TDD with Mock Objects: Design Principles and Emergent Properties

Luca Minudel, 2009-2011
tdd@minudel.it

ABSTRACT

A team began to write code much easier to read, change and extend after adopting the practice of TDD with Mock Objects. And later the team developed the understanding of the design principles with the ability to put them into practice in the code written everyday.

This observation originated the intriguing conjecture that TDD with Mock Objects led that team to write code compliant with S.O.L.I.D. design principles and partially with the Law of Demeter as an emergent property. This originated the second intriguing conjecture that these tangible improvements of the code-base led that team to deeply understand the design principles and their practical applications as a result of a process of coevolution.

This is an exploratory observational study with the goal of understanding the phenomenon observed, identifying relevant variables, turning conjectures into a verifiable hypothesis whose general validity can be comprehensively investigated with a rigorous research and controlled experiments. This study is to recognize the language ambiguities about TDD and the differences between person to person and team to team in the actual practice of TDD that have relevant consequences on the outcome. And recognize that while talking about engineering practices intended for people in professional software production, people and context are relevant variables that matter.

Test-driven development (TDD) is the technique that relies on very short development cycles, every cycle starts writing a failing automated test case and finish with the refactoring of the code [1]. TDD with Mock Objects emphasizes the behavior verification and clarifies the interactions between classes [8], [3] and [4]. Law of Demeter (LoD) is a design principle that promotes loose coupling between objects, encapsulation and helps to assign responsibilities to the right object [7]. S.O.L.I.D. are 5 object oriented principles of class design to write code that is easy to reuse, change, evolve without adding bugs [9]. Emergent property is a novel and coherent structure that arise during the process of self-organization in a complex system [15]. Coevolution is a process where two interdependent systems change together in mutual adaptation [16] [17][18].

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques, Object-Oriented design methods.

General Terms

Design, Verification.

Keywords

Test-Driven Development, Mock Objects, Design, Learning, Emergent properties, Coevolution.

1. INTRODUCTION

The software development team of a leading F1 Racing Team was implementing software for the Formula One Racing Championship. The team was working with a large and complex code-base, with high pressure to deliver as much new features as possible and in very short deadlines.

The team was trained on Object Orientation with the goal of writing code that was easier to understand, change and evolve without adding new bugs. After the training, the style of the code written by the team did not changed significantly.

A year later the team was trained on the job using TDD with Mock Objects.

Here follow the report of the qualitative observations of the team and the code-base and the report of a qualitative experiment made outside the team with small code exercises. Both team members and the developers that voluntarily participated to the experiment are uncontrolled groups acting in an uncontrolled environment.

2. OBSERVATIONS

2.1 Initial training on Object Orientation

During 2006 the team members, divided in two groups and in two different moments, had an intermediate training on Object Orientation.

2.2 After the training on Object Orientation

After this training some of the team members more experienced with the code-base and the application domains proposed some improvements to the design at the level of namespaces and assemblies (intended as the fundamental unit of deployment, versioning and reuse of compiled code like an EXE or DLL file) and a top-down approach to implement these changes.

These ideas were not implemented and so the improvements in the quality of the code produced day by day have not been noticed during 2006.

During 2006 the team was also practicing unit testing, with tens of thousands unit tests running on the automatic build server. The majority of the unit tests were actually more integration tests then real unit tests. For example most of the tests were involving external systems like the database and were involving many different objects and layers at the same time.

The tests suites were slow and some of them brittle. An effort was made using advanced features of commercial mocking tools to mock static classes, concrete classes, classes provided by external libraries and classes instantiated directly inside the class under tests. The code-base overall was hard to test.

2.3 The training on TDD with Mock Objects

In the beginning of 2007 two groups of software developers attended an internal hands-on training on TDD with Mock Objects.

At the beginning of the Sprint one group of team members went into a meeting room. They brought with them one PC, one keyboard, one projector with the screen and the user stories selected for that Sprint. Team members contributed to the Sprint goal with their knowledge of the code-base, the application domain and the technology stack in use.

Two software engineers extremely experienced in the practice of TDD with Mock Objects joined the group and contributed to the Sprint goal with their knowledge of TDD and refactoring on large complex and legacy code-base. They showed how to implement the user stories guided by TDD and mocks, in quick (5-15 minutes) red-green-refactor cycles, constantly discussing together and rotating pairs at the keyboard. At the end of the week the user stories were implemented and accepted by the end users and released.

The week after the team and the two software engineers went back to the office and completed another Sprint. This time team members were working in pair as usual at their workstations and rotating pairs with the two software engineers.

