Fundamental Test Driven Development Step Patterns

EDUARDO GUERRA, National Institute for Space Research, Brazil

Test Driven Development, popularly known as TDD, is a software development technique where the test code is developed before the application code. There were many advances on design techniques for test code automation, however many people still have difficulties on using TDD as a design technique. One of the reasons for this is that it is hard to identify which test should be created to drive the production code development. This paper is the result from a study that analyzed the TDD process and abstracted the kinds of steps that can be done through the creation of tests. Each kind of step was documented as patterns that capture the motivation and consequences of each one. As a result, through these patterns it is expected that developers create a better understanding of the TDD process and how each kind of test can be used to guide production code development.

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: Test-driven development, patterns, TDD

ACM Reference Format:

Guerra, E. 2012. Fundamental Test Driven Development Step Patterns. 19th Conference on Pattern Languages of Programs (PLoP), Tucson, Arizona, USA (October 2012), 15 pages.

1. INTRODUCTION

Test Driven Development (TDD) is a development technique in which automated tests are developed before the production code (Beck 2002). When tests are codified, the developer is defining the client view of the developed class, how the class API will be used to invoke functionality, and what the expected behavior is for each test scenario. TDD practices also state that after the test, the implementation should be the simplest possible solution to make all the existing tests pass. Refactoring the code after the tests pass is also a key part of TDD and is used to keep the design clean, to make code more clear, and remove any kind of code duplication. Tests help to ensure that the class behavior remained unchanged and nothing breaks during refactoring. A TDD session is a set of successive short test-code-refactor cycles to develop a desired functionality or to implement a user story.

The java framework JUnit as evolved from the SUnit framework, is the base for most XUnit test tools on other languages. These frameworks allows a developer to quickly and easily write unit tests for basic functionality thus enabling the development to be primarily driven by tests (Beck 2004). Due to the rise of popularity that TDD reached on the software development industry, other test frameworks were developed to enable and facilitate the implementation of tests for specific kinds of classes. For instance, DBUnit focus on tests for classes which access the database, providing functionality to initialize, clean and verify data on databases (Smart 2008). Another example is frameworks for mock objects (Freeman et al. 2004), which create at runtime fake objects that mimic the API from the class dependencies and is able to simulate behavior to generate a test scenario and to verify the expected behavior from the tested class.

Recent advances on tools and frameworks have been key to help TDD to become more successful, making it possible and viable to create tests for different platforms and architectures. However, the creation and design of automated tests is not the main difficulty when TDD is being used as the development technique. Since the application code is guided by the tests that are being created, the order in which these tests are introduced and which piece of functionality each one focus are important to let the code and design to emerge. Frequent

This work is supported by the Widget Corporation Grant #312-001.
Author's address: Laboratory of Computing and Applied Mathematics (LAC), National Institute for Space Research (INPE), P. O. Box 515 – 12227-010 São José dos Campos, SP, Brazil; email: guerraem@gmail.com;

questions for beginners on TDD are: How do I start? Where do I start? What is the next test that I should create? When should I implement class functionality instead of only making the tests pass?

On TDD terminology, the metaphor "Baby Steps" is used to refer that each test introduced should represent a small increment on functionality considering the previous tests. The increment size can vary with the developer experience, but it is important to alternate fast between the development of test and production code. The present work borrow this metaphor to refer as a "step" the process to create a test that have an intent on moving forward the development on the TDD process. Each pattern documents a kind of step that can be done on a TDD session based on how the developer wants to guide the evolution of the production code. On other works, each pattern will describe a TDD cycle iteration. The scope of this paper is to introduce the basic TDD step patterns, which are described on the following sections.

Most of the examples of the patterns were taken from the development of Esfinge QueryBuilder framework (Esfinge 2012), which is a framework that was developed and design completely using TDD. The basic functionality of this framework is to interpret the method signatures on an interface and generate database queries based on code conventions. During the development of this framework the step patterns repeated on different classes and scenarios, providing material to abstract the documented patterns. Some known uses were also taken from the following projects where the TDD session is described step by step: Money project (Beck 2002) and Movie project (Astels 2003).

