Preparing for a Test Driven Development Session

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

Test-driven development (TDD) is a development technique used to design classes in a software system by first creating tests before implementing the actual code. However, even before you start creating tests, there are some preparation tasks that the developer should do. This involves gathering information about the class(es) that will be worked on along with how the test(s) should be created. This paper presents some patterns that represent practices that should be applied before starting a TDD session.

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., Aniche, M. and Gerosa, M. , Yoder, J.. 2014. Preparing for a Test Driven Development Session. Proceedings of the Conference on Pattern Languages of Programs (PLoP). September 2014, 11 pages.

1. INTRODUCTION

Test-driven development (TDD) is a technique in which the tests are written before the production code (Beck 2002). With TDD, the development occurs in cycles, comprised of the creation of an automated test, an update of the software to make the test pass and possibly a code refactoring to improve the solution. A set of TDD cycles in a continuous time box with the goal to develop a given functionality is called TDD session. 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 order to improve class design (Beck, 2002; Martin, 2006; Astels, 2003).

This paper continues to document a series of patterns that can be used by developers that are using TDD as a design technique. The two preceding papers documented respectively the basic TDD step patterns (Guerra 2012) and patterns for designing object dependences (Guerra et al. 2013a). Both papers present patterns used during the TDD session to move forward on class development. This paper focuses on practices that should be applied before starting a TDD session, preparing the developer for the following development. Before starting any TDD session it is important to know the **Functionality List** of the scenarios for the software to be created, **Understand the Class Role in Architecture** of any classes that will be modified, **Know your Neighborhood** of any related or collaborating classes, and **Choose Your Weapons** for the best tools to implement the test suitable to the class functionality. This paper presents a study based on the authors' experiences with TDD and some documented TDD experiences to identify the patterns whose names are highlighted on the previous sentence.

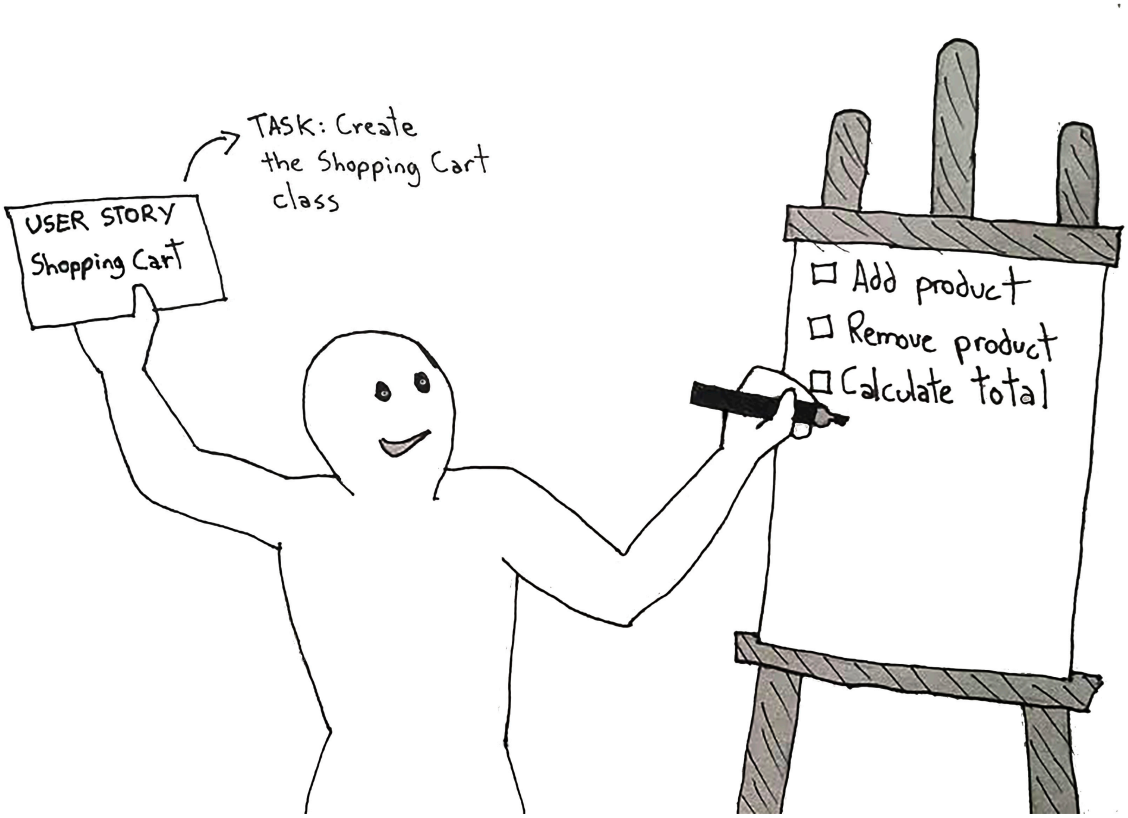
The target audience for these patterns is developers with knowledge on unit testing that understand TDD basic concepts, and want to refine and understand better the TDD process. The patterns consider TDD as a design technique, and not only used by development after a previous design. It includes considerations of how the class being developed by TDD relates to the architecture and to the other classes around it. Since TDD is an agile technique, the practices documented here do not need to part of an explicit and formal process, but they should be part of a lightweight and implicit process.

