New Trends in Software Methodologies, Tools and Techniques
H. Fujita and R. Revetria (Eds.)
IOS Press, 2012

doi:10.3233/978-1-61499-125-0-57

Towards Model-Based Testing Patterns for Enhancing Agile Methodologies

Darioush JALALINASAB[1] and Raman RAMSIN
Department of Computer Engineering, Sharif University of Technology, Tehran, Iran

Abstract. Quality is one of the most important issues in the context of agile and lightweight methodologies. These methodologies recommend automated testing as the main method for quality assurance; however, they are plagued with several deficiencies in this regard, including complex and difficult-to-maintain test case scripts. Model-based testing is an approach for automating the test creation process through replacing individual test-case design with abstract models. In this paper, we explore a set of patterns based on current methods used in model-based testing which can be used to ameliorate the abovementioned deficiencies in agile/lightweight methodologies. We then demonstrate how these patterns can be applied to a concrete agile methodology – namely Feature Driven Development – to address problematic testing issues while maintaining the agility of the process.

Keywords. Software testing, model-based testing, agile methodologies, feature-driven development

Introduction

Agile methodologies have permanently changed the way we view software development. Since agile methodologies were conceived, academia and industry have focused a great deal of attention upon them; XP is a prominent example [1]. The promise of rapid application development while retaining high levels of quality has made these methodologies appealing to software engineers worldwide.

Iterative testing, especially test-driven development (TDD) [2], is the cornerstone to quality assurance in all agile/lightweight methods. Through bringing testing into the main development cycle and focusing on automated testing, these methodologies aim to achieve a low defect rate while remaining faithful to the principles of agility.

The spread of agile methodologies was accompanied by a decline in the value of modeling in the agile/lightweight context. However, recent works such as [3] focus on reintroducing a minimalistic approach towards modeling. Scalability problems along with research demonstrating dubious results about quality levels achieved with TDD [4] may be the root causes for the advent of these approaches.

Model-Based Testing (MBT) techniques predate agile methodologies; however, many of these methods are not directly suitable for application to these methodologies. MBT's need for precise and elaborate models, often deliberately avoided by agile/lightweight methodologies, is the main reason for this mismatch. The recent focus of MBT on the use of practical, object-oriented models (e.g., UML) on the one hand [5],

[1] Corresponding Author: Darioush Jalalinasab, Department of Computer Engineering, Sharif University of Technology, Azadi Ave., Tehran, Iran; E-mail: jalalinasab@ce.sharif.edu

and the utilization of models in agile processes on the other, has re-instantiated the opportunity of benefiting from MBT in agile/lightweight approaches.

This opportunity has already been noticed by research efforts, including [6], [7], and [8]. Unfortunately, most of this research does not provide general, direct solutions to problems in agile testing. It must also be considered that due to the fact that each project has its own special conditions and requirements, it would be naive to provide a single-solution MBT approach and claim that it would be successful in all agile/lightweight contexts.

We have identified three major areas of agile/lightweight testing in which MBT may lead to beneficial solutions. The first area is the abstraction level of test cases. Achieving a higher level of abstraction is often cited as one of the major benefits of MBT [9]. Since test cases are usually developed in an ad-hoc and individual manner in agile/lightweight methodologies, the benefits of MBT can be significant in this context. The second area investigates how MBT can benefit agile/lightweight testing in resolving design and requirements ambiguities. The third area deals with how MBT can provide overall direction to the testing process and provide meaningful test coverage goals.

Since the now-popular design patterns [10] were introduced, patterns have become a favorite method of conveying families of solutions to specific problems. Therefore, we express possible solutions applicable to the mentioned areas as patterns. For each pattern, we will discuss the situation in which it should be applied and provide examples of relevant literature.

The rest of this paper is structured as follows: In section 1, we review MBT and other related literature with a focus on methods that may be applicable to testing in agile/lightweight methodologies; in section 2, we discuss some of the major testing problems these methodologies are facing, and we propose a set of patterns (as solutions) based on the literature reviewed in section 1; section 3 demonstrates how these patterns may be used to augment agile methods by applying them to a concrete agile methodology, FDD; and the final section is dedicated to providing the conclusions and proposing possible directions for future research in this context.