This paper presents a study based on the author's experience on TDD and some documented TDD sessions to identify patterns on which kinds of test can be introduced to move forward on a TDD session. The patterns do not focus on the test itself, but in the role of it on the evolution of the application code. The goal is to provide a perspective of the TDD process and the options that can be used to move forward on the development. The primary goal of this paper is to help developers that are new to TDD in taking "baby steps" in order to be successfully in learning this development technique.

2. TOWARDS A PATTERN LANGUAGE FOR TDD STEPS

The goal of this paper is to document the fundamental patterns that can be the base for a pattern language for TDD steps. The following presents a list of the patterns presented on this paper and Figure 1 presents the relationship between them considering the structure of a TDD session:

- **`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 section, add a test that increments a little the functionality verified with the existing 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 that are tested separately, create a more complex scenario where these features should work together.

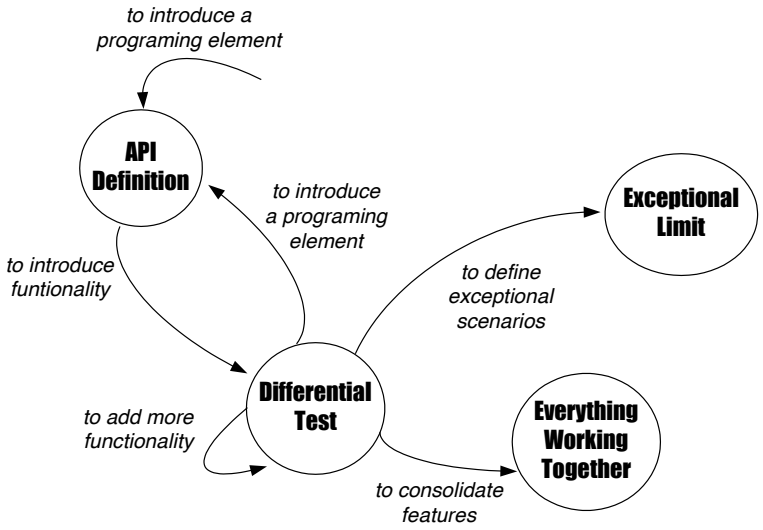


Fig. 1. TDD Step Patterns Navigation.

The patterns described on this paper focus on the basic steps of a TDD session. However there are other patterns that still need to be documented. The goal is in the future to define a pattern language that can represent the dynamics of a TDD session. The following presents a list of some identified patterns that still need to be documented and Fig. 2 presents a draft that presents the relationship of all candidate patterns of the patterns language:

- **`Bug Locator`**: When a bug is found, create a new test that fails because of it. 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 auxiliary methods or classes, ignore temporarily the current test and start a new small 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.
- **`Dependence Exposure`**: When you need to define an API from an explicit dependence of the application class, create a test that creates a Mock Object and define the expected calls to the dependence API.