The same experience was repeated with another group and after that the two software engineers joined the team full time.

Team members from both groups immediately appreciated that the production code and the unit tests written during the training session were better than before.

We learned:

- how to use the refactoring tool to extract interfaces, break dependencies [2] and how to inject dependencies into parameterized constructors and in methods arguments,
- how to replace static variables and singletons with more testable code,
- how to wrap third-party libraries,
- how to use a mocking tool to mock dependencies and declare and verify expectations,
- how to test non-trivial objects in isolation,
- how to quickly navigate in the IDE between interfaces and the classes and tests,
- about the practice of avoiding getters and instead using *Smart Handlers* that are Visitor-like objects [6]. However this practice was not followed.

2.4 After the training on TDD with Mock Objects

Team members after the training and after continuing to practice TDD with mocks discussed the effects of this new practice on the code.

For example there were discussions about the parametric constructors used only by the unit tests; discussions about the large use of Interfaces (as intended in Java, C# or like Abstract classes in C++ with only pure virtual functions or like protocols in Smalltalk) defined to enable the mocking of objects; discussions about the larger number of small classes each one with a narrow responsibility; discussions about the use of default constructors or factories or Dependency Injection frameworks; discussions about wrapper created to break dependencies to external libraries and external systems; discussions about the increased use of containment over inheritance; discussions about avoiding the use of static classes and singleton; discussions about the change of the point of view when writing tests with expectations on how objects interacts; discussions about where to use of strict mocks and where instead to use stubs.

The practice of TDD with mocks significantly changed our production code and our test code. We observed and recognized that the result was better code easier to understand, change and evolve. Then we tried to understand which changes were causing the improvements, which changes were just side effects needed by unit tests and which changes were caused by our inexperience with TDD and mocks.

2.5 One year after the training on TDD with Mock Objects

Between 2007 and 2008 TDD with mocks become an established practice for the team. A group of team members were constantly discussing and striving to improve our practice of TDD with mocks, another group were keen on practicing correctly and systematically TDD with mocks and on adopting new improvements proposed by the first group, and finally another group were less interested about the practice still were supportive in maintaining existing tests and in practicing TDD with mocks when pairing with a team member experienced in that technique.

The difficulties with slow and fragile tests suits observed before the training were solved in the new and changed code and in unit tests written after the training. In addition the code written was easier to understand, change and evolve then the code written before, without TDD and mocks.

A group of team members striving to understand the relation between the changes caused to the code-base by the practice of TDD with mocks begun to study S.O.L.I.D. design principles and the Law of Demeter, discussed the relation between the practice of TDD and the adherence to design principles and reached a deeper practical understanding of the design principles and were able to further improve the code produced day by day intentionally removing more violations of the design principles then before.

2.6 Documentation of experience

Between 2009 and 2011 this experience has been documented, reviewed by team members involved, discussed and compared with other experiences, i.e. [21], to search for similarities and differences.

3. HYPOTHESIS

As a result of these observations we were intrigued by the conjecture that code developed with TDD and mock objects tends to conform to the S.O.L.I.D. principles and to the Law of Demeter as an emergent property.

By emergent property we understood this to mean that the tendency to the conformance is obtained without an explicit policy to do so, without training the team on the design principles or without requiring the team to produce code that conforms to the principles. This means that an improved conformance is obtained as a positive unanticipated consequence of applying the practice of TDD with Mocks Objects [15].

The number of violations of the design principles can be measured every time a class is changed observing the code-base. Therefore the positive trend of this number of violations after the adoption of TDD with mocks can be verified. This is a hypothesis that can be verified with observations and also with code metrics.

Team members learned by observing positive effects of the changes in the code induced by the practice of TDD and mocks and this led to a deep practical understanding of the design principles and team members were able to further improve the code produced day by day intentionally removing more violations of the design principles. As a result of these observations we were intrigued by the conjecture that the tangible improvements of the code-base produced by the practice of TDD with mocks led the team to deeply understand the design principles and their practical applications as a result of a process of coevolution.

By coevolution we meant that the better understanding of the design principles and their practical applications in the code written is obtained without training the team on the design principles. This means that an improved understanding of the design principles and their practical applications is obtained as a result of the process of a mutual adaptation of the code-base and the team, where the positive change of the code-base is initiated by the adoption of the practice of TDD with mocks and the change in the team follows with a mutual adaptation process [16] [17][18].