1. Related Work

The main benefits of applying MBT to agile development have been explained in [6]; improving issues related to meaningful coverage, low flexibility, high maintenance costs, and how MBT may help in acceptance testing are among the major benefits mentioned. However, the MBT approach prescribed by [6] is a specific approach to MBT with limited applicability, whereas we take a more general viewpoint and discuss how MBT may be applied to agile/lightweight methodologies using testing techniques based on UML diagrams, metamodels, requirements, and architectural models. The following subsections present samples of research conducted in each of these areas.

1.1. MBT based on UML diagrams

The research reported in [11] proposes a method for using UML models for different testing purposes. Models are used as test data, test drivers, and test oracles. Object diagrams are proposed to be used as test data (and oracles) by demonstrating the initial (and final) conditions; objects involved, their interconnections, and the values of

concrete attributes relevant to testing are specified in these diagrams. A prominent consequence of utilizing object diagrams as test data is the opportunity to reuse a single diagram in multiple test cases after applying small modifications (such as changing attribute values), as mentioned in [11]; in effect, this is similar to the "shared fixture" pattern discussed in [12].

Furthermore, [11] suggests the use of sequence diagrams as test drivers when the interaction scenario is not simple. This allows a comprehensible description of assertions on inter-object behavior, due to the fact that object interactions are well displayed. OCL constraints can also be used in this context in the same manner as assertions are used in test scripts.

The spread of UML-based testing methods has led to the creation of the UML Testing Profile (UTP) as an OMG standard [13]. UTP proposes extensions to sequence diagrams, such as test verdicts and timing issues, which allow these diagrams to be used as bases for test case generation. In addition, the UTP pays attention to the "test context," which allows for the complete visualization of the SUT, test data (including rudimentary support for data partitioning techniques), and test doubles. Finally, although the UTP is not executable as is, guidelines are provided for automated translation to well-known testing frameworks, including jUnit and TTCN-3.

In [14], an interesting MBT technique is proposed. The main idea is to raise the abstraction level of the "design by contract" [15] concept to the level of design models. Each method's functionality is to be specified by a "visual contract"–which consists of two UML composite structure diagrams [14]. One diagram designates the pre-conditions, while the other documents the post-conditions. The proposed notation also allows for negative assertions. Visual contracts are then automatically translated to JML [16] assertions. JML is an extension to Java which allows such assertions to be monitored, and an exception will be raised upon the violation of a contract [14].

Another important issue that [14] touches upon is from the process point of view. It is explicitly suggested that rather than attempting to perform code generation from behavioral models automatically, as is usual in model-driven engineering approaches, only structural models be defined; it leaves method bodies to be developed by coders. As noted in [14], this allows the concept of TDD to be raised to a higher level of abstraction, since visual contracts can serve as unit tests for methods. This point of view may allow claims such as relying on TDD as a form of design (as envisioned in [17]) to be made with a higher degree of confidence.

The technique discussed in [18] allows for QA on static aspects of class diagrams; therefore, it might not be classified as MBT, or even a testing technique. However, since class diagrams are among one of the few models allowed in agile/lightweight methodologies, we have chosen to discuss this technique. This approach proposes an extension to object diagrams, namely the "modal object diagram". This diagram allows defining positive/negative object configurations that the class diagram should allow/disallow. In addition, it is possible to limit the values of attributes to specific sets. It is therefore possible to consider each "modal object diagram" as a test case that the class diagram should satisfy. After these test cases become available, a fully automated technique based on model-checking (detailed in [18]) performs the verification.

The potential benefit of the technique proposed in [18] to agile/lightweight methodologies is that it allows verification to be performed on models created in a minimal fashion. Even experienced modelers may make mistakes in subtle issues such as association multiplicities, and preventing them through this method may prove valuable further on in the process, such as when code is generated from the class

diagram. In addition, these "test cases" may serve as a form of regression testing and allow the model to evolve alongside the code.

The issue of testing UML models as independent from implementation is discussed in [19]. In this research, several testing adequacy criteria have been proposed for various UML diagrams. The criteria related to class diagrams involve creating instances with all possible multiplicities (unbounded associations must have an arbitrary limit assigned), creating all possible subclasses, and creating objects with different field combinations. Different field combinations may be created based on the knowledge contained in OCL constraints, or by considering the data type of the field [19].