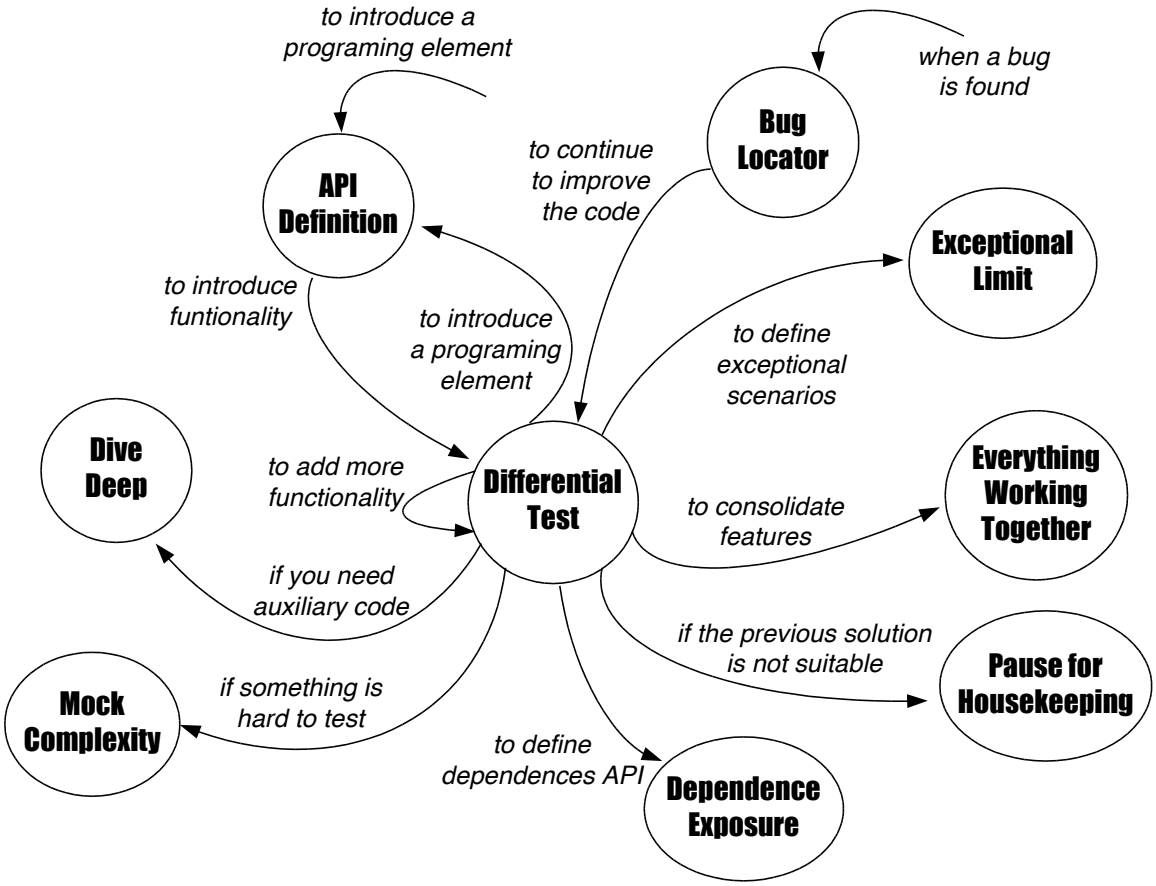


Fig. 2. Complete TDD Steps Pattern Language.

3. API DEFINITION **

Also known as FIRST TEST OF CLASS, DO NOTHING, CREATION TEST, EMPTY CLASS TEST

You first need to worry about how the client looks at your class, before concerning about functionality behind it.

It is a very hard task for anyone starting with TDD to begin a new development session. What is the first test that needs to be introduced? This test is important because it will determine the core behavior and methods that will be programmed on the target production class. This pattern focuses on the first step of a TDD session, presenting what should be the main concerns of the developer during the beginning of a session. This pattern also applies when a new programming element is introduced, such as a new class or a new method.

* * *

The first test of a TDD session introduces new programming elements, such as what classes and/or methods and what the desired functionality should be in order to make the test pass. What kind of test should be used to introduce a new programming element?

The "Baby Steps" principle on TDD says that the development should be done in small increments, alternating between test and production code. Based on that, it is not desirable that the first test that introduces a new programming element includes a complex scenario that will demand a lot of development effort. On the other hand, even being very simple, the first test should verify some functionality. The goal is to verify simple functionality, but something meaningful enough to be the first iteration on the production class.

An initial test can feel very simple for the developer, giving the impression that no progress is being made. That can motivate the introduction of a complex first test scenario that will demand a great effort on the production code. It is important to notice that the simple introduction of a new programming element on the API is a meaningful advance.

Therefore:

Create a test that verifies the most trivial scenario about the class or method being introduced. In case of a method, use the parameters that return an immediate response. In case of a class, create a new instance and verify the state of the just-created object.

In this first test scenario that introduces a programming element, the goal is not to have the functionality implemented. The primary goal should be to define the class API and/or the method signatures. In this step, the developer should only be concerned about what the class API should look like for its clients, in other words, the external class API. Since the test simply verifies a trivial scenario, the implementation can be very straight forward, just returning the value expected by the test. That might seem like cheating at first, but it is important to highlight that the actual implementation is not the objective at this point and it will be handled on future tests.

As an example, consider the test of a class that represents the data structure Stack. The first test can be the expected return of some class methods after the creation of a new instance. It is important to remember that the main job at this moment is to define the class API, so the developer should think about what method calls would make sense at this point. In the example, a method that returns the stack size should return zero and a method that answers if the stack is empty. The test method is presented on Listing 1 and the resulting class on Listing 2.