The practical ability to avoid and remove violations of the design principles, even the ones that are not related to the adoption of TDD with mocks can be easily measured in the code-base and tested with exercises, before the adoption of TDD with mocks and after the adoption of the practice. The hypothesis of improvements of the practical ability to remove more violations can be verified with observations of the code-base and also with exercises.

4. EVALUATION OF THE HYPOTHESIS

In order to evaluate the hypothesis, in addition to the evidence that the code was easier to understand, change and evolve, we evaluated the conformance of the code to the design principles by observation, sampling the code we were changing. We found that the code produced was more adherent to the S.O.L.I.D. design principles then before. And we found the code was only partially adherent to the Law of Demeter and this was compatible with what is reported in [8]. Indeed while every access to objects getter was usually wrapped to avoid “train-wreck”, this had not removed all the violations of the Law as instead the delegation of behavior does.

Together with some of the team members we discussed in retrospective about this experience and we reported that initially we noticed in practice that code developed with TDD using Mock Objects was easier to understand and change, we observed in the code the characteristics that made it easier to read and evolve, we learned from these observations and we adapted our coding style to pursue that useful characteristics. Some of the software engineers perceived commonalities in the source codes that were easier to understand and change and autonomously and voluntarily began to study the design principles and apply them in an aware and intentional way.

Then we found that what we reported was explainable with a well known *secondary effect* of the emergent properties called *coevolution*.

5. EVALUATION OF THE HYPOTHESIS IN OTHER TEAMS AND CONTEXTS

Discussing the observations of the experience of the team with other teams and experts helped to identify common misunderstanding and hidden assumptions that so need to be explicitly stated and described as preconditions in order to verify the hypothesis in different teams and in different contexts.

Also factors as people, the requirements, the technology used, the environment where the development happen must be taken into account as possible relevant variables [19] in order to verify the hypothesis in different teams and in different contexts.

5.1 TDD with Mock Objects defined

TDD is generically described with the red-green-refactor cycle, how every phase is actually executed can substantially change from team to team, from programmer to programmer.

The style of TDD with mocks referred here is the one originated in 1999 in the London-based software architecture group and then experimented and evolved in the Connextra team and later also in the London Extreme Tuesday Club (XTC).

It is the one described in the paper presented at the XP2000 conference [8] and the one presented at the In OOPSLA 2004 conference [3] and is the unit testing approach described and explained in great detail in the GOOS book [4].

5.2 Properly trained developers

While here is made the hypothesis about learning and developing a deep understanding of the design principles through a process of coevolution, the ability to practice effectively TDD with mocks is a given precondition. There is no claim here that the practice of TDD with mocks can survive inadequately trained developers.

5.3 The people and the environment

Since no one can be forced to learn a new technique, it is relevant that the people in the team have a purpose to learn TDD with mocks. In the team we had the tests suites that were slow and some of them brittle and we were striving to solve those issues.

The environment was also a relevant variable. There was a high pressure to deliver new functionalities, a volume ten times bigger than the actual capacity of the team. Because of this, only the top priority functionalities were implemented and so they were used immediately after released. Because the short deadlines we had to implement the features incrementally and so after the first release of a new feature the team usually had to reuse, change and extend the code just created or changed in the previous Sprint to extend the feature. And the deadlines were often of 1 or 2 weeks and less often of 3 weeks.

The relevant variables are:

- early feedback from the users: immediately after every feature is released defects and bugs are reported;
- early feedback form the code: immediately after every feature is released its code is often reused and changed and extended and this make it tangible how easy the code just written is easy to change and extend;
- very frequent releases: the feedback loops are really short and so the actions and the outcomes are under the same learning horizon enabling the team to learn from the experience

5.4 No centralized point of control

The code-base was large including a large number of different integrated applications. And distinct autonomous interdependent departments were driving the evolution of the applications.

The lack of a central point of control for the evolution of the system makes it clear that a centralized policy to evolve the design of the code could not be effective [20].

This encouraged the team to investigate other ways as emergent design driven by TDD and mocks.

6. THE EXPERIMENT

To better understand the phenomenon in general, between 2009 and 2011 an experiment was made: some developers outside the team voluntarily accepted to solve small coding exercises and answering to a survey.

The exercises consisted in refactoring some code that had various violations of the S.O.L.I.D. principles and the LoD, with the goal of making the code testable and write the unit tests.

The survey's questions were about the proficiency of the developer in TDD with mocks, in TDD in general and in S.O.L.I.D. design principles.

The solution of the exercises were compared with the level of proficiency declared in the survey and a conversation with the developer followed to clarify possible doubts.

This experiment was conducted with an uncontrolled group and in an uncontrolled environment, the results were qualitatively measured.

