The Last Two Test-Driven Development Step Patterns: Bug Locator and Pause for Housekeeping

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

Abstract. *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 TDD Steps pattern language is an effort to document the different kinds of actions that a developer can perform on TDD to drive class behavior and design. In two previous papers, we introduced eight patterns of this language. This paper aims to introduce the two remaining patterns. One pattern focuses on the creation of a test to simulate the scenario of a known bug isolating it to a specific area of code, and the other one a pause for a more deep refactoring before introducing a new functionality.*

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, 2006).

The terminology used by the TDD community uses the metaphor “baby steps” (Beck 2002). 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 desired design and implementation. 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.

Previous papers (Guerra 2013; Guerra et al. 2014) documented eight patterns of the proposed pattern language. The goal of this paper is to present the last two patterns that are presented in the current version of this language: **Bug Locator** and **Pause for Housekeeping**. The pattern **Bug Locator** describe the step used to locate a new bug by using a test in order to start a TDD session from this point. The **Pause for Housekeeping** describe an approach to be used when the solution adopted by the class is not suitable for the next requirements.

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. This study comprehends a pattern mining effort which used as a source of research documented TDD sessions in books and information about the usage of TDD in real projects where the authors worked on. Next section describes briefly the 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. In this context, a TDD session can be defined a continue amount of time where some implementation is performed by using TDD. 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 on the system development.

The target audience of this pattern language are software developers interested in using TDD to design and develop software. It can be used by beginners to understand the mechanics of this design technique and, by more advanced practitioners to enable a better understanding of their design choices. The discussions presented in each pattern intend to clarify the consequences of each choice of step. The patterns names form a terminology to reference the alternative steps that developers can perform.

Instead of being inflexible about the dynamics of a TDD process, this pattern language prefer 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 choices. Future evolutions and additions for this pattern language may reveal other possible steps that can complement and enhance 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 (Guerra 2013) and the ones in black are documented in (Guerra et al. 2014). The remaining ones, in white, are the ones that are the focus of this paper.

The arrows in this diagram represents the paths that you can follow to choose a pattern aiming to move forward in a TDD session. The patterns with starting arrows are patterns where you usually start a TDD session. If TDD were a dance, the **Differential Test** would be the basic step. From it, you can decide to apply some other pattern based on the direction where do you want to drive the design. After applying them, the developer should step back to the central pattern and continue the “dance”.

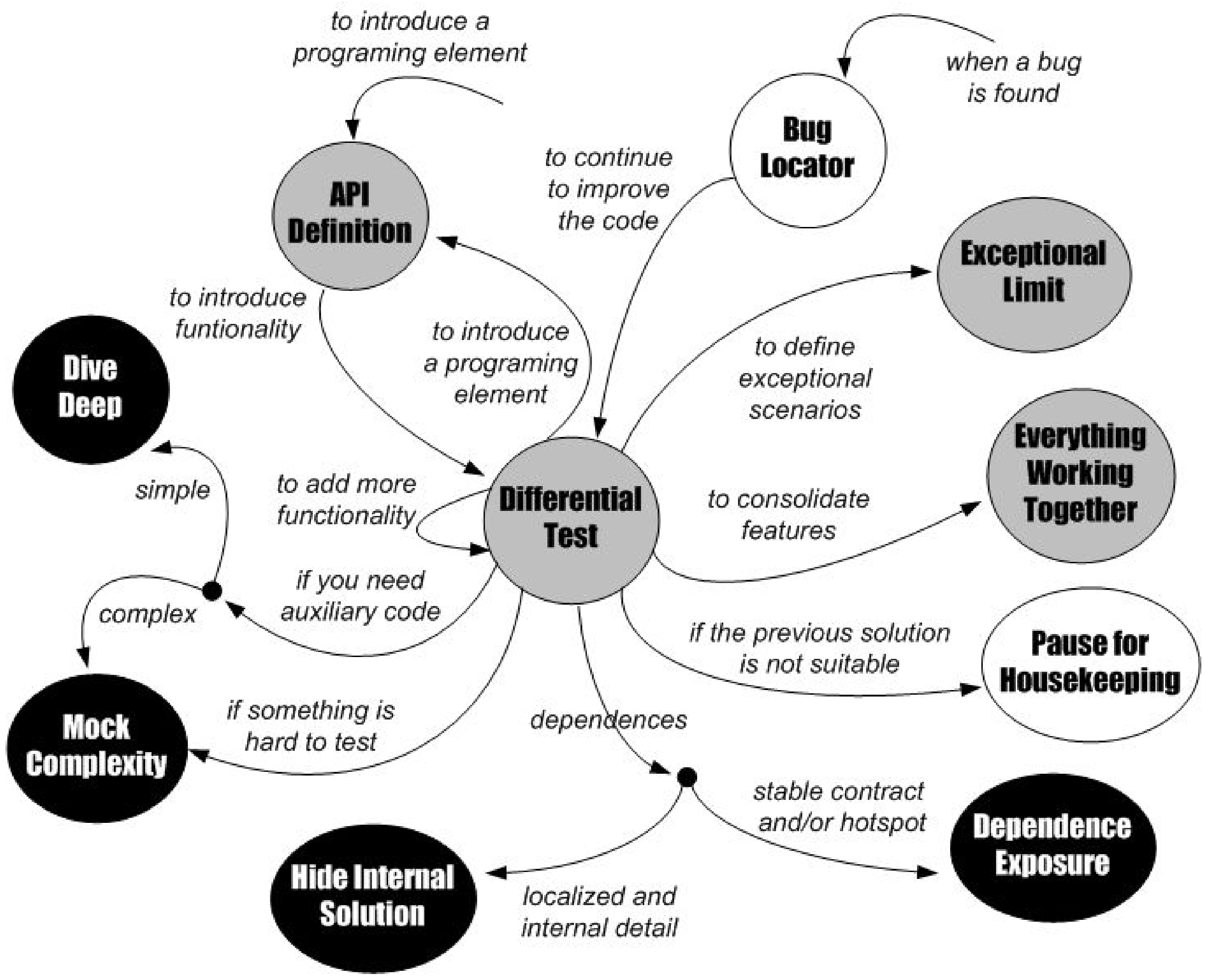
to introduce a programing element
API Definition
to continue to improve the code
Bug Locator
when a bug is found
Exceptional Limit
to introduce funtionality
to introduce a programing element
Dive Deep
to define exceptional scenarios
Everything Working Together
simple
to add more functionality
Differential Test
to consolidate features
complex
if you need auxiliary code
if the previous solution is not suitable
Pause for Housekeeping
Mock Complexity
if something is hard to test
dependences
stable contract and/or hotspot
Dependence Exposure
Hide Internal Solution
localized and internal detail