Listing 1. First test on a TDD session for a Stack class.

```
@Test
public void emptyStack(){
  Stack s = new Stack();
  assertTrue("Stack should be empty",s.isEmpty());
  assertEquals("The stack size should be zero", 0, s.size());
}
```

Listing 2. Implementation of production class returning the expected value.

```
public class Stack{
  public boolean isEmpty(){
    return true;
  }
  public int size(){
    return 0;
  }
}
```

Listings 3 and 4 present additional examples that illustrate the application of this pattern to develop another classes. The example on Listing 3 represents the first test of a method that parse a string in camel case. In this case, the most trivial scenario is to parser a string that does not have an uppercase letter, in which the return should be the string itself. Despite the implementation is straightforward, this is an important test scenario that should still working when the class evolves. Listing 4 presents the initial test for a class that represents a connection. The most trivial class state in this case is when the connection is not initialized yet.

Listing 3. API Definition on the creation of a method that parser a string based on a camel case.

```
@Test
public void simpleCamelCase(){
  String s = “common“;
  String r = splitCamelCase(s);
  assertEquals(s,r);
}
```

Listing 4. First test to develop a class that represents a connection.

```
@Test
public void disconnected(){
  Connection c = new Connection();
  assertFalse(c.isConnected());
}
```

* * *

An **API Definition** does not need to be included only on the beginning of a TDD session. It can also be used when the functionality is incremented, but a new API element is introduced. For instance, a **Differential Test** can be done for functionality, while an **API Definition** is being performed for a method introduced in that test. For complex scenario, sometimes the simplest scenario is the one in which does not work, so the **API Definition** can also be a **Exceptional Limit**.

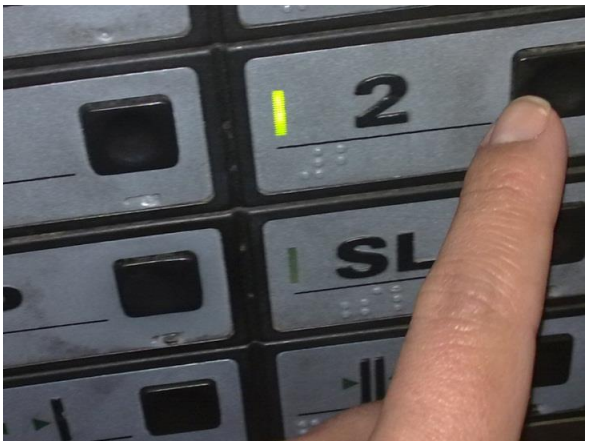
On Esfinge Framework (Esfinge 2012), this kind of test can be observed for the class QueryObjectMethodParser, which parsers a method signature to identify a related class. The tests started with a method called fitParserConvention() that identifies if that method signature is can be parsed by it. The first three tests included distinct scenarios where the method should simply return false.

Another example of this pattern was on the test of a class which encapsulates the usage of a component that interprets an expression language. The first test was the most trivial expression that is a simple variable, which is a scenario where no additional configuration on the component needed to be done.

On the Movie project (Astels 2003), the first test introduced for the class MovieList is the test of an empty list, where tje returned size should be zero. The implementation on this first step was to fake the implementation and make the test pass to move forward on the TDD session.

4. DIFERENTIAL TEST **

Also known as MOVING FORWARD, JUMPING THE BOUNDARIES, SMALL INCREMENT, NEXT SIMPLEST SCENARIO



When someone enters a lift, he knows that the expected behavior if he does nothing is for the lifter to stay immobile. To test the other lift features he should do something different, like to press a button.

After the first test, a new test should be introduced to allow the development to move forward. Depending on what functionality is verified in this test, a new set of features are included on the production class. This increment should be small to enable a quick interchange between test code and production code. This pattern describes how a new test should be introduced to induce a small increment on application class functionality.

* * *

A TDD session should move forward by adding a test that induces a small increment on the application class behavior. How to create a new test that increments the behavior that is verified by the previous tests?

The addition of a test that does not test new functionality will not help the TDD session move forward. A common statement on TDD is *"add a new test and watch this test fail"*, which means that a test should challenge the developer to make it pass. On the other hand, to have a test that adds too much new information will not be helpful either. This will demand a lot of effort on the developer and the application code in order to make the test pass, thus breaking the interchange rhythm between the test and the application code.