The results of the experiment suggest that developers not proficient in TDD with mock, especially the ones that wrote integration or acceptance tests more then real unit tests, removed fewer violations of the design principles. Even the ones that claimed to be proficient in the S.O.L.I.D. design principles. Those developers proficient in TDD but not in TDD with mocks, that wrote real unit tests, removed more violations of the previous group. The group of developers proficient in TDD with mocks removed the major number of violations.

Some of the violations were not removed by any of the participants to the experiment.

7. EVALUATION OF THE RESULTS

When discussing the conjecture that originated this study with other experienced software engineers and TDDers a comment was that the skills and expertise required to design properly an application are vast and cannot be replaced just by applying TDD.

The preparation for this experiment made it very clear that the kind of the design improvement discussed here is the one that relate to the design of the classes, the distribution of responsibilities among different classes and how objects collaborate to each other sending messages at run-tine, and this is consistent with earlier research results [22]. The design at a more coarse grained level that focus on the organization of namespaces, components and sub-systems, domain models and on the definition of a compact & expressive languages to implement features in that domain, is outside the scope of the hypothesis of this study, indeed is more related with Acceptance-TDD.

The results of the experiment showed that developers proficient in the S.O.L.I.D. design principles and very capable of arguing and explaining the principles removed fewer violations of the ones practicing TDD with mocks.

A possible explanation is that the design principles are not specific to a language, a technology stack and a domain, so they are described in general and abstract terms. And the connection between the general abstract description and how to apply them in the code is not given. Because of this, the help TDD with mocks gives to remove violations and write code adherent to the principles make a huge difference.

This huge difference is evident and tangible and helps developers to make the connection between the general and abstract definitions and the practical applications in the code.

8. ANALYSIS OF THE RELATION BETWEEN TDD WITH MOCK OBJECTS, S.O.L.I.D. CODE AND THE LAW OF DEMETER

Here is analyzed how the team practiced TDD with Mock Objects and how this promotes the conformance to the design principle.

TDD with Mock Objects defines [3] [4] [8] ways to write testable code, below these are labeled as *Practice*. For example it tells to pass dependencies in through the constructor. TDD with Mock Objects describes also a set of test code smells in the unit tests code that are related to possible problems in the design of the production code. Below are labeled as *Smell*. For example one smell is a bloated constructor. And for every test smell a list of possible solutions are suggested.

A practice explicitly describes what to do, while a smell requires to the developer a judgment based on knowledge and experience. Indeed a test code smell is a *hint* that something *might* be wrong somewhere in the code under test. It is not a certainty. It is up to the developer to check out the design of the code under test, and based on his/her knowledge and experience decides whether the code actually need fixing, whether can be tolerated or whether is just ok as is [13].

8.1 Open-Closed Principle and Dependency Inversion Principle

The Open-Closed Principle states that classes and methods should be open for extensions and strategically closed for modification: so that the behavior of a class can be changed and extended adding new code instead of changing existing code and many dependent classes.

The Dependency Inversion Principle states that both low level classes (e.g. representing the persistence details or intra-systems communication details) and high level classes (e.g. representing application domain concepts or business transactions) should both depend on abstractions (e.g. interfaces): high level classes should not depend on low level classes. This improves the re-usability of classes and enables the evolution of the existing code with small local changes.

8.1.1 Practice

When writing a unit test with TDD using Mock Objects, a parameterized constructor is added to the class in order to inject all the dependencies, directly or through a factory that can return more than one instance of a dependency and permits to instantiate a dependency later in time. Look [3] at paragraph 4.9.

```
public class MonitoringSystemAlarm
{
  public MonitoringSystemAlarm()
  : this(new TirePressureSensor(), 17, 21) {}

  public MonitoringSystemAlarm(
     ISensor sensor,
     double lowPressureTreshold,
     double highPressureTreshold)
  {
    // ...
  }
```

The point here is that all the dependencies implement their own interface and the interface type is used for the parameters in the constructor. The same holds true for dependencies that are passed as arguments of a method of the class. All this makes it possible to pass a mock object everywhere a real object is expected. This is not a work-around for a limitation of the mocking tool that cannot mock a concrete class, instead this is the deliberate way that TDD with Mock Objects adopts to break dependencies between classes, to make relationship explicit, to promote the coding of classes that are easy to reuse and that can be changed without provoking an unpredictable cascade of many changes. This is how TDD with Mock Objects helps to write classes that adhere to the DIP. Look [4] at chapter 20, paragraph "Mocking Concrete Classes".