However, the most interesting set of coverage is defined for UML collaboration diagrams. This set includes predicate and term coverage of UML guards assigned to message edges and the execution of all messages and paths on these diagrams [19]. This sort of coverage can automatically be applied to the models created, and may furthermore result in test cases that can be carried on to code (the latter aspect, however, is not discussed in [19]).

1.2. MBT based on metamodels

In the research reported in [20], a program is discussed that processes the models themselves. The testing of such applications is explored, and a technique based on the use of metamodels is proposed. It is mentioned that well-defined metamodels allow the automatic (or at least, semi-automatic) creation of simple test cases. Some of the problems that arise in this context are related to the constraints that have not been mentioned in the metamodel. The approach presented in [21] proposes a method which allows further automation of this technique, by composing test cases from valid, manually created "model fragments".

While [20] and [21] discuss the generation of valid test cases from metamodels, [22] intends to define viable mutation operators which function based on the knowledge contained in the metamodel. It is demonstrated in [22] how these operators can be defined for a specific domain; in this case, model-processing utilities. The implication of this research for agile/lightweight methodologies is that mutation testing is considered on a more abstract level than procedural or object-oriented code. This technique can be applied both to domain-specific metamodels and technical models created during development.

1.3. MBT based on requirements

Requirements-based MBT forms a large body of the literature in the field. The research reported in [23] presents one of these approaches. This approach is based on use cases, and their related activity and collaboration diagrams. After activity diagrams are obtained for each use case, the data that it operates upon are parameterized; this means that the parameters required for the use case to operate are specified, much like the formal parameters of a procedure. Next, the valid paths between use cases are documented using a language similar to regular expressions. The higher-level abstractions are then mapped to sequence diagrams. These are in turn once more mapped to regular expressions, which form the basis of test cases showing the order in which individual methods should be invoked.

Use case maps are another model that can guide testing, as suggested in [24]. This model supports documenting the relationships among use cases, the potential for concurrency, and the conditionals. Since this model resides at a very high level of abstraction, it cannot be used by itself for generating test cases [24]. Techniques such as the one proposed in [23] may be used to provide the necessary details.

The first step in directing the test effort using use case maps is flattening the map using patterns, as discussed in [24]. Next, a coverage criterion must be selected; the approach presented in [24] suggests some of these criteria. Also, by knowing the valid paths, invalid paths may also be generated and tested for enhancing system resilience.

In the research reported in [25], a method is discussed for automatic generation of test cases from activity diagrams. Activity diagrams are first enriched with data descriptions. These data descriptions are similar to the parameters mentioned in [23]. Next, the dependencies among activities are extracted in a table. This table, which can be automatically generated, shows data and control flow dependencies among the activities. Finally, a graph is created based on the dependency table, and all paths in this graph are enumerated to be used as a basis for testing. The authors of [25] compare their work with another method which involves both activity and sequence diagrams, and conclude that their method has been successful in decreasing the complexity of MBT. Unfortunately, the example is a small one, and the result may not be scalable to more general cases.

The research reported in [7] applies MBT to the GUI of the Nokia-S60 Smartphone. Through the use of state machines, which are labeled as "abstraction machines" and "activity machines," use cases are mapped to low-level GUI actions, such as selecting menus and pressing buttons. What this approach denotes as use cases are in fact scenarios that involve the invocation of multiple use cases. These high-level use cases are then re-written according to the abstractions provided by the state machines, and can be used as bases for test creation.

1.4. MBT based on architecture

The research presented in [26] introduces an approach which defines test adequacy measures based on the structural model of the software architecture. This approach views the architecture as components and the connectors between them. A number of interfaces are defined for each component and connector, and then the data flow dependencies between the interfaces are modeled. The adequacy measures involve testing each component and connector in isolation, testing the direct connection of component-connector-component paths, indirect paths, and all paths coverage. The research presented in [27] proposes another similar approach that emphasizes the structural role of architecture during testing.

1.5. Other relevant work

While not directly related to testing, the research reported in [28] proposes a language that may be helpful in applying MBT to agile/lightweight methodologies. The proposed role-based modeling language (RBML) [28] is a pattern description language based on UML which allows the user to impose structural and behavioral constraints on objects. It is possible to describe the patterns of a specific domain in RBML. The application of languages with formal semantics to patterns allows enforcing their correctness.