The pattern format used is based on the classic Alexander format. It starts with the pattern name and some alias, followed by a picture that illustrates the pattern idea. The intent of these images is to present a strong

metaphor that helps the reader to memorize its core idea, and not to be something that fully represents the pattern. The first section is the pattern context, where it describes a scenario where the pattern is applicable. Right after the context is a 3 star (*) divider followed by a bold problem statement. After the problem statement is an explanation of the forces. This is followed by, the word "Therefore:" which is immediately followed by a solution statement, also in bold, and an explanation of how it could be implemented and its positive and negative consequences. After the second divider, there is an explanation about some related patterns and, in italics, a description of some known uses.

2. FUNCTIONALITY LIST

Also Known as **To-do List, Class Checklist, TDD Session Roadmap**.



Based on a task that is part from a User Story, create a list of test scenarios that the created class should execute.

Agile methodologies usually use a high level description of system requirements, such as user stories. These User Stories are usually brock in tasks that captures a development task that should be done in order to implement it. The tasks give an overall vision of what class or classes should be developed, but they don't describe the details and scenarios that should be handled in the implementation. It is expected to collaborate and discuss with the client or product owner to better understand the details of what should be implemented. Imagine the user's story: "*As a hotel receptionist, I want to book a room so that it is reserved for a customer*"; it is somewhat clear about what the system should do, but it clearly does not contain all the details about what rules should be implemented.

* * *

A developer can feels lost if he or she doesn't know where to begin, how much work it is and what exactly needs to be done to implement a task of a User Story. Additionally, the developer might also start working on a non-relevant or incorrect functionality, which can certainly be a waste of time. What scenarios or references can a developer use or be provided to help understand, discuss and record this details of the functionality that needs to be implemented?

There are different types of classes in a system, such as some that handle business rules, and others that deal with technology and non-functional behavior. In order to implement a user story, there will be tasks for the development of different kinds of classes. The stakeholder or the client does not know what kind of scenarios a class should be able to handle. They rather focus on the specific scenarios that are important to the system.

A TDD session should be objective and have a development rhythm. It also should be continuous and fit in a viable time box. It is not desirable for the developer to be lost in the middle of a session asking himself questions like: *"Am I done yet?" or "What should I do now?".* He/she should start the development knowing the functionality and scope of the classes that he is going to develop using TDD.

Since TDD is considered an agile technique, it is desirable that the artifact used to express granular tasks is lightweight. However, it should have enough information to enable discussions with the team and the client when needed.

A long and well-described detailed reference of the tasks to implement takes time to produce and can get the developer lost in the details. However, without knowing what to do, the developer might not implement the solution correctly.

Therefore:

Create a checklist with the scenarios that the class or classes that are part of the current TDD session scope should be able to handle. This can be used as a lightweight reference for the tests to be created.

If the developed class should implement application business rules, the developer can talk to the product owner or customers to better understand what kind of business scenario is expected for that class to handle. However, if the class is a helper class or should handle non-functional concerns, it is possible to extract from other classes and from system's needs what it is expected from that class.

Based on this information, the developer should generate a list of scenarios that the class should be able to handle. It is important for all of them to be good test scenarios. This list can be created in any way that is comfortable to the developer. It can be put on a sheet of paper, saved in a file such as a spreadsheet or a simple text document or even in the IDE itself. Even if the developer does not write it, should be clear for him the scenarios that the class needs to perform. Figure 1 presents an example using Eclipse to create and control the task list.