8.1.2 Practice

Since with the practice of TDD with Mock Objects almost all the dependencies of a class are interfaces, all these dependencies give the possibility to create new implementations which extend the possible use of the class behavior. E.g. a *logger* class could log on different implementations of *IAppender* interface: file, console or db; a *deposit* class could work with different implementations of *IOnlinePaymentsMethod*: PayPal or Credit cards.

The interfaces and implementations are separate so it is possible to completely substitute anything at any point by providing another implementation of the interface. Moreover the use of interfaces prevents the use of public member variables (aka class fields), and singleton and static variables are discouraged because they are not unit test friendly and mock friendly.

This help to write classes that adhere to the OCP.

8.1.3 Practice

The frequent refactoring during the red-green-refactor cycles of TDD with Mock Objects helps to remove conditionals (i.e. *if* and *switch* statements) and also the conditionals that check for object type (e.g. through C++ Run-Time Type Information or through Java and .NET Reflection).

This too helps to write classes that adhere to the OCP.

8.1.4 Where TDD with Mock Objects doesn't help in the matter of OCP and DIP

A way to adhere to the OCP not directly enforced by TDD with Mock Objects: the use of the *template method* design pattern, call-back functions, events (*publisher-subscribers* design pattern) and policies as sorting criteria delegated to other classes.

A way to adhere to the DIP not directly enforced by TDD with Mock Objects: the use of the *template method* design pattern to encode a high level algorithm implementation in an abstract base class and have details implemented in derived classes. Thus, the class containing the details depends upon the class containing the abstraction. The same result can be obtained with the *builder* design pattern.

8.2 Single Responsibility Principle and the Interface Segregation Principle

The Single Responsibility Principle states that there should never be more that one reason for a class to change: a class should have one and only one responsibility.

The Interface Segregation Principle states that clients should not be forced to depend upon interfaces that they don't use: fat interfaces should be avoided, while interfaces that serve only one scope should be preferred.

8.2.1 Smell

Writing a unit test with TDD using Mock Objects can lead the class under test having a bloated constructor: a constructor that has a long list of arguments used to inject dependencies. This is the smell that the class has too many responsibilities and one suggested refactoring is to break up the class into more classes each one with a single responsibility. Another suggested refactoring for this smell is to package a group of dependencies into a new class that contains them and deals with the related responsibility. Fore more details look [3] at paragraph 4.8 and [4] at chapter 20 the paragraph "Bloated Constructor".

8.2.2 Smell

A unit test with a lot of expectations is a smell that the class under test has more than one responsibility and the suggested refactoring is to extract into a new class a group of those collaborations declared in the expectations. [3] at paragraph 4.7 and paragraph 5.4 and [4] at chapter 20, paragraph "Too Many Expectations".

8.2.3 Smell

When a group of test cases uses the same group of member variables (aka class fields) of the text fixture class, this too is a smell that those test cases deal with a distinct responsibility and the suggested refactoring is to extract from the class under test the responsibility into a new class. For more details look [14].

8.2.4 Smell

When writing a unit test with TDD using Mock Objects it can happens to mock one method call of a dependency (e.g. set an expectation) and at the same time to stub another method call on the same dependency (e.g. set the return value for the method that could be invoked zero, one or many times).

```
[Test]
public void Send_Diagnostic_String_&_Receive_Status()
{
  var mockTelem=mocks.StrictMock<ITelemetryClient>();
  mockTelem.Stub(m => m.Connect());
  mockTelem.Stub(m => m.OnlineStatus).Return(true);
  mockTelem.Expect(m => m.Send(DiagnosticMessage));
  //...
}
```

This is the smell that the dependency might have 2 distinct responsibilities. The suggested refactoring is to split the two responsibilities into two different classes.

8.2.5 About those smells

In all those cases, after breaking up the class, the result is new classes that adhere to the SRP. The class interface too is split into distinct interfaces that will adhere to the ISP [4] chapter 20, paragraph "Mocking Concrete Classes". The interfaces obtained with this process often mimic the implicit public interface of their class, so as a result you see pairs of things, like *ITelemetryClient* and *TelemetryClient*.

8.2.6 Practice

Another way to put too many responsibilities in a class is the abuse of inheritance. TDD with Mock Objects encourages the use of composition over inheritance and this prevents the abuse of inheritance and also the violation of the SRP caused by the abuse of the inheritance. For an example look at [3] paragraphs 2.1, 3.3.1 and 3.7.