Currently, mainstream languages do not provide the necessary features for verifying the correctness of the implementation of even the simplest of patterns.

2. Challenges and Patterns for Applying MBT in Agile Methodologies

In this section, we discuss a number of challenges that testing is currently facing in agile/lightweight communities. We will then attempt to address these issues through the application of MBT. For each problem, a set of patterns representing a class of MBT methods will be introduced, along with representative samples from related literature. The patterns discussed in this section have been summarized in Table 1.

2.1. Problem Issues pertaining to the level of abstraction

Tests are often developed as individual units, and therefore they tend to reside at a low level of abstraction. One of the main promises of MBT is to raise the level of abstraction by focusing on models rather than individual test cases; in turn, this will result in less time spent on testing activities.

In [29], increased development time has been cited as one of the impediments to the applicability of TDD. Increased abstraction will result in less work. MBT methods may also include automated methods for tasks previously accomplished in a manual or ad-hoc manner, which will also decrease total implementation time.

Maintenance costs for large test suites are among the major costs of software engineering projects. If MBT is practiced, such costs are also expected to decrease, since applying changes to the models and invoking the (mostly) automated techniques of MBT is easier than applying changes individually to each test case. Some of these benefits have been highlighted in [6].

Another problem with traditional test cases is that they are tightly coupled with the module(s) they are testing, and this provides little opportunity for reuse. If, through MBT, testing is brought to higher levels of abstraction, more promising opportunities arise for reuse. If the level of abstraction is sufficient enough, cross-project reuse may become possible, as demonstrated in [30].

In [31], an online survey has been performed which indicates that several of the most common mistakes when practicing Test-Driven Development are related to test-case complexity. In fact, "the need for writing complex test scenarios" has been cited as the most common mistake. It may initially seem paradoxical to attempt to solve an issue of complexity by requiring additional software models; however, as we show in the patterns, strategic use of modeling can simplify some of the more complex test scenarios.

2.1.1. Pattern: Use of models in place of test data

The initial data that tests operate on, or the results which they must be compared to, may be complicated or grow to become so. Often in such cases, the test script setup or verification logic will grow in complexity to match the complex structure it is creating or verifying. This will usually be the case when the relationships between objects are intricate.

Some solutions to this situation are known, such as creating a shared fixture [12] or storing data in a relational data store. Another alternative is using models to specify the

structure of the initial or expected data. Since the software under test is often written in the object-oriented paradigm, UML models are best suited to this task. Since the model provides an abstract, graphical view of the object connections, it is easier to understand and maintain. Examples of such work can be found in [11] and [14].

2.1.2. Pattern: Use of models in place of behavioral assertions

Mock objects are usually used to monitor the correct invocation of methods on objects. Although the experience of test developers and the advancement of mock object libraries (such as jMock [32]) have improved the readability of test cases involving mock objects, one must admit that the resulting code may be complicated or not easily legible, and may well become troublesome during maintenance.

Sequence diagrams are well suited to the task of displaying method calls, and can therefore be used to model the expected behavior of objects. If such diagrams are augmented with methods of modeling expectations and assertions (such as with OCL), it is possible to use them (and other behavioral models) in place of writing complex scripts for mock objects. The approaches proposed in [11] and [13] involve methods based on sequence diagrams which can be used to this end.

2.1.3. Pattern: Monitoring objects at runtime

Object invariants have been known since the advent of the "design by contract" approach. If taken further, it is possible to use invariants for defining configurations of objects that should always exist, as well as for expressing configurations that should never be allowed to exist.

Test cases which attempt to verify such issues will become overly complex if modeling is not used. Indeed, it may not be possible to verify that a certain configuration of objects did not occur during the execution of the system under test. In some cases, mock objects may allow the verification of certain methods not being invoked; however, even that solution is typically complex.

If the runtime environment is instrumented with facilities to detect certain patterns of objects, test scripts which tend to assert object configuration invariants and negative invariants will become much simpler or obsolete.