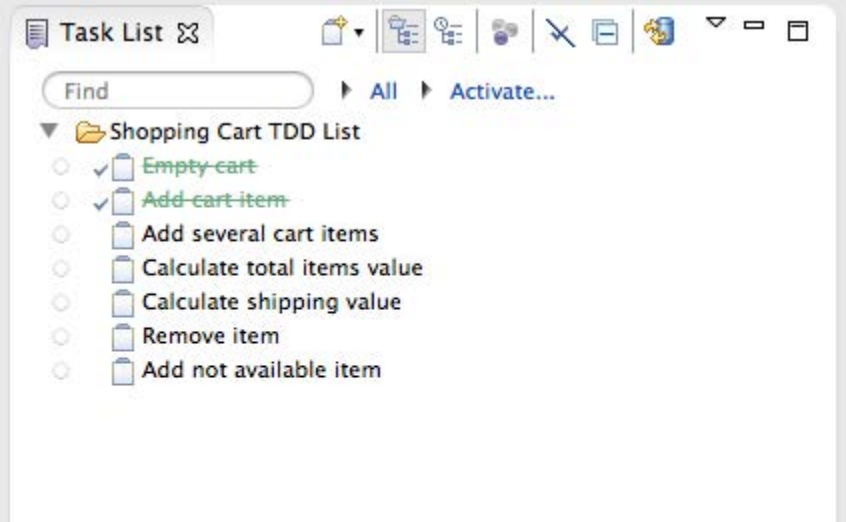


Fig. 1. *Keeping the task list in Eclipse IDE.*

The developer should not try to spend much time brainstorming all the possible scenarios. She should do a simple list and update it during the TDD session. As soon as she perceives that a scenario was not included, she adds it to the list. The list should be enough to determine the scope of the TDD session. Usually new tasks that are added to it reflect alternative scenarios on the requirements expressed on the list and not new functionalities.

It is important to state that it is still possible to forget something when doing the list. In other words, it is not possible to guarantee that the session will start with the developer knowing the overall scope. The developer should not spend much time trying to get "all" the details. It is rather an iterative process, where any missing features that appear through client feedback can be added to the system in a future TDD session.

Returning to the example of the user's story "As a hotel receptionist, I want to book a room so that it is reserved for a customer", an initial checklist for the business class might include: (a) book a free room; (b) try to book an occupied room; (c) try to book a room that is occupied in part of the period; (d) try to book a room when the hotel is full; and e) successfully book a room that is available during the time period. Observe that each item can be "translated" to a test, according to the specific requirements. Note that this doesn't describe the full behavior that should be implemented, which can be discussed with the product owner or client and formalized in the automated tests.

* * *

To create the task list for a single class, it is important to **Understand Class Role in Architecture**. Based on this, it is easy to determine what kind of functionality this class should handle. When you **Know your Neighborhood**, any collaborations needed by the class clients can be used to create the task list.

The TDD session can include the development of a single class or more than one class, depending on the scenario, class characteristics and the system architecture. The patterns for designing object dependences using TDD (Guerra et al. 2013a), namely **Dive Deep**, **Mock Complexity**, **Hide Internal Solution** and **Dependence Exposure** can be used as a reference to define which classes should be included in a TDD session. 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.

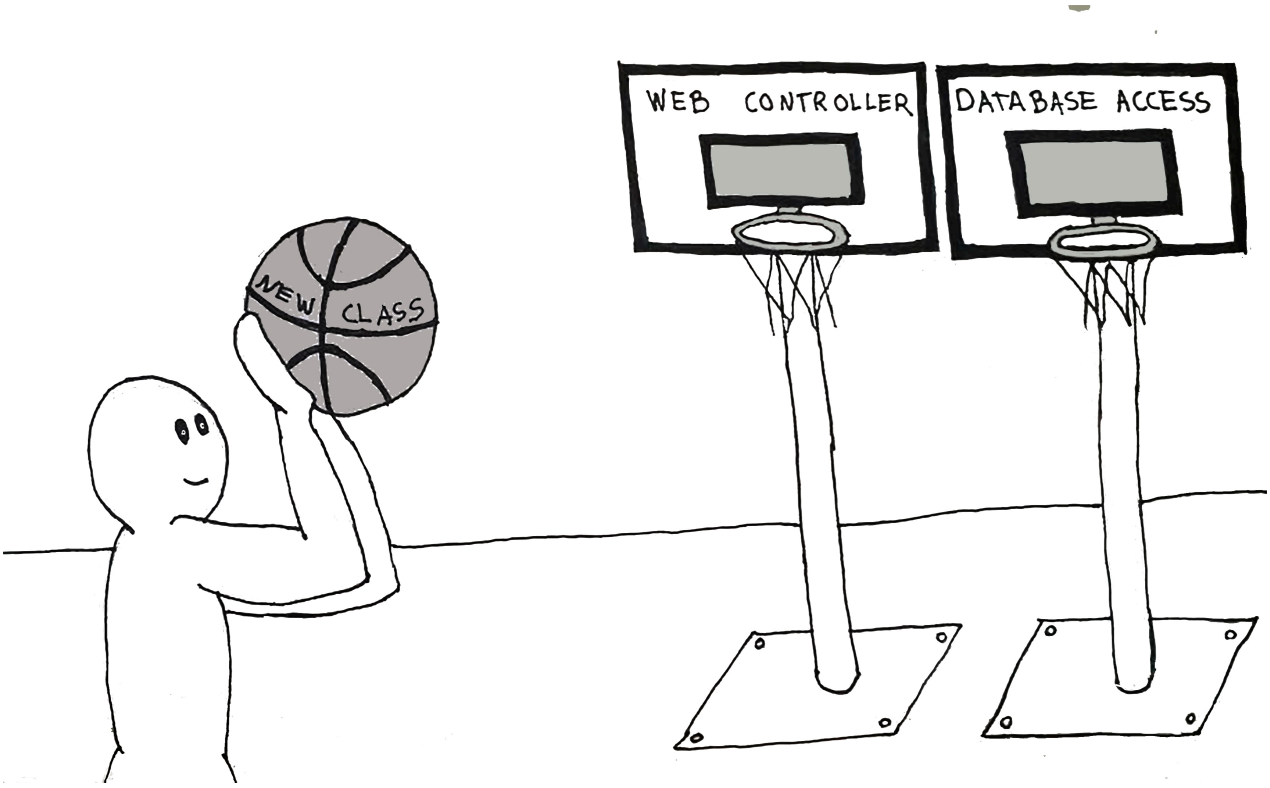
In Test-driven Development by Example (Beck 2002), the first running example shows how to keep a "To Do:" list where there are test scenarios that still need to be added to the class. As long as the TDD session is developed, new items are added into the list. Based on this example, the list is used as a reference for starting the TDD session, but there is not a need for it to be complete before you start.