8.2.7 Where TDD with Mock Objects doesn't help in the matter of ISP

A way to adhere to the ISP not directly enforced by TDD with Mock Objects: even when an interface mimic the implicit public interface of a class that already has a single responsibility, sometimes there can be chances to further break up the interface into distinct interfaces aimed at different clients, with the goal of eliminating an inadvertent coupling between clients and between DLLs. This decreases the number of dependencies and the number of recompiles needed after a change. And the result is a better conformance with the ISP.

8.3 Liskov Substitution Principle

The Liskov Substitution Principle states that methods that use pointers or references to a base class must be able to use instances of derived classes without knowing it: all the derived classes must honor the contract defined by the base class.

8.3.1 Practice

A method implementation that checks for the object type of the actual argument (e.g. through C++ Run-Time Type Information or through Java and .NET Reflection) violates the LSP as well as the OCP. With the practice of TDD with Mock Objects the bar become red when changing the method parameter type from the base class type to the interface type in order to mock the argument. The LSP violation is surfaced by the failing test, and to get a green bar the violation must be removed.

8.3.2 Practice

TDD with Mock Objects and TDD in general change the design of base and derived classes from a process of invention into a process of discovery: first commonalities among different classes are found and then are extracted in a common base class. The commonalities are found after the test is green (red-green) and the duplication is removed refactoring the code (green-refactoring). This prevents many violations of the LSP that can happen when a base class is designed upfront or when classes are derived upfront. Furthermore TDD with Mock Objects promotes the use of

composition over inheritance. For more details and examples look [3] at paragraph 2.1, 3.3.1 and 3.7. This avoids many violations of the LSP too.

8.3.3 Practice

Furthermore, a derived class that overrides a virtual method violates the LSP when it replaces the precondition of the base class method with a stronger one and when it replaces the post condition with a weaker one. This violation can be detected executing the unit tests of the base class also against the derived class. This holds true for TDD and for unit testing in general.

8.3.4 Where TDD with Mock Objects doesn't help in the matter of LSP

All the previous practices prevent or avoid violations of the LSP.

Adherence to the LSP is easier to verify in the context of its clients using the base class and the derived classes. The LSP makes clear that in OOD the ISA relationship pertains to extrinsic public behavior that clients depend upon. The main focus when writing a unit test with TDD using Mock Objects is on the behavior on the Design by Contract, in this case the behavior of the method that is overwritten in the derived class. Look [3] at paragraph 2.1. When there is a violations of the LSP it can be highlighted by some unit tests e.g. when the expectations on the same interface methods on two different tests are inconsistent. It is up to the programmer to notice the inconsistency and finding how to fix the LSP violation. It is also up to the programmer to spot refused Bequest smell and fixing it when appropriate.

8.4 Law of Demeter

The Law of Demeter states that methods of an object should avoid invoking methods of an object returned by another object method, the motto of LoD is "Only talk to your friends" and the goal is to promote loose coupling.

8.4.1 Practice

Avoid the use of getters; replace them with *Smart Handlers* that are Visitor-like objects [6] that are passed to the object without getters. With this practice code tend to conform to the LoD just like when applying the *Tell, Don't Ask* principle. For an example look [8] at paragraph 4.3.

8.4.2 Smell

A single modification in the code that requires changes to expectations in two different tests is a smell that design is broking the Law Of Demeter. This is true especially when the initial modification in the code involves getters. The suggested refactoring is to replace getters, with *Smart Handlers*. For an example look [8] at paragraph 4.3.

8.4.3 Smell

Also a unit test with a lot of expectations with mocks that return other mocks is a smell that the class under test has a responsibility that belongs to another object and the suggested refactoring is to apply the heuristic "Tell, Don't Ask". Fore mode details look [3] at paragraph 1.2, and [4] at chapter 2 paragraph "Tell, don't ask" and at chapter 20 paragraph "What the Tests Will Tell Us (If We're Listening)" and also [5].

9. FINDINGS

The results of the observations, the experiment and the analysis are compatible with the two initial conjectures and lead to identify the preconditions, the relevant variables and the hypothesis that can be tested.

The precondition is that the developers must be properly trained in the practice of TDD with Mock Objects and able to apply it properly as described in [3][4][8].

Relevant variables of the environment, within the software team operates, are:

- early feedback from the users about defects and bugs in the new releases
- early feedback form the code: features are developed and release incrementally so the code just released is immediately reused and changed and extended;
- very frequent releases: every week or two in order to have very frequent feedbacks that enable learning from the practice

Another relevant variable is the pressure and the will to release working and valuable software as fast as possible, and the presence of mentors for the practice of TDD with Mock Objects to support safe experimentations and improvements.

