TDD with Mock Objects: Design Principles and Emerging Properties

Luca Minudel, 2009-2011
tdd@minudel.it

ABSTRACT

This is an exploratory study based on the experiences of a team that began writing code more compliant with S.O.L.I.D. design principles and partially with the Law of Demeter after adopting the practice of TDD with Mock Objects. The exploratory study analyze the relationship between the S.O.L.I.D. principles, the Law of Demeter and TDD with Mock Objects as well as investigate the intriguing hypothesis that TDD with Mock Objects encourages programmers to write code that adhere to these design principles as an emergent property.

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].

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, Emerging properties.

1. INTRODUCTION

The software development team I was part of, an F1 Racing Team, was working with a large and complex code-base. Initially part of the team was trained about OOP and OOD 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 change dramatically.

A year later another part of the same team was trained on the job using TDD with Mock Objects.

2. INITIAL OBSERVATIONS

As part of the training on the job some team members began to work in pairs with two other software developers very experienced in using TDD with Mock Objects. We coded in quick (5-15 minutes) red-green-refactor cycles writing tests that declared and verified expectations of how objects should interact with each other and we refactored the code to remove duplications and clarify code intent.

We also learned:

- how to use the refactoring tools 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 a third-party library,
- how to quickly navigate in the IDE between interfaces and the classes and tests,
- how to use a mocking tool to mock dependencies and declare and verify expectations,
- how to test non-trivial objects code in isolation,
- about the practice of avoiding getters and instead using *Smart Handlers* that are Visitor-like objects [6]. However this practice was not followed.

At that time I observed that the result was better code that was easier to understand, change and evolve. In comparison with the code written before, the code then produced was more adherent to the S.O.L.I.D. design principles, even code written by team members that didn't have training or previous knowledge about S.O.L.I.D. principles. The code produced was also partially more adherent to the Law of Demeter.

3. HYPOTHESIS

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

The idea that conformance to the S.O.L.I.D. principles and to the Law of Demeter can emerge from the practice of TDD with Mocks Objects is particularity interesting because:

- While the *values* of good software design and the *principles* of software design are abstract concepts, TDD with Mocks Objects is a *practice* that puts them to use in a very concrete way;

- Learning S.O.L.I.D. design principles and the Law of Demeter from the practice of TDD with Mocks Objects can be much simpler and effective than teaching them and requiring the team to conform to them;

- Mocking Tools could be explicitly designed to support and exploit all the potential of the emergent properties of TDD with Mocks Objects.

4. EVALUATION OF THE HYPOTHESIS

After the initial observations and after formulating the hypothesis we searched for similar studies about emerging properties and design principles and we found that the tendency of the code developed with TDD using Mock Objects to conform to the Law of Demeter as an emergent property has already been observed and reported before in [8]. By emerging property we understood this to mean that the tendency to the conformance is obtained without an explicit policy to do so, for example without training the team on the design principles or without requiring the team to produce code that conforms to them. This means that the tendency to the conformance is obtained simply as a consequence of applying the practice of TDD with Mocks Objects.

In order to evaluate the hypothesis, 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 and only partially adherent to the Law of Demeter and this is 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.

We discussed in retrospective of the experience together with all the software engineers involved 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 emerging properties called *co-evolution* [10], [11].

I also described the practices that we followed while doing TDD with Mock Objects and analyzed how each practice promote the conformance to the design principle. This is detailed in the following paragraph 5

Furthermore I conducted qualitative experiments using TDD micro-exercises by different developers using different programming languages. Developers were asked to refactor existing code that had different violations of the S.O.L.I.D. principles and the Law of Demeter, with the goal of make it testable and write the unit tests and enable the development of new change requests and new features with TDD. The resulting code has been evaluated comparing the code and the tests produced by developers using TDD with Mock Objects with code and the tests produced by developers not using TDD with Mock Objects.

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

TDD with Mock Objects prescribes practices useful for writing testable code and prescribes refactoring to fix specific code smells surfaced by the unit tests code [3] [4] [8]. The unit tests guides the design and serve as a mirror through which the defects in the design of the code are surfaced.

We analyzed how the practices we used to apply TDD with Mock Objects and how in our opinion they prevented and avoided violations of the S.O.L.I.D. principles and the Law of Demeter in the code we wrote.

5.1 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.

5.1.1 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. Look [8] at paragraph 4.3.

5.1.2 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*. Look [8] at paragraph 4.3.