On MetricMiner project (Sokol and Aniche, 2013), every time a new code metric was about to be implemented, the team wrote a list of a few code examples and how the metric should behave in that specific case. Then, based on the list, developers started the TDD session.

On the Esfinge QueryBuilder project (Guerra 2014), before starting a TDD session, a list of empty test methods was written to represent the scenarios of the next steps. The tests were annotated with @Ignore (JUnit 4 annotation) to not be executed. These methods worked as a functionality list.

3. UNDERSTAND CLASS ROLE IN ARCHITECTURE

Also Known as **Know Your Place, Every Jack to his Trade**



Depending on its role, every class has responsibilities that are expected to perform and others that are forbidden.

The classes in a system have different roles depending on what role they play in the architecture. Some classes are responsible for interacting with resources, such as configuration files and databases, while others are forbidden to perform this task. Before starting the development of a class, the developer should understand the class' constraint.

* * *

How classes interact and are part of an overall architecture is imperative to evolving or maintaining a system. When making changes to a class, how can the developer know which changes or actions are allowed and which ones are forbidden in order to not violate architectural constraints?

The division of responsibility is an important concept used in object-oriented systems. When a class invades the scope of other classes violating the *Single Responsibility Principle*, several bad smells may emerge, such as *Inappropriate Intimacy*, *Feature Envy* or *Fat Method* (Fowler 1999). The information about what class is going to be the client of the developed class and what others it may interact with is important to know as part of the scope of the class being developed.

Additionally, it is not desirable for every class in a system to be able to perform any kind of action. For instance, when there is a database layer provided to interact with the database, the other classes should usually collaborate with classes in that layer to perform this kind of operation, and not do that directly. Limitations in architectures are meant to prevent classes from performing actions that should be responsibilities of other classes or components.

Depending on the development approach, the definition of the architectural constraints can be more or less formal. Some approaches can be: to create a model that reflects the architecture, to define the constraints on a tool enabling their verification at compile time (Merson et al. 2014), to use CRC cards to divide responsibilities or to have an informal agreement of the team based on the adopted reference architecture.

Therefore:

Define explicitly the class role in the architecture or among the other classes, which defines its responsibilities and allowed collaborations.

Depending on the technical risks, the architecture can have different levels of definition before the implementation starts (Fairbanks 2010). When challenging non-functional requirements are detected at the beginning of the project, it is recommended to define early in the process how the architecture will handle these requirements. When this kind of requirement is not present, the implementation can start without an explicit architectural definition. However a lot of development is usually based on a reference architecture that has some predefined roles.

There might be a little upfront design of the architectural, or none if you follow reference architecture. It is important to have whatever is considered enough for the given requirements. However, it is not recommended to develop software without a minimal reference of responsibility divided among the architectural layers or the architectural roles. Based on this definition, the class that is being developed should be positioned in the architecture and created based on the role that it will play.