The first hypothesis: the number of violations of the design principles when a class is changed in the code-base decrease significantly more when the team practice TDD with mocks.

The second hypothesis: after the adoption of the practice also the number of violations of the design principles not directly related to the adoption of TDD with mocks (for example the ones describe in the paragraph 8.1.4) decrease progressively more.

The result should be different from team members that don't practice TDD or practice TDD improperly writing using tests that are more similar to integration tests. In that case the number of violations in the code-base is not expected to decrease as much as for the team doing TDD with mocks.

10. DISCUSSION

The analysis here documented about the relation between the practice of TDD with mocks and the design principles is useful to evaluate the conjecture that the improved conformance to the design principles is an emergent property.

Indeed the *Practices* as described in the analysis show that some of the violations of the principles are removed as direct

consequence of those practices, this cause-effect relationship does not indicate an emergent behavior even if this it still is a positive unanticipated consequence. So we can name this a weak emergence.

Ad the same time the *Smells* described in the analysis don't have a direct relation with removing violations of a design principle, it is the result of a judgment based on knowledge and experience of the developer that is developed practicing TDD with mocks. We can call this proper emergence.

The coevolution used to explain the process of learning the design principles and their practical applications when practicing TDD with mocks as well as the emergence used to explain the improved conformance they both arise during the process of self-organization in a complex system. And since team members are humans it is a socially complex system [10][19].

Joseph Pelrine is one of Europe's leading experts on Agile software development, has worked as assistant to Kent Beck in developing eXtreme Programming, is an accredited practitioner for the Cognitive Edge Network, and his work focus is on the field of social complexity science and its application to Agile processes. He suggested the use of the ABIDE model (Attractors, Barriers, Identity, Dissent/diversity and Environment) developed by Dave Snowden at the Cynefin Center for Organisational Complexity and now at Cognitive Edge [11] to search for relevant parameters of the socially complex system. In particular he suggested that the two software engineers extremely experienced in the practice of TDD with Mock Objects that trained the team and then joined the team acted as Attractors in the process of self-organization of the socially complex system.

Following the ABIDE model, the practices or TDD with mocks acted as Barriers in the self-organization.

While the frequent feedback from users and the code that define a structure of the interaction between team members and the users and the code contributed to define the Environment where the self-organization had place. This is consistent with research results about iteration and learning [23].

The conjecture reported here that the process of learning is emergent phenomenon has been studied before also by Dr. Sugata Mitra.

Dr. Sugata Mitra, Education scientist, professor of Educational Technology at New Castle University UK and Chief Scientist of NIIT since 1999 with his 'Hole In The Wall' experiments is testing his speculations about education as a self-organising system where learning is an emergent phenomenon [12].

Sphere College in Phoenixville Pennsylvania, and Khabele School in Austin Texas have an educational philosophy that incorporates elements of self-organisation and emergent education.

Here follow comments and quote form experts in TDD and in TDD with mocks that are relevant to this study.

A relevant quote from Steve Freeman: *No technique can survive inadequately trained developers.*

A relevant quote from Nat Pryce: *TDD does not drive towards good design, it drives away from a bad design. If you know what good design is, the result is a better design.*

A relevant quote from Kent Back: *TDD doesn't drive good design. TDD gives you immediate feedback about what is likely to be bad design.*

A relevant quote from Michael Feathers: *writing tests is another way to look the code and locally understand it and reuse it, and that is the same goal of good OO design. This is the reason of the deep synergy between testability and good design.*

11. THREATS TO VALIDITY

Since this an observational study based on observations in an uncontrolled experiment it is not free from overt biases as i.e. in the sampling of the code that has been observed and in the judgment of the code observed in regard to the adherence to the design principles.

There is also the possibility of hidden biases as i.e. lot of tacit knowledge of good design by the observed team.

Since the observations have been documented in retrospective, potentially suffer from the Texas sharpshooter fallacy.

12. CONCLUSION

The observations, the analysis of the relation between TDD with mocks and the design principles and the qualitative experiment are compatible with the conjecture that the practice of TDD with Mocks Objects led the team to write code more conformant to the S.O.L.I.D. design principles and partially to the Law of Demeter.

They are compatible also with the conjecture that the practice of TDD with Mocks Objects led the team to learn and develop a deep understanding of the design principles and their practical applications.

And finally they are compatible with the conjecture that the conformance to the design principle is an emergent property and the learning of the design principle is a process of coevolution.