The research reported in [14] focuses on pre- and post-conditions; the concept of invariants may also be considered. The research reported in [18] proposes static verification; however, if this is not possible (perhaps due to the low rigor of the class diagram) applying the ideas during runtime is also viable. Languages with well-defined semantics, such as RBML [28] can also be utilized for defining the invariants and assertions.

2.2. Problem Issues pertaining to design and requirements ambiguity

Agile/lightweight methodologies have sometimes been criticized for their lack of design and lack of rigor in documenting requirements. This issue is in part due to their heavy reliance on testing as their prime method of quality assurance. Pioneers of TDD even label this method of testing as design [17]; however, this method is not completely flawless. As an example, research reported in [29] shows TDD has caused architectural problems in three different case studies (two academic, and one industrial).

In the aforementioned methodologies, requirements are usually captured as "user stories" [1] or informal use-cases. Certain acceptance-test-driven development (ATDD)

techniques such as FIT [33] have emerged which allow the end-user to specify their requirements in a more direct fashion. However, since each model, regardless of how informal or lightweight it may be, is accompanied by certain well-defined syntax and semantics, it will help specify the requirements in a more reliable manner.

Here, we enumerate methods which may help improve software quality through design and clarification of the requirements.

2.2.1. Pattern: Using architecture to drive tests

Many agile and lightweight methodologies pay attention to architecture. Even methods such as XP, which refrain the most from modeling, encourage the use of a "system metaphor" as a representative of architecture [1]. This implies that in most projects, a model of the architecture – albeit very informal – will be available.

However, agile/lightweight methodologies fail to enforce or verify the correct implementation of the architecture; due to the fact that most of the QA effort is focused on unit testing and validation through end-user demos and prototypes [34], deviation from architecture can be expected, as mentioned in [29].

To further ease the burden of such tasks, a library of pattern-like tests can be created for different architectures. Such a library would contain abstract test cases based on the connectivity of architectural components. Since architectural specifications in agile/lightweight methodologies are usually not detailed, it is possible to view and test them as such. The approaches proposed in [26] and [27] provide examples as to how architecture can be used for devising tests.

2.2.2. Pattern: Using domain patterns

Specific domains may be associated with certain structural and behavioral patterns. For example, accounting software systems will have patterns involving increasing, decreasing, and viewing the history of accounts. In cases where such patterns have been documented (e.g., as analysis patterns [35]), it is possible to devise a parallel set of models that will serve as patterns to create individual test cases.

In addition to domain patterns, several of the design patterns [10] may be accompanied by tests which verify their implementation according to their specified intent. As an example, the Observer pattern may be accompanied by tests which verify subscribing, unsubscribing, and the intended broadcast behavior.

The RBML language introduced in [28], along with the examples provided, can serve as a sample method on how patterns can be formalized. However, the applicability of such patterns to testing is yet to be explored.

2.2.3. Pattern: Using static analysis

Many agile/lightweight methodologies create some kind of object or class diagram; in such cases, static analysis can be used as a tool to verify model correctness. Through static analysis, it may be asserted whether certain states in the system may be reached (or not) at runtime. This technique is specifically applicable to the verification of the subtle issues previously mentioned. When the details of the design artifacts are improved in this manner, they will provide a better basis for code generation and testing.

Research examples in this area include [18] and [19]. Additionally, it is mentioned in [14] that if visual contracts are consistently used throughout the project, static

analysis methods can be applied by using the pre- and post-conditions of each method and the initial system state.

2.3. Problem Issues pertaining to obtaining meaningful coverage

If devising tests is left entirely to the developer, test cases will usually be developed in an ad-hoc manner, and the sole measure of test suite quality will become code coverage. However, studies have shown that this measure can be deceptive [6].

MBT is a plausible solution to this issue, since test coverage can be defined at a higher level and in a more meaningful manner. In addition, these high level goals can provide direction to the testing effort, both at the system level and the unit level. In fact, as described in [6], MBT can play an important part in the definition of work products in agile methods where this concept is important.

2.3.1. Pattern: Using architecture to drive tests

In addition to the design-enforcing aspect of this pattern, which was previously discussed, architectural coverage metrics can be used in place of, or alongside, line code coverage metrics. These metrics can also be used to provide direction to testing, especially on the system and integration levels; this can be beneficial to agile/lightweight methodologies, since they usually emphasize unit testing only. The research reported in [26] provides several coverage metrics that can be used in this context.