The need of new classes or even new roles on architecture can be noticed during a TDD session or a refactoring. When that is the case, the architecture should be evolved and new definitions can emerge by necessity. It is important to consider the previous defined architectural roles as a reference, but the developer should not be completely stuck to them.

For instance, imagine that a system will use the Java EE reference architecture and does not have an explicit architectural project. If the developed class plays the role of a "managed bean", it is known that it is a controller and it is responsible to redirect page flow and manage the user session information. However, if this class is a "stateless session bean", it is known that it should implement business rules but it cannot retain session data as an instance variable.

* * *

Understand Class Role in Architecture can be used in the identification of the Functionality List, since it will help to determine the class scope. The architecture role can also help when you **Know your Neighbourhood**, specially when a class in the architecture should interact with another one, such as a business class that needs to use a **Data Access Object** (DAO) to access the database.

Based on the role of the developed class, it is possible to know what kind of resource it needs to interact with for testing. Examples include access to databases, interaction with the file system and invocation of

remote systems. Based on that, it is possible to **Choose your Weapons** appropriately to use a tool that can help to create the kind of test needed.

The architecture definition can use **Continuous Inspection** (Merson et al. 2014) to verify if the developed code obeys architecture constraints. This way, if the class role is not implemented appropriately, the developer can have a fast feedback on this.

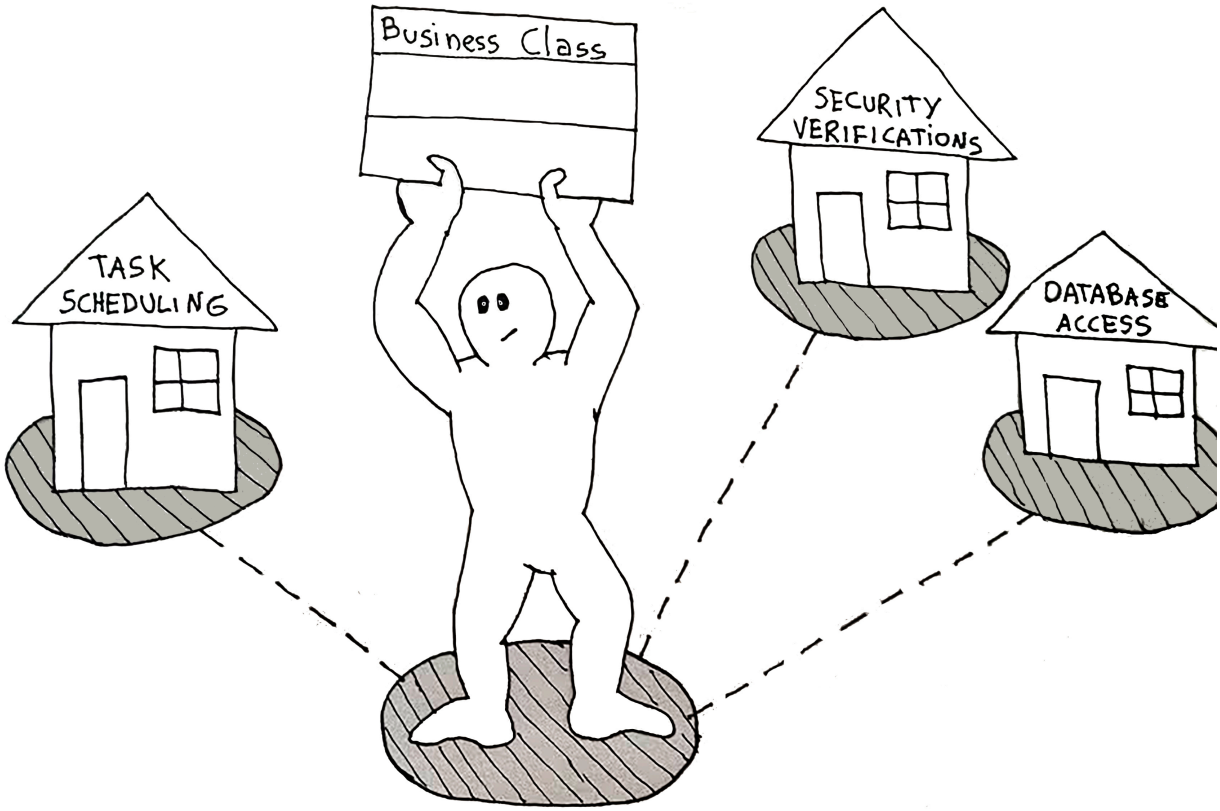
In the development of Esfinge QueryBuilder (Guerra 2014), the role of the classes in the framework were explicitly defined, although there was no explicit documentation to define them. The roles were based on extensibility requirements, which defined what characteristics should be extensible. Because of that, there was an explicit decoupling between certain classes. These definitions were used in the TDD sessions to determine what mocks should be created.

