have complete code coverage just by the MethodParser tests. This solution was adopted to allow the evolution of this structure without affecting the tests.

Code metrics on MetricMiner (Sokol et al 2013) are implemented using a parser generator that provides a way to visit the generated abstract syntax tree. These metrics contain the visitor as well as the code responsible to persistence. All the logic inside the visitor is hidden inside the class that implements each metric. The visitor is not exposed as part of the API, and therefore, it does not contain specific tests; it is tested through the Metric class itself.

To do the content negotiation, Restfulie.NET (Aniche 2010) reads the header that comes in a HTTP request and understands the type of media the clients are requesting. To do that, besides reading the headers, it is also necessary to understand the regular expression that represents the priorities over media types. When we implemented it, we decided to keep all that regex logic inside of the Content Negotiation class. As in all examples before, this logic is not exposed as part of the API, allowing the implementation to be changed easily.

7. CONCLUSIONS

As discussed, the practice of TDD and the inherent unit tests that are created during the process can be very valuable to developers, even from the design point of view. In this paper, we described a set of patterns, which a TDD practitioner can use, to improve the class design and dependency management. Based on these patterns, it is possible to understand the tradeoffs on different approaches for handling object dependences in a TDD process.

We intend to keep working on this catalogue. There are also other patterns, related to coupling, cohesion and simplicity, which will be discussed in future works.

ACKNOWLEDGMENTS

We thank Filipe Correia for being our shepherd. His help was fundamental during the development of this paper. We also thanks all the writers workshop participants for reading this work carefully and for giving their feedback.

REFERENCES

Aniche, M. 2010. Restfulie.NET. http://www.github.com/mauricioaniche/restfulie.net. Last access on June, the 1st, 2013.
Astels, D. 2003. Test-Driven Development: A Practical Guide. Second edition, Prentice Hall.
Beck, K. 2002. Test Driven Development: By Example. Addison-Wesley Professional.
Brown, M. and Tapolcsanyi, E. 2003. Mock Object Patterns. Proceedings of the 10th Conference on Pattern Languages of Programs.
Caelum. Alura E-Learning System. 2011. http://online.caelum.com.br. Last access on June, the 1st, 2013.
DWR – Easy ajax for java. Available at http://directwebremoting.org. Last access on June, the 1st, 2013.
Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of ReusableObject-Oriented Software. Addison-Wesley (1994).
Esfinge Framework. Available at http://esfinge.sf.net accessed on 31/05/2013.
Fowler, M. 2007. Mocks aren't Stubs. Available on http://martinfowler.com/articles/mocksArentStubs.html acessed on 31/05/2013.
Freeman, S. and Mackinnon, T. and Pryce, N. and Walnes, J. 2004. Mock roles, not objects, In: OOPSLA ´04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM, p. 236-246.
Freeman, S. and Pryce, N. 2006. Evolving an Embedded Domain-Specific Language in Java. In Proceedings of the Object-Oriented Programming, Systems, Languages & Applications (OOPSLA) 2006.
Guerra, E. 2012. Fundamental Test Driven Development Step Patterns. Proceedings of the 19th Conference on Pattern Languages of Programs.
Martin, R. 2006. Agile Principles, Patterns, and Practices in C#. First edition, Prentice Hall.
Meszaros, G. 2007. XUnit test patterns: refactoring test code. Person Education.
Perillo, J. and Silva, J. and Varga, R. and Guerra, E. 2011. SADE – Sistema de Atendimento de Despacho de Emergências em Santa Catarina. In Proceedings of XIII Simpósio de Aplicações Operacionais em Áreas de Defesa (XIII SIGE).
Sokol, C.Z., Aniche, M.F., Gerosa, M.A. 2013. MetricMiner: Supporting Researchers in Mining Software Repositories. 13th IEEE International Working Conference on Source Code Analysis and Manipulation.