5.1.3 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". [3] paragraph 1.2, and [4] chapter 2 paragraph "Tell, don't ask" and chapter 20 paragraph "What the Tests Will Tell Us (If We're Listening)", [5].

5.2 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.

5.2.1 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.

5.2.2 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. [3] paragraph 4.8 and [4] at chapter 20, paragraph "Bloated Constructor".

5.2.3 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".

5.2.4 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 the responsibility into a new class.

5.2.5 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*.

5.2.6 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 [3] at paragraph 2.1 and paragraph 3.3.1 and paragraph 3.7.

5.2.7 Where TDD with Mock Objects doesn't helps 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.

5.3 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.

5.3.1 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 [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. And 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".

5.3.2 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.

5.3.3 Practice

The frequent refactoring during the red-green-refactor cycles of TDD with Mock Objects helps to replace conditionals with inheritance 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.

5.3.4 Where TDD with Mock Objects doesn't helps 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.

5.4 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.

5.4.1 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.

5.4.2 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.

5.4.3 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 [3] at paragraph 2.1 and paragraph 3.3.1 and paragraph 3.7. This avoids many violations of the LSP too.

5.4.4 Where TDD with Mock Objects doesn't helps 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 notices the inconsistency and find how to fix the LSP violation.

6. THE EXPERIMENT

The experiment documented here is an exploratory study conducted between 2006 and 2008, with the goal to better comprehend the nature of the relation between the practice of TDD and adherence to the design principles. At that time no studies have been conducted in this area. This work is useful to understand the phenomenon and to enable the development of a model and set up a rigorous plan for a comprehensive investigation.

TDD is a practice commonly used in Agile software development that's an empirical process which take into account the people involved in the project and the social dynamics, the technologies

and the requirements changing and evolving during the project. This means that the people that implement the software and the organizational context where the software is developed together with the code-base and the project all are variables that influence the result of the experiment.

Because of this nature of Agile software development, in order to understand the phenomenon of the relation between the practice of TDD and adherence to the design principles, it is interesting to observe the practice of TDD in a real context with professional software developers working on a real project. This permits to investigate and understand the phenomenon for a specific team and later this permit to investigate the generalization of the findings to other teams and contexts.

Observational study here means that the experiment described is a real experience report, documented in retrospective, about a real team implementing software used during the Formula One Racing Championship from a leading F1 racing team. An observational study by definition is an uncontrolled experiment based on observations, an otherwise controlled experiment would have not been practical in this context. Also the collection of quantitative data would have not been practical in this context and also not possible because of confidentiality requirements.

6.1 Findings

6.1.1 Initial training on OO design principles

At the beginning of 2006 all the team members completed an intermediate training on OOP and OO design principles.

6.1.2 After the training on OO design principles

During 2006 no improvements of the quality of the code and of the design of the code-base has been reported by any software engineer. During 2006 software engineers continued to report difficulties to unit test the code and problems with slow and fragile automatic tests each one testing cluster of objects interacting also with external systems (i.e. the database, the mail server).

During 2006 have been reported also many efforts to use advanced features of commercial mocking tools with the goal of mocking static classes, classes referenced from external libraries and classes instantiated directly inside the class under tests.

6.1.3 The training on TDD with mock objects

During 2007 two groups of software developers attend an internal hands-on training on TDD with mock objects and some of them have the chance to continue training on the job.

6.1.4 Just after the training on TDD with mock objects

During 2007 software developers that attended the course and continued to apply the practice of TDD argue the effect on the code of the practice, if they are good or not. In example there are discussions about the parametric constructors that have been added and used only by the tests, about the numerous small classes with less responsibilities that are created while doing TDD, about the use of interfaces defined to enable the mocking of objects, about the use of default constructors of factories to instantiate objects and wire the system, about the increased use of containment over inheritance.

6.1.5 One year after the training on TDD with mock objects

During 2007 and 2008 TDD become an established practice for many software developers of the team. Difficulties with slow and fragile automatic tests and problems are reported on tests created before the training while no problems or difficulties are reported for the unit tests written with TDD.

Some software developer in the team starts to study SOLID design principles and the Law of Demeter while some other software developers discuss additional practices and smells for TDD and their relation with the resulting design. The discussions about the relation between the practice of TDD and the adherence to design principles start. Adherence to the design principles are observed in the code during daily programming activity.