2.3.2. Pattern: Directing testing by modeling the permissible order of operations

In most agile methods, requirements are captured in some informal format. As mentioned before, "user stories" and use cases are commonly used. However, if the order in which these "user stories" or use cases can be run is also modeled, it can be used as a strong basis for automated MBT at the system level.

The significance of running automated tests at the system level becomes apparent when we consider that most agile/lightweight methodologies have ignored testing at this level. Although techniques for automated acceptance testing have emerged, according to a recent Internet survey [34], in practice they are used significantly less than unit testing methods.

The coverage provided by this sort of testing can be more reliable than line coverage, since it demonstrates that the system is capable of performing the normal paths of execution. Much of the MBT literature is focused on deriving tests from use cases or other requirements specifications; examples include [7], [23], [24], and [25].

2.3.3. Pattern: Use of models to generate invalid states

Since the models used in software creation (especially UML models) have well-defined metamodels, it is possible to use valid models of system behavior to create invalid and irregular behavior through mutation.

By forcing the system through invalid states, it can be ensured that critical errors will not occur, or that the system handles these states gracefully. Since such test cases can be generated automatically, this approach can be classified as online MBT [9].

It must be noted that this form of testing, although similar to "fuzz testing", can achieve more fruitful results. Common fuzz testing techniques involve mutating valid

input to find defects; however, the knowledge contained in models allows mutation at a higher level of abstraction. Therefore, complicated defects (e.g., those involving several intermediate states) are discovered more efficiently.

Even though the main application of this pattern is in system testing, if the valid order of operations is specified for a class (e.g., as a regular expression), this method may be used in unit testing as well. As discussed in the previous pattern, once the valid ordering of system operations is specified, invalid sequences may be used to automatically test the system under test for robustness through high level interfaces, such as the GUI. Since this form of testing is fully automated and may reveal serious security and reliability bugs, it can be an economic addition to agile/lightweight methodologies. Additionally, since automated acceptance tests are usually fragile, this form of testing may be used to obtain a certain degree of system reliability when proper automated acceptance testing is not feasible.

In addition to the three categories described, the research reported in [36] mentions that test suites describe a model of the system, even if not comprehensible to developers and users. This is indeed true, since tests do describe the expected system behavior. MBT helps make this model explicit and useful for developers and users.

3. Applying MBT to FDD

In this section, we demonstrate how the suggested patterns can be applied to a concrete agile methodology. First, a description of this methodology will be provided, and then we will explain how the patterns should be applied. Feature-driven development (FDD) [37] was chosen for two reasons: FDD has a well-defined process, and it provides ample support for modeling activities.

This does not imply that the proposed patterns are not applicable to other methodologies, such as XP; however, the use of MBT in such methodologies must be sporadic and opportunistic, and it must not conflict with the level of agility desired by the team. This issue is not only limited to MBT; it must be taken into consideration whenever modeling practices are applied in agile methodologies.

3.1. A description of FDD

The FDD methodology consists of five main phases: The first three are performed in a sequential manner, and the remaining two are iterated as required. In addition to these steps, FDD advocates a three-layer system architecture. The architecture contains components for the UI, business logic, data management, and also a component for interfacing with external systems. More details on FDD may be found in [37] and [38].

1. Develop an overall model: In this phase, modeling is performed to capture the major features of the domain. While the main focus is on creating a class diagram, important behavioral patterns of the domain may also be documented with sequence diagrams [38].
2. Build a features list: In FDD, requirements are captured through the concept of a feature. Each feature is an informal statement of a requirement which is stated in the format of "action result object". An example could be "calculate the sum of an invoice". Next, features are composed into "activities," and then categorized into "areas" for easier management.