In MetricMiner (Sokol and Aniche, 2013), since extensibility was a requirement, most of the interfaces were created during TDD sessions. The tests defined the contract between classes and how they should be extended. So this pattern was used to separate the role of the class that should handle the main functionality, from the interface that defines the extension points. If a class implement the role of a the class that implements the extension point, this class needs to implement the interface previously defined.

In project LEONA, Transient Luminous Event and Thunderstorm High Energy Emission Collaborative Network in Latin America, the application developed used a reference architecture defined in an initial user's story implemented by the team. After this explorative story, the architecture was defined and it was also defined how each type of class should be tested.

4. KNOW YOUR NEIGHBOURHOOD

Also Known as **Identify the Collaborations, Understand your Surroundings**



Understand the surroundings to know what classes will use it and will be used by it.

.

The methods and the API provided by the class being developed depend on the needs of its clients. In other words, it is dependent upon the classes that would use it. On the other hand, the developed class can also use the collaboration of other classes that can help fulfill its requirements. Other system classes can have influence on how the current class should be implemented.

* * *

Any interesting system has many classes collaborating and interacting to solve requirements. When evolving classes or adding new classes, how can the developer make sure what should be changed in the specified class and what part of functionality should be delegated to other classes?

A system is usually composed by a combination of software elements that collaborate to implement the functionality. However, a TDD session usually focuses on the development of a single class or of a small set of classes. Consequently, in order to know the piece of functionality the current class should have, the developer should understand the contribution of the other classes.

When one class is being developed, the other classes that interact with it might already be implemented or might not. In the case where the client class is already implemented, the interface that this class should implement is already defined based on its requirements. In this case, since the dependent class is already developed, the class being developed will have a well-defined interface of the services that it can delegate to that class.

Therefore:

Identify the classes that are related to the current class under development, identifying the ones that have already been implemented and how they collaborate and consider their existence in the creation of the tests.

Since TDD is a design technique, it is relevant to know what has already been developed and what should be designed during that session. That can includes the interface of the class itself or the interface of one of its dependents. To know what classes interact with the current class, it is important to determine how the tests should be created.

When a design pattern is being applied, it can help to determine how the class interaction is supposed to be and what is the role of each one. Patterns like **Model-View-Controller** and **Data Access Object** create layers with well-defined purposes (Fowler 2002), which can be helpful to divide the responsibilities among the classes. Other patterns, like **Observer** (Gamma et al. 1994), define a clear communication protocol between the involved classes, which also helps the definition of how classes should interact.

If something is already implemented, it does not mean that it cannot be changed during the TDD session. However, an existing class should be changed through refactoring, making sure that the existing tests are still green. Then the new tests of the other classes can consider the new version of that class.

Depending on what is already implemented, the developer may have difficulty in creating tests based on classes that do not yet exist. In such cases, the developer can consider changing the implementation order or using Mock Objects. By developing the neighbor class using TDD, the developer can gain more knowledge that can help in the development of other surrounding classes.

* * *

When the developer **Understands Class Role in Architecture**, he understands the class limitations according to the architecture and knows which kind of classes it is allowed to interact with. A further step is to **Know your Neighbourhood**, defining more specifically to which classes this interaction will actually happen, even if they do not exist yet. When the interactions are defined, they should be consistent to the class role constraints.