During 2009 start the documentation of this experience that has been verified with some of the software developers involved.

6.2 Discussion

Reflections and opinions about the findings are described in the paragraphs 2.Initial Observations, 3.Hypothesis, 4.Evaluation of the Hypothesis.

Joseph Pelrine suggested that the two software developers very experienced in using TDD with Mock Objects that joined the team and delivered the on-the-job training acted as Attractors as defined in the ABIDE (Attractors, Barriers, Identity, Dissent/diversity and Environment) model developed by Dave Snowden at the Cynefin Center for Organisational Complexity and now at Cognitive Edge [13].

Michael Feathers at the Norwegian Developer Conference 2010 during the session ‘The Deep Synergy Between Testability and Good Design’ says that writing tests is another way to look the code and locally understand it and reuse it, and that it is the same goal of good OO design. This is the reason of the deep synergy between testability and good 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 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 Steve Freeman: *No technique can survive inadequately trained developers.*

A relevant quote from Stafford Beer: *If you wish to tell someone how to reach the top of a mountain that is shrouded in mist, the heuristic 'keep going up' will get him there.*

6.3 Threats to validity

Since this an observational study based on observation of an uncontrolled experiment 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.

An experiment with the questionnaire and the code should mitigate those threats to validity.

7. CONCLUSION

The red-green-refactor cycles of TDD with Mocks Objects led us to write code more conformant to the S.O.L.I.D. design principles and partially to the Law of Demeter. The observations and the analysis of the relation between the practice of TDD with Mocks Objects that we used and the improved conformance to the S.O.L.I.D. principles and the Demeter's Law in the code we wrote, is compatible with the hypothesis that the TDD with Mocks Objects tend to avoid and prevent violations of the S.O.L.I.D. principles and to detect and remove existing violations of the S.O.L.I.D. principles and of the Demeter's Law.

The hypothesis that code developed with TDD using Mocks Objects tends to conform to the S.O.L.I.D. principles and of the Demeter's Law as an emergent property, without an explicit policy to do so, is also compatible with the observations and the analysis.

In the qualitative experiments conducted using TDD micro-exercises the code produced by developers using TDD with Mock Objects had less violations of the design principles. While the resulting code produced by developers not using TDD with Mock Objects generally had more violations of the design principles. The developers not using TDD with Mock Objects that produced a code with few violations of the principles has been asked how the adherence to the design principle has resulted, the answer was: by intentionally refactoring the code to improve the design, not in the process of making the code testable.

In conclusion the results of the exploratory study are compatible with the hypothesis that code developed with TDD using Mock Objects tends to conform to the S.O.L.I.D. principles and to the Law of Demeter as an emergent property, and that the practice of TDD with Mocks Objects is an effective way to design code and apply design principles in large and complex code-base and the *secondary effect* called *co-evolution* results to be a valid tool to learn and fully understand and apply design principles.

8. FURTHER WORK

The hypothesis explored here has interesting applications in the code design also for large and complex code-base and in the teaching of the design principles. And suggests the possibility of the existence of others emerging properties in the coding practices.

This explorative study is based on observations on the experience of a team and on evaluations of the result of qualitative experiments with a small numbers of developers using TDD micro-exercises.

More observations and exploration of the hypothesis based on the experiences of other teams adopting the practices of TDD with Mock Objects are needed to test the validity of the hypothesis in different teams and contexts.

The experiments with TDD micro-exercises can be extended to a larger number of developers and the results can be evaluated in a more formal way, using questionnaires to collect information from the developers and using software metrics.

9. 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 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 anyone who helped to reflect on what we have learned and who helped to review this paper since the first incomplete draft and develop all the embryonic ideas to their full extent, and set the foundations for further advancing our understanding and practice of coding.

Thanks to the XPUG-IT members.

10. REFERENCES

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

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

[3] Freeman, S., Mackinnon, T., Pryce, N. & Walnes, J. 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. *Growing Object-Oriented Software Guided by Tests*, Addison-Wesley, 2010

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

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

[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. *Agile Software Development, Principles, Patterns, and Practices*, Prentice-Hall, 2002.

[10] Highsmith, J. & Cockburn, A. *Agile software development: the business of innovation*, Computer 34, 120-127 (2002). DOI= http://dx.doi.org/10.1109/2.947100.

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

[12] Poppendieck, M. & Poppendieck, T. *Lean Software Development: An Agile Toolkit*, Addison-Wesley, 2003

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