The step size can vary according to the developer's experience. What is a big step for some developers can be a small one for others. The step should be enough to allow a safe increment and a fast alternation between test and production code.

Therefore:

Create a test scenario that expects a result different from the previous tests. It will induce a modification on the application class to handle that new situation. This new test should verify new functionality, but should be a small addition compared to the behavior already verified by the previous tests.

If you already have a test in which the expected return is true, than create a new test where the return is false. This will provoke a change on the application class necessary to handle both situations. Nothing stops the developer to use conditionals to fake the result, thus making the test bar green and moving on to the next test. Faking implementations is advised when the implementation would be a large step for the moment, or when the step goal is more to define API then to implement behavior. However, in this case, this pattern should be applied again with a different scenario from the previous ones. At some point, this conditional that fakes the class behavior will smell bad, containing dumb code duplication, and the actual implementation will need to emerge.

When the application class behavior is faked on the first test, it is valid to add a similar test with different data to induce the implementation. However, to repeat this again will cause the test to pass without changes on the application class. To move forward, the next test needs to focus on a different ***Equivalence Class Partition*** (Burnstein 2003). The equivalence class partitioning is a test technique that divides the test cases considering values intervals where a class or a method behave similarly. In order to motivate a change on the application class, the next test should explore new possible behaviors.

It is important to remember that the tests created on TDD are not completely black box. On the one hand, the developer should obey the external class API avoiding dependences on internal details, but on the other hand, he knows what is already implemented on the application class. This knowledge can be used to identify scenarios in which the code will need some implementation effort to work.

Continuing the Stack example, to move further, the next test should verify a different behavior than the emptyStack() test. Since on this test the isEmpty() method should return true and an invocation in size() should return zero, the next test should expect different results. The test presented on Listing 5 pushes an item on the Stack to provoke a different behavior these two methods. It is important to notice that two new methods are introduced in this test.

The method push() is used to provoke a change on the other methods behavior, so some kind of state should be stored to cause that. The method top() only returns a value, and, since it is not the main focus on this test and it is the first time that it appears, its implementation can be faked. Listing 6 presents the Stack source code evolved to make the tests to pass. The method isEmpty() and size() now receives a non-fake implementation, but that does not means that it should be the final one. This one is just enough to satisfy the current tests.

Listing 5. Moving forward on the TDD session for a Stack class.

```
@Test
public void pushOneElement(){
  Stack s = new Stack();
  s.push("item");
  assertFalse("Stack shouldn't be empty",s.isEmpty());
  assertEquals("The stack size should be one", 1, s.size());
  assertEquals("Inserted item should be on top", "item", s.top());
}
```

Listing 6. The evolution of the production class motivated by the test.

```
public class Stack{
  private int size;
  public boolean isEmpty(){
    return size==0;
  }
  public int size(){
    return size;
  }
  public void push(Object o){
    size++;
  }
  public Object top(){
    return "item";
  }
}
```

To move forward from here, the next test could add a different scenario for the method top(), which have a faked implementation. That will provoke a change on the application class to add an implementation for this method. The knowledge of what implementation is faked or deficient can be used to generate the next test scenario.

* * *

When the TDD session focus on an entire class, other methods from the class API can be introduced on the test in order to motivate a change in behavior. This test can be an **`API Definition`** for that method, while it is a **`Diferential Test`** for other ones.

On Esfinge Framework (Esfinge 2012), the **`Diferential Test`** *was used several times for the test of the implementation of the interface MethodParser. Since this class parses a method signature in order to identify a database query that it represents, each test add a new element to the signature, such as a new term on the method name or a code annotation, in order to guide the development of that feature on the application class.*

Still on Esfinge Framework, the implementations of QueryVisitor also use this pattern several times. This class is expected to generate an object that represents a query based on the method call sequence that it receives. On the test class, the sequence elements were continuously being incremented through the tests in order to evolve the implementation support of different kinds of query.

On the Money project (Beck 2002), the equality was first tested and implemented for the same type of currency. Further, a new test was introduced considering the equality between two different currencies, motivating a change on the application class to handle this scenario.

5. EXCEPTIONAL LIMIT**

Also known as ERROR TEST, ERROR SCENARIOS, TEST EXCEPTIONAL BEHAVIOR

There are some scenarios that cross a limit in which the class should not execute its normal behave. This error scenario is part of the behavior that should be implemented, and consequently a test for it should be introduced.