Figure 1 - TDD Step Patterns Map.

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 reported, either by an user or during an exploration test, 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 nested 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.

3. Bug Locator

Also Known as **Isolate the Problem, Finding the Bug**

To be able to eliminate a bug from your system, you should create a test that traps it in an isolated place where you can easily find.

Developing an application by using TDD does not make it immune to bugs. So, an exploratory testing might detect a bug, or an user can find that something was not working properly on a given scenario. When a bug is detected on an application developed using TDD it is probably because that scenario was not covered on the developed tests.

* * *

How to handle bug correction by using TDD?

When a problem happens in a software, usually there is a lot of pressure from the users and product owners for a correction to be provided as soon as possible. Facing this pressure, some teams and managers sometimes bypass the development process and techniques used in the development of the target software in order to correct the software faster.

A TDD process leaves a suite of automated tests that can be used as regression tests to verify if the behavior verified on them does not change after future changes. The behavior in scenarios that are not covered by these tests cannot be verified in future code changes.

Sometimes, when the developer go straight to the code to correct the bug, he can change a piece of code that is not responsible for the problem. Without being sure about the scenario in which the problem happen, there can be waste of time by working on the wrong part of the system.

Therefore:

Create an automated test which express the scenario in which the bug happens. If that test fails, start by making it pass and proceed with the TDD session until the problem is completely corrected.

A difficulty that happens when a bug is detected is to locate where it is in the source code. Since unit tests focus on single classes, when a test with the failed scenario can be introduced in a class tests and fail because of of the bug, it is clear that, at least, part of the problem is there. By doing that, the bug is localized and contained, and now, what the developer needs to do is to follow the TDD process making that test to pass with the simplest possible solution.

After the implementation of the first test, some **Differential Tests** could also be introduced to explore other similar scenarios that could also contain bugs. A unadvisable approach can be to try to handle bug scenarios without having a failed test, because the developer can try to correct an error that does not exist in the target class.

To exemplify the usage of this pattern, consider that an user see in the system a vehicle licence plate with a invalid character and report a bug about it. When trying to solve this issue, the first assumption is that the problem is in the class which validate licence plates. Since there were no tests that focus on this scenario, a new test, such as the one presented on Listing 1, is added to try to locate the bug on the suspect class. If the test fail, the source of the problem is found and the developer just need to make this test to pass.

If the test pass, that means that the test does not captured the right failed scenario or that the problem is somewhere else. If the problem is on the chosen scenario, the test should be changed in order to try to provoke the bug appearance. If the developer thinks that the problem might be on another class, a new test should be introduced on the test suit that cover that other class.