The qualitative experiment and the analysis of the relation between TDD with mocks permitted to roughly quantify the expected improvement of conformance to the design principles due to the practice of TDD with mocks.

Information and understanding developed with this study permitted to identify preconditions and relevant variables and to turn the conjectures into hypothesis that can be tested in a subsequent empirical software engineering research in other teams.

13. ACKNOWLEDGMENTS

Thanks to Paolo Polce e Gerardo Bascianelli that joined the team and shared with us their knowledge and deep experience on TDD with Mock Objects. Thanks to Antonio Carpentieri and Riccardo Marotti and all the dev team members of the F1 Racing Team for their curiosity to explore new ways of writing code, for their courage to give up old skills for new ones, for their trust and respect that permitted us to engage in discussions, open disagreement and coding experiments and come out with new useful understanding and insights.

Thanks to those members of the XPUG-IT and UGIdotNET Italian community that voluntarily participated in the experiment.

Thanks to the many of you that helped to review this paper, who suggested ideas and improvements and who shared and discussed their own experiences with TDD.

14. REFERENCES

[1] Beck, K. 2002. *Test Driven Development: By Example*, Addison Wesley

[2] Feathers, M. 2004. *Working Effectively with Legacy Code*, Prentice-Hall

[3] Freeman, S., Mackinnon, T., Pryce, N. & 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, 236-246. Available also from: http://www.planningcards.com/papers/

[4] Freeman, S., Pryce , N. 2010. *Growing Object-Oriented Software Guided by Tests*, Addison-Wesley

[5] Hunt, A. and Thomas, D. 1998. *Tell, Don't Ask*, Available at: http://www.pragmaticprogrammer.com/ppllc/papers/1998_05.htm

[6] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. M. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional

[7] Lieberherr, K. and Holland, I. Assuring Good Style for Object-Oriented Programs *IEEE Software*, September 1989, 38-48.

[8] Mackinnon, T., Freeman, S., Craig, P. Endo-testing: unit testing with mock objects. In *Extreme Programming Examined*, Addison-Wesley, Boston, MA. 2001. 287-301. Available also from: http://www.planningcards.com/papers/

[9] Martin, R. C. 2002. *Agile Software Development, Principles, Patterns, and Practices*, Prentice-Hall

[10] Arrow, H., McGrath, J. E. & Berdahl, J. L. 2000. *Small Groups as Complex Systems: Formation, Coordination, Development, and Adaptation*, Sage Publications

[11] Cognitive Edge: http://www.cognitive-edge.com/

[12] Mitra, S. Self organising systems for mass computer literacy: Findings from the 'Hole in the Wall' experiments, In *International Journal of Development* Issues, 4(1), pp 71-81 (2005).

[13] Beck, K., Fowler, M. 1999. *Refactoring: Improving the Design of Existing Code*, Chapter 3 Bad smells in code, Addison-Wesley,
See also http://en.wikipedia.org/wiki/Code_smell

[14] Feathers, M. 2004. *Working Effectively with Legacy Code*, pp 251 Heuristic #4: Look for internal relationship, Prentice Hall

[15] Goldstein, J. Emergence as a Construct: History and Issues, In *Emergence: Complexity and Organization* 1 (1): 49–72 (1999)

[16] Kauffman, S. A. 1993. *The origin of order: self-organization and selection in evolution*, Oxford University Press

[17] Kauffman, S. A. 1995. *At Home in the Universe: The Search for Laws of Self-Organization and Complexity*, Oxford University Press

[18] Goerner, S. 1994. *Chaos and the evolving ecological universe*, Langhorne PA: Gordon & Breach

[19] Pelrine, J. On Understanding Software Agility - A Social Complexity Point Of View, In *E:CO* Issue Vol. 13 Nos.1-2 2002 pp 26-37

[20] Forrest S. Balthrop J. Glickman M. Ackley D. Computation in the wild. E. Jen, editor, Robust Design: *A Repertoire of Biological, Ecological, and Engineering Case Studies*, pages 207–230. Oxford University Press, 2004. Reprinted in K. Park and W. Willinger Eds. *The Internet as a Large-Scale Complex System*, pp. 227-250. Oxford University Press (2005).

[21] Madeyski, L. 2010. Test-Driven Development: An Empirical Evaluation of Agile Practice, Springer.

[22] Madeyski, L. 2006. The Impact of Pair Programming and Test-Driven Development on Package Dependencies in Object-Oriented Design - An Experiment. Lecture Notes in Computer Science, 4034:278-289, Springer

[23] Taylor, K. , Rohrer, D. 2010. The effects of interleaved practice,