It is important to define the limits in which the class should behave correctly and for which kinds of scenarios the normal class behavior is not expected. Making sure that exceptions for a class are being handled properly is important. This pattern describes the introduction of tests which define the exceptional limits of class behavior.

* * *

A TDD session should not only motivate the implementation of application class regular behavior, but should also make it raise errors on the expected scenarios. How to define tests which helps to introduce and define exceptional behavior on application classes?

When a new class is developed through TDD, the main concern is to develop the class functionalities and to add **`Differential Tests`** which motivates the introduction of new behavior. Because of that, the developers often forget to define scenarios where the class is not expected to behave normally, including exceptional scenarios, invalid values and other situations where an error should be raised. Following the TDD process, if these scenarios were not included on the test cases, they will not be implemented on the application classes.

Because the addition of **`Differential Tests`** improves class functionality, there is a temptation to forget about behavior which deals with exceptional scenarios. Sometimes it is easier to think about good scenarios proving that they work through the implementation of class functionality.

Therefore:

Create a test where invalid values are introduced or an invalid scenario is defined, and verify if the class behaves accordingly. It will motivate the handling of these situations on the application class.

There are different ways in which an exceptional behavior could be handled on an application class. On languages that implements exceptions, the class usually throw an exception representing that error. However there are other ways in which the errors could be represented and returned to the class client. Common

examples are a special method return value, such as a boolean value representing execution success, null values, Null Objects (Wolf 1998) or an instance of a special subclass or with a flag value. The error could also be stored on a class variable to be retrieved by the class client.

One of the goals of the test used to verify these exceptional situations is to define what strategy the class will use. For instance, if the class should throw an exception on a given scenario, your test should provoke that situation and verify if the exception is thrown accordingly. By defining this test, the next step would be to implement this behavior on the application class. It is not in the scope of this pattern to discuss the alternatives for handling errors on the API, but to propose the introduction of a test which defines how it is done.

A common mistake is usually made by the usage of the "expected" attribute on JUnit 4 @Test annotation, which should receive an Throwable type which is expect to be thrown on the test method. When this JUnit feature is used, the test pass if the exception is thrown anywhere on the test method. If the same exception can be thrown on other steps of the test method, that can make the test to pass with a wrong application class behavior. When the exception can happen on more than one step of the test method, a try/catch block should be used to capture the expected exception on the expected line of code.

In the Stack example there are scenarios where the class should raise an error, such as to pull an element when the stack is empty and to push an element when the stack is full. Listings 7 and 8 present the tests that could be added for these two situations. On the first one, there is only one method call which can throw the expected exception, which make possible the usage of the "expected" attribute on @Test annotation. On the next test, the exception can be thrown on every invocation of push() method, but according to the test scenario it should happen only on the last one. Because of that, the try/catch block should be used around the last call to capture and verify the expected exception. The fail() calling is to make the test to fail if the exception is not thrown on the previous invocation.

Listing 7. Defining the limit to do not allow the pull on an empty stack.

```
@Test(expected=EmptyStackException.class)
public void removeFromEmptyStack() throws EmptyStackException {
  Stack s = new Stack();
  s.pull();
}
```

Listing 8. Defining the limit of the stack capacity.

```
@Test
public void addOnFullStack() throws FullStackException {
  Stack s = new Stack(10);
  for(int i=0;i<10;i++)
    s.push("item"+i);
  try{
     s.push("overload");
    fail();
  }catch(FullStackException e){}
}
```

After the introduction of each test, the developer should work on the Stack class to implement the specified behavior. In this example, internal verifications should be performed based on the class state to throw the exception on the scenarios specified in the tests.

* * *

When the application class behavior in very complicated, sometimes a good choice is for the **`API Definition`** to be an **`Exceptional Limit`**. This way the developer starts by defining the normal behavior limits, and then

proceeds to the actual behavior implementation. A **`Exceptional Limit`** can also be considered a special case of **`Diferential Test`** where the exceptional scenario represents a kind of equivalence class partition.