Listing 1 - Adding a test to verify a wrong licence plate found by an user

```
@Test
public void licencePlateWithSimbol(){
   LicencePlateValidator v = new LicencePlateValidator()
   assertFalse(v.validate("A#A8934"));
}
```

A good consequence of this pattern is that the developer avoids messing with code that are not related to the bug, since the test should be detecting it. Another benefit is that because there is a new automated test that will be executed in future regression tests, if the bug returns it will be caught by the test.

The negative side of this pattern is when the bug happens in a scenario that is hard to represent in automated tests. For instance, a bug in the synchronization mechanism of a parallel processing software, may be hard to provoke in an automated test, and would be easier to fix directly. Other problems related to non-functional requirements also fall in this scenario.

For bugs hard to simulate in the current test suite, the developer might explore other testing approaches. For instance, a error in a thread synchronization might manifest by performing a load test. It is important to state that the testing approach used in these cases may not be fully automated.

* * *

As with **API Definition**, this pattern usually is a starting point in a TDD session. After the first test, if there is any other scenario to be implemented, a **Differential Test** should be introduced and a new TDD cycle begins. It is pretty common for a **Bug Locator** to define an **Exceptional Limit** for a scenario that was not previously predicted.

In a blog post (Sobral 2012), the application of this pattern is described. The developer found the bug motivated by a piece of data that was not being produced by the software. He described that it uses an automated test to reproduce the bug and then find where it is located.

In Alura (Caelum, 2012), an e-learning system, the pattern is always applied. Every time a bug happens in production, the team is expected to write an automated test to reproduce the bug, and then fix it.

O Esfinge QueryBuilder (Guerra 2014) a bug was found during integration testing. This bug was about a parameter that was not being included on a query. In order to locate the bug, a test was introduced on the unit testing of the class that generated the query, however surprisingly the test passed. The next guess was the class that generated the parameters, and a new test was introduced in its unit tests. This test failed, revealing that there was a collision on the name given to the parameters in that case.

3. Pause for Housekeeping

Also Known as **First Refactor and Then Add Functionality, Prepare for Changes**

Before adding new functionality to a class, it should be clean and prepared to receive it.

The TDD technique states that the simplest solution should always be adopted in order to make the current test suite to pass. However, sometimes, the simplest solution for the previous implementations, makes impossible to implement the next requirements.

*   *   *

How to proceed with the implementation when the current class solution is not suitable for the next functionality?

When a developer knows the functionality he needs to implement in a TDD session, it is very improbable that he will adopt a solution that is completely incompatible with the next requirements. However, new unexpected requirements can arise from customer feedback or in the next iteration that will make the current solution unsuitable. An example of this scenario occurs when the current data structure is not enough to store all the information needed by the next requirement.

Searching always for the simplest solution for the current requirements, opens the possibility to reach a design "dead end", and the need to search for another solution that will be suitable for the next ones.

If the developer tries to add new functionality that will demand huge changes in the class, he creates risk to breaks the previous tests by making the current one to pass. When these changes are big, it is hard to know exactly which part was responsible for the failure.

Therefore:

Ignore temporarily the test that represents the new functionality, and refactor the current code to be more suitable for its needs. Then, reintroduce the test and follow the TDD cycle based on the most suitable step pattern.

This pattern proposes the division between the refactoring task and the introduction of new functionality. During the first step, it is advisable to skip the test that introduces the new class scenario to focus only on making the previous test suite to work on the refactored solution. For instance, in JUnit that can be done by adding the @Ignore annotation on that test. That will avoid the "psychological pressure" of seeing a failing test, and sending a message that it is not the focus to make it pass right now.

The refactoring should be done in small steps in order to make the test suite to pass between their implementation. This practice will avoid errors happening after a change that involved several code changes, making hard to identify which modification was responsible for it. After the refactoring, the test is reintroduced in the test suite and the TDD process will follow its regular path.

This situation often happen when it is necessary to change the data structure used internally by the class. For instance, consider the class presented in Listing 2, which uses a list to store the items of a shopping cart. Imagine that a future requirement to access the items based on their ID make this data structure not suitable for the next steps.

Listing 2 - ShoppingCart using a list as data structure

```
public class ShoppingCart{
      private List<Item> items = ...;

      public void addItem(Item item){
            items.add(item);
      }
      public double total(){
            double total = 0;
            for(Item i : items){
                  total += i.price() * i.qtd();
            }
            return total;
      }
}
```

To perform a **Pause for Housekeeping**, the developer should introduce the new data structure in small steps in order to make the tests still run between them. The next listings presents a path that can be followed to perform this refactoring: [a] introduce the new data structure (Listing 3); [b] update the new data structure when data is inserted and updated (Listing 4); [c] substitute the usage of the old data structure for the new one (Listing 5); [d] remove the old data structure (Listing 6).