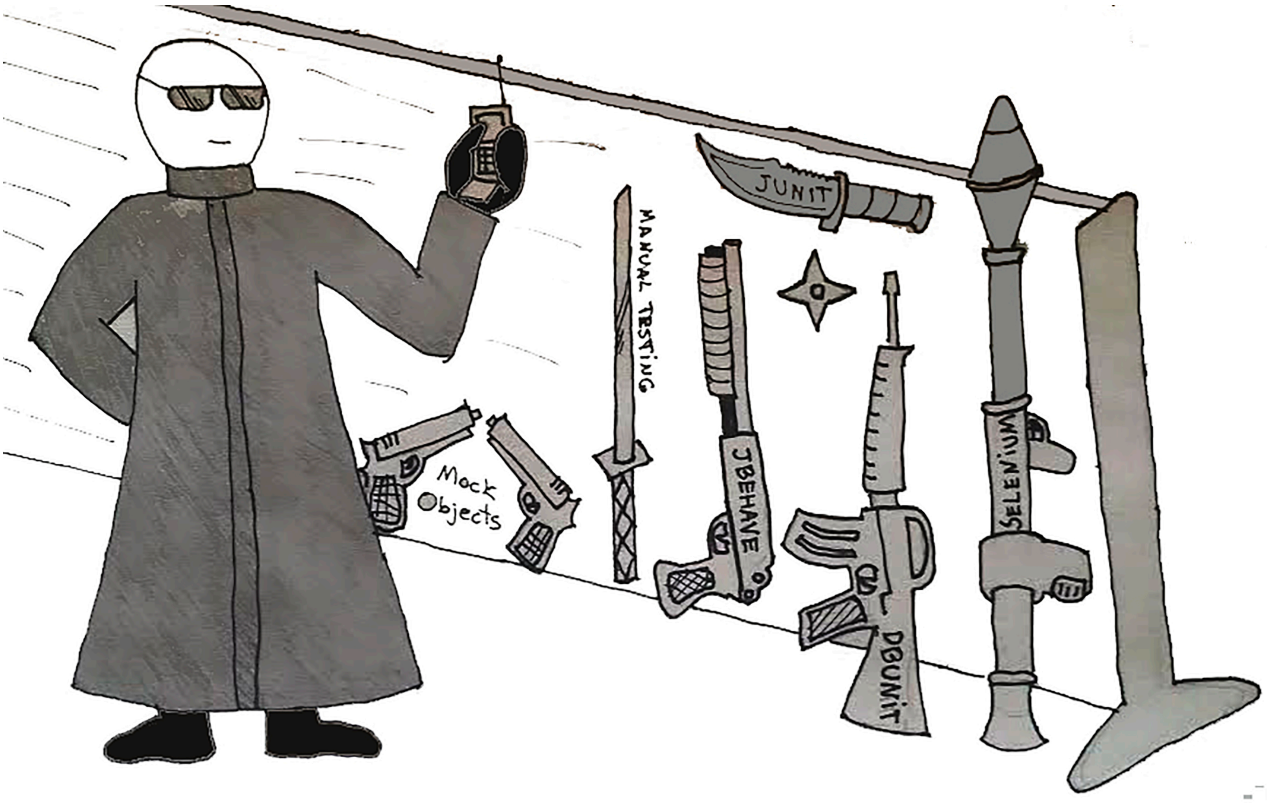
The patterns for designing object dependences using TDD (Guerra et al. 2013a), namely **Dive Deep**, **Mock Complexity**, **Hide Internal Solution** and **Dependence Exposure** can be used as a reference of how these dependents can be further defined during the TDD section. The present pattern only states that you should think of this division before starting the TDD session.

In the development of the system SADE (Perillo et al. 2011), based on the role of business classes on the architecture, it was known that they should interact with DAOs responsible for the persistence. So, based upon class needs, the necessary DAOs to handle the model classes used by the business class were identified before starting to develop the unit tests. Then, while developing the tests it was known that they should be mocked.

During the development of Alura (Caelum, 2012), every time the developer needed to access the database, the right DAO class had to be identified prior to the test implementation. The opposite also happened: when the software had to communicate with external systems, such as Paypal, and no code has already be written to do this, the interface emerged while writing the tests. Then, after the testing and implementation of the main class was done, the concrete implementation to the PayPal interface was written.

5. CHOOSE YOUR WEAPONS

Also Known as **Use the Right Testing Tools**



When you have a complicated testing task, you should choose the right tools to perform it.

TDD is a development technique in which the developer usually creates the test as the first step. For classes that handle concerns that are hard to test, such as database access, remote calls or parallel computing, the necessary time to create the tests can be an obstacle for the TDD adoption.

* * *

Some scenarios can be very hard to test such as involved GUIs or those that involve services and database setups. How can we enable a fast start and evolution of a TDD session for classes that involve hard testing concerns?

Tests that take much time to create, or involve complexity related to the environment that the class should execute, can compromise the practice of TDD. When a developer gets stuck for technical reasons, she can be demotivated for using TDD for the creation of that class. Mock objects, for instance, can be easily created for small interfaces, but can be very hard to work with for more complicated interfaces.

Other developers independent of TDD have faced a lot of these test difficulties. There are several tools that can be helpful in providing additional functionality to help test classes in different kinds of environments. Examples are: DBUnit for persistence; JMock, Mockito and EasyMock for mocking; and ClassMock for classes that uses reflection.

Therefore:

Identify the concerns that your tests will need to handle, and choose an appropriate tool to create and/or support these tests.