On Esfinge QueryBuilder (Esfinge 2012), the TDD session to create JPAQLQueyVisitor contained a lot of **`Exceptional Limits`** *to throw the InvalidQuerySequenceException when the invocation sequence on the* **`Visitor`** *is invalid. The number of wrong possibilities was so many, that further this part of the implementation was refatored and separated in the class ValidationQueryVisitor, a* **`Decorator`** *which encapsulates the class with the main functionality.*

Still on QueryBuilder, the test class QueryObjectMethodParserTest started the TDD session with scenarios where the method signature does not follow the conventions expected by the class. The three first tests focused on these Exceptional Limits, and only on the forth test it verified a correct method signature. This is an example of **`Exceptional Limit`** *which is also an* **`API Definition`***.*

On the Movie project (Astels 2003), the feature that supports the renaming of the class Movie has restrictions that the Movie should not be renamed to a null nor an empty string. The first test introduced verified the modification of the name to a valid value. The following ones define these limits and verified if an IllegalArgumentException is thrown when a null or an empty string is passed as an argument.

6. EVERYTHING WORKING TOGETHER *

Also known as CONSOLIDATION TEST, COMPLETE SCENARIO

After you verify that functionalities are working insolated, you need to be sure if they are also working together.

On a TDD session, by the introduction of Differential Tests, the developer is concerned to add continuously small pieces of functionality. By creating the simplest test in which every feature works in isolation, the implementation not necessarily supports a more complex scenario. This pattern describes the test that is added on a TDD session to make sure that the currently implemented features can work together on a more complex scenario.

* * *

The `Differential Tests` are created to continuously add functionality on a TDD session usually focus on the simplest scenario in which a feature can be verified. Implementing the simplest solution on each step does not guarantee that the features will work together on a more complex scenario. How to make sure that the test suite verifies if the integration is implemented on the application class?

To keep the TDD cycle rhythm, alternating between test code and production code, the tests should be simple and focus on a single and small concern. With this strategy, it is possible to make the tests to pass quickly and

move forward using a new **`Differential Test`**. However, the implementation of the simplest solution for each one of the tests does not guarantee that the features will work together appropriately. At some point, it is necessary to add a test which will induce the developer to handle the integration or the coordination of both features.

Therefore:

Create a test with a scenario in which two or more features are used combined. If the integration is already working, the test should pass. However if the test does not pass, the developer should modify the application class to handle this scenario.

One of the pieces of TDD cycle is to add a new test and watch that test fail. This new test should fail because it should motivate a change on the application class to make it pass, so it can evolve through the addition of tests. To see if **`Everything`** is **`Working Together`** is one of the scenarios where it worth to add a test and may be it can pass without an additional implementation.

Different from a **`Differential Test`**, the goal of this TDD step is not to add new features on the application class, but to verify if the existing features work well combined in the same scenario. The implementation of the simplest scenarios for the features could already take care of the integration, however if not, it is necessary this new test to induce a change on the application class. This step is also important because a refactoring can affect how the features work together and this test will help to detect a change on that behavior.

When TDD is used as the development technique, the test coverage considering only lines of code is usually very high. However if the features are only tested isolated, the tests will not cover a high number of paths considering the code control flow. The introduction of this kind of step usually provokes the verification of an unexplored path, possibly detecting a fail or a bug on the features integration.

On the Stack running example, consider that a method pushAll(), which pushes all elements on a list, was already implemented motivated by a previous test. A further step to verify if is **`Everything Working Together`**, it would be to verify if overloading the stack using the pushAll() method would also throw the FullStackException. The test is presented on Listing 9 and also mix the addition of elements using push() and pushAll(). The pushAll() feature was already introduced on previous test, as well as the exception throwing when the Stack is full, and this new test's goal is to make sure that the are working together.

Listing 9. A scenario combining two features.

```
@Test
public void fullStackWithPushAll() throws FullStackException {
  Stack s = new Stack(10);
  List<String> list = Arrays.asList("a", "b", "c", "d", "e");
  for(int i=0;i<6;i++)
    s.push("item"+i);
  try{
     s.pushAll(list);
    fail();
  }catch(FullStackException e){}
}
```

A clearer example of this pattern is presented in listings 10 and 11. Two tests of the class ShoppingCart are presented in Listing 10, respectively to verify features to add a coupon and to introduce a discount on the tested class. Despite these tests verify if each functionality is working separately, nothing guaranties that they will work together. So, using `Everything Working Together`, Listing 11 introduce a new test with a scenario where these features are used combined.