Listing 3 - Refactoring step 1: add new data structure

```
public class ShoppingCart{
      private List<Item> items = ...;
      private Map<String, Item> itemMap = ...;

      public void addItem(Item item){
            items.add(item);
      }
      public double total(){
            double total = 0;
            for(Item i : items){
                  total += i.price() * i.qtd();
            }
            return total;
      }
}
```

Listing 4 - Refactoring step 2: add logic in data insertion and update

```
public class ShoppingCart{
      private List<Item> items = ...;
      private Map<String, Item> itemMap = ...;

      public void addItem(Item item){
            items.add(item);
            itemMap.put(item.getId(), item);
      }
      public double total(){
            double total = 0;
            for(Item i : items){
                  total += i.price() * i.qtd();
            }
            return total;
      }
}
```

Listing 5 - Refactoring step 3: changing code that uses the data

```
public class ShoppingCart{
      private List<Item> items = ...;
      private Map<String, Item> itemMap = ...;

      public void addItem(Item item){
            items.add(item);
            itemMap.put(item.getId(), item);
      }
      public double total(){
            double total = 0;
            for(Item i : itemMap.values()){
                  total += i.price() * i.qtd();
            }
            return total;
      }
}
```

Listing 6 - Refactoring step 4: remove old data structure

```
public class ShoppingCart{
      private Map<String, Item> itemMap = ...;

      public void addItem(Item item){
            itemMap.put(item.getId(), item);
      }
      public double total(){
            double total = 0;
            for(Item i : itemMap.values()){
                  total += i.price() * i.qtd();
            }
            return total;
      }
}
```

To finalize the example, it is important to state that the tests should be executed after every refactoring step. This practice will give confidence that the behavior does not change on it and he can proceed to the next one.

A positive consequence of this pattern is that the class solution evolves to a more suitable internal structure, and by running the tests it is possible to make sure that it is compatible to the previous test scenarios. By doing that, it is possible to avoid introducing functionality and restructure the code at the same time.

Despite an eventual **Pause for Housekeeping** is natural on a TDD process, a frequent need of it is a hint that something is not right. If you need big refactorings on the same point of the system frequently, may be it is a sign that a better understanding of that part of the domain is necessary.

* * *

This pattern can be necessary when a **Bug Locator** or a **Differential Test** is introduced in the test suite, when the production class solution is not suitable anymore.

When the **API Definition** pattern is applied, usually a trivial solution is used in order to make the test pass and continue the development with a new **Differential Test**. In the next tests, it is common to change the data structure and the previous solutions until having a significant amount of implemented functionality. When the amount of functionality is small, is is safe to implement directly the functionality without making a **Pause for Housekeeping.**

In the development of Esfinge QueryBuilder (Guerra 2014), an implementation of the pattern generated a query for the database while it receives the method invocation. The next requirement makes the query to be generated dependent on the next call, which makes unsuitable the query generation on the fly. Based on that, a was made to refactor the solution in order to store the information and then generate the query in the end. After that, the new functionality was implemented.

During the development of MetricMiner (Sokol, 2013), we needed to implement a highly flexible way to calculate different code metrics for the same source code. As soon as we finished it, the next requirement was to implement metrics for a group of files. Based on that, we decided to stop the implementation and refactor the solution to create two different subsets of metrics. After that, we continue the implementation.

4. Conclusion

This paper finished the documentation of the initial set of patterns of the TDD Steps Pattern Language**.** This pattern language does not intend to be complete, and it is expected that other patterns complements and enhance it in the future. However the intent was to capture the current knowledge about designing by using TDD.

References

Astels, D. 2003. Test-Driven Development: A Practical Guide. Second edition, Prentice Hall.

Beck, K. 2002. Test Driven Development: By Example. Addison-Wesley Professional.

Caelum. Alura, E-Learning System. http://www.alura.com.br, 2012.

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.

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

Guerra, E., 2014 . Designing a Framework with TDD: A Journey. IEEE Software, v. Jan/Fe, p. 9-14.

Martin, R. 2006. Agile Principles, Patterns, and Practices in C#. First edition, Prentice Hall.

Sobral, D. 2012. Bugs, TDD and Functional Programming, available at http://dcsobral.blogspot.com.br/2012/09/bugs-tdd-and-functional-programming.html

Sokol, F. Z., Aniche, M. F., & Gerosa, M. A. (2013, September). MetricMiner: Supporting researchers in mining software repositories. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on* (pp. 142-146). IEEE.