The first step is to identify the issues that can make the test hard to create, and define some requirements for the test tool or framework. These requirements can arise when the first test is being created, especially to perform actions in the test setup to create its scenario or verifications after the class execution. For instance, if you define that a test will be hard to create because the tested class will need to access a remote service, the test tool to choose in order to make this task easier should be able to start a server and deploy a service for testing purposes. Then, the first test should be created to validate if the tool is really suitable for the test that needs to be created; i.e. simply create a server, deploy a service, and test if the service is available.

When there is no tool available, the developer can create the first test as a spike solution, verifying the best approach to implement it. After that, the common features can be extracted to a library or testing framework. These might be utility classes that are developed in order to help with these kinds of tests. When there are other classes that should handle the same concern, this approach will make these support classes more valuable. In other words, when there is no tool available, a new one can arise from the implementation of the existing tests to help with the implementation of others. Of course that creating a new tool can cost time and the team should evaluate if the reuse benefit will worth spending the time building it.

To exemplify how a specific testing tool can emerge from a project, imagine that a test needs a file on a given format to be executed. For the first test, it was created an internal method that generates the file, setting the needed information and putting random data on the other fields. It this same testing functionality is needed on other classes, this method could be extracted to a common test library. Further, if some variations of this same functionality are needed by other tests it can be evolved in a testing tool for the project. The team should evaluate if the effort worth based on the benefits that it will generate.

* * *

Understand Class Role in Architecture helps to identify the concern that the test class will need to handle. After choosing a tool to create the first test for the class in a given role, other classes in the same role can follow the same approach.

In the development of Esfinge Comparison (Guerra et al. 2013b), it was identified the need to use classes with different structures for several test cases, because its behavior should vary according to the parameter class structure. Based on this need, the framework ClassMock (Guerra et al. 2010) was used to generate dynamic classes for each test, making them less verbose and easier to develop.

In MetricMiner (Sokol and Aniche, 2013) there was a need to facilitate the writing of the source code that was about to be analyzed by the code metrics. Because of this need, a class was written to make the creation of all the fake code easier. This is an example of a testing tool creation that is more specific for the project being developed.

6. ACKNOWLEDGEMENTS

The authors thank José Maria Guerra for helping to give an appropriate treatment to the pattern pictures. We thank Christian Köppe for being our shepherd. His help was fundamental during the development of this paper. We also thank all the participants on the Writers Workshop that provide an important feedback. We also thanks for the essential support of FAPESP through the project "*LEONA - Transient Luminous Event and Thunderstorm High Energy Emission Collaborative Network in Latin America*", proc. #2012/20366-7.

APPENDIX A - SHORT DESCRIPTIONS OF THE TDD STEP PATTERNS

- **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.

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, 2012. Alura. E-learning platform. http://www.alura.com.br. Last access on May, the 1st, 2014.
FAIRBANKS, G. 2010. Just Enough Software Architecture: A Risk-Driven Approach, Marshall & Brainerd.
FOWLER, M. 1999. Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional.
FOWLER, M. 2002. Patterns of Enterprise Application Architecture, Addison-Wesley Professional.
GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J., 1994, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional.
GUERRA, E. 2014. Designing a Framework with TDD: A Journey. IEEE Software, v. Jan/Fe, p. 9-14.
GUERRA, EDUARDO ; YODER, J. W. ; ANICHE, M. ; GEROSA, M. . Test-Driven Development Step Patterns For Handling Objects Dependencies. In: 20th Conference on Pattern Languages of Programs, 2013, Monticello, IL.
GUERRA, E.; ALVES, F. ; KULESZA, U.; FERNANDES, C. 2013. A reference architecture for organizing the internal structure of metadata-based frameworks. The Journal of Systems and Software, v. 86, p. 1239-1256.
GUERRA, E. M. . Basic Test Driven Development Step Patterns. In: 19th Conference on Pattern Languages of Programs - PLoP 2012, 2012, Tucson.
GUERRA, E. M. ; SILVEIRA, F. ; FERNANDES, C. 2010. ClassMock: A Testing Tool for Reflective Classes Which Consume Code Annotations. In: Workshop Brasileiro de Métodos Ágeis (WBMA 2010), Porto Alegre.
MARTIN, R. 2006. Agile Principles, Patterns, and Practices in C#. First edition, Prentice Hall.
MERSON, P. ; YODER, J. W. ; GUERRA, E. M. ; AGUIAR, A. 2014. Continuous Inspection: A Pattern for Keeping your Code Healthy and Aligned to the Architecture. In: AsianPLoP 2014 - 3rd Asian Conference on Pattern Languages of Programs, Toquio.
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, F. ANICHE, M. MetricMiner: Supporting researchers in mining software repositories. IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2013.