Listing 10. A scenario combining two features.

```
@Test public void coupon(){
  ShoppingCart s = new ShoppingCart();
  s.addItem("Product",1000);
  s.addCoupon(100,"23473495734957");
  assertEquals(900, s.total());
}
@Test public void discount(){
  ShoppingCart s = new ShoppingCart();
  s.addItem("Product",1000);
  s.addDiscount(0.15);
  assertEquals(850, s.total());
}
```

Listing 11. A scenario combining two features.

```
@Test public void couponWithDiscount(){
  ShoppingCart s = new ShoppingCart();
  s.addItem("Product",1000);
  s.addCoupon(100,"23473495734957");
  s.addDiscount(0.15);
  assertEquals("Coupon before discount",765, s.total());
}
```

* * *

This kind of TDD step should be applied after a sequence of Differential Tests, which can also contains some Exceptional Limits. It is advised to finish the implementation of each feature separately before adding a test that considers **`Everything Working Together`**. The introduction of this kind of step prematurely can generate a great implementation effort breaking the TDD cycle rhythm.

Esfinge QueryBuilder (Esfinge 2012) uses this kind of TDD step repetitively for the development of the classes that are responsible for parsing the method signature to extract the information of the query that should be generated. Features like the introduction of query parameters, query ordering and the addition of new domain terms are first developed by tests that verifies each one of them in isolation and after tests combining the features are also introduced.

On the Money project (Beck 2002), the conversion among currencies and the addition of amounts was already being tested and implemented. A new test integrated both functionalities considering the sum of amounts from different currencies. In this example, this new test demanded changes on the application classes to work.

7. RELATED PATTERNS AND PATTERN LANGUAGES

The xUnit Test Patterns (Meszaros 2007) describe a pattern language for testing, describing best practices and problems that can happen on an automated test suite. These patterns focus on techniques to create automated tests not necessarily inside a TDD session. They are useful in the context of TDD since a good code quality is important for the sustainability of iterative practices. In this sense, these patterns complement the ones on this paper, since they provide best practices on test design, which can be applied on any TDD step.

On the third part of "Test-driven Development by Example" (Beck 2002), some TDD patterns were presented. The first difference from the present work is that each patterns there contained just a small text, which is not on a pattern format. Another difference is that the author presents patterns that embrace test best practices, ways to make a test pass and even about the physical setup for TDD. All the patterns on this paper belong to the same category, which documents the kinds of TDD steps which can be made to move the implementation forward.

8. CONCLUSION

Test-driven development is a design and development technique where the tests are created before to guide the implementation of the application code. One of the difficulties to start on TDD is to understand which kind of steps can be performed to drive the code towards the desired direction. This paper abstracted on patterns the kinds of steps used on a TDD session. The patterns included the motivation and consequences of each kind of steps.

A natural continuation of this work is to document the remaining patterns and organize them on a pattern language. Other sources of TDD sessions are also important to be considered in order to find more known uses to confirm the documented patterns. Finally, a future work can also evaluate the usage of these patterns to teach the TDD technique.

ACKNOWLEDGMENTS

I would like to thank to the shepherd and friend, Joseph Yoder, for the discussions, suggestions and advices to this pattern collection. Additionally, I thank to all participents on the writters workshop that gave me essential feedback to improve this paper. I also thank for the essential support of CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) to this research.

REFERENCES

Astels, D. 2003. Test-Driven Development: A Practical Guide. Prentice Hall.
Beck, K. 2002. Test Driven Development: By Example. Addison-Wesley Professional.
Beck, K. 2004. JUnit Pocket Guide. O'Reilly Media.
Burnstein, I. 2003. Practical Software Testing: A Process-Oriented Approach, Springer.
Esfinge Framework. Available at http://esfinge.sf.net accessed on 5/32/2012.
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.
Meszaros, G. 2007. XUnit test patterns: refactoring test code. Person Education.
Smart, J. 2008. Java Power Tools. O'Reilly Media.
Woolf, B. 1998. Null Object. In *Martin, Robert; Riehle, Dirk; Buschmann, Frank. Pattern Languages of Program Design 3*. Addison-Wesley.