Table 1. Summary of the proposed patterns for applying MBT in agile/lightweight methodologies

Pattern	Context	Situation	References
Use of models in place of test data	Unit Tests	Test script has become complex due to complexity in setup or verification logic. Usually will occur when many objects are involved or inter-object relationships are intricate.	[11], [14]
Use of models in place of behavioral assertion	Unit Tests	Test script has become complex due to complex behavioral assertions. An important symptom is the presence of many mock objects or other test doubles.	[11], [13]
Monitor objects during runtime	Unit Tests	Test script has become complex due to multiple inspections of object or inter-object state, trying to enforce pre- and post-conditions or object invariants. Negative behavioral assertions, such as checking that a method is never called, are another situation where this pattern should be applied.	[14], [18]
Using architecture to drive tests	System Test	Project has deviated from architecture, or developers have become ignorant of architecture. TDD is not achieving expected results through emerging and evolving design. Unit test code coverage is high, yet various components do not interact well. Tests are developed in an ad-hoc manner and no measure of progress is available.	[26], [27]
Using domain patterns	Unit Tests System Test	An abstract description of the expected system behavior is available in part in the form of domain or technical patterns. Enforcing these patterns ensures a higher level of quality than unit test code coverage.	[28], [34]
Using static analysis	Class Diagrams	A static model (class/object diagram) of the system is to be created and maintained. The quality of the model is important for future use (e.g., code generation), and subtle mistakes should be prevented. The model is subject to evolution and a form of regression testing is required.	[18], [19]
Directing testing by modeling the order of operations	System Test (Unit Tests)	A model demonstrating the valid order of operations on a system (or object) is available (e.g., in the form of regular expressions). It is important to exercise valid execution paths through the system (or object). Unit test code coverage is high, yet various components do not interact well. Tests are developed in an ad-hoc manner and no measure of progress is available. It is desired to have a form of automated acceptance testing. Only high-level interfaces such as the GUI are available for testing.	[7], [23]–[25]
Use of models to generate invalid states	System Test (Unit Tests)	Models have well-defined metamodels, and valid examples are available. Mutation operators are definable for the relevant domain. It is important to test the resilience of the system (or object) against invalid states. Proper acceptance testing is too expensive, but some degree of system reliability is desired.	[20]–[22]

3. Plan by feature: During this phase, a development plan is devised based on the features defined. This plan will describe the dependencies among features, the development schedule, and the assignment of feature groups to programmers.
4. Design by feature: Developers assigned to feature groups meet and determine how objects will realize the needed behavior. This task is accomplished by drawing sequence diagrams and modifying the class diagram as needed.
5. Build by feature: Classes and methods needed for the realization of the features are coded and unit tested.

3.2. Applying the proposed MBT patterns to FDD

During the *"Develop overall model"* phase, *using domain patterns* can be a helpful pattern to get started with. Applying this pattern allows domain experts and modelers to specify test behavior from the beginning. These tests may be later used as acceptance tests, since they verify the compliance of the system under test to the expected behavior.

Use architecture to drive tests can also be applied during this phase. By considering the default three-tier architecture and also the logical modules and components specified in the overall model, tests can be defined that will enforce the architecture throughout the development process.

Since a relatively complete class diagram is devised, and furthermore, is used as a guide during the *"Design by feature"* phase, it is a worthwhile activity to ensure the correctness of this artifact. Applying *using static analysis* will allow modelers to be able to test their overall model.

The next modification to FDD must occur in the *"Build a features list"* phase. The allowed order of feature execution should be defined. This ordering will allow the application of *directing testing by modeling the permissible order of operations* at the system level. These tests may be used as automated acceptance tests later on.

The *"Design by feature"* phase must also be reconsidered to accommodate for changes. *Use models in place of behavioral assertion* allows the sequence diagrams to be augmented with relevant test data. Such diagrams may later become the basis of component-level testing. *Use models in place of test data* may also be applied to describe the expectations from class operations at a more abstract level. *Monitoring objects at runtime* may also be taken advantage of in order to define invariants at the object or component level. Finally, *using static analysis* will enable designers to verify the object model. Furthermore, the static analysis test cases produced in *"Develop an overall model"* may be used as a form of regression testing at this point.

During the *"Build by feature"* phase, unit testing takes place. In case the symptoms described in *use models in place of test data* or *use models in place of behavioral assertion* arise, the respective pattern should be applied.

The final modification involves adding a new phase to FDD. This phase will be concerned with acceptance testing. Since the order of allowed operations have been determined in *"Build a features list,"* the patterns *direct testing by modeling the order of operations* and *use of models to generate invalid states* can be applied as further measures of quality enhancement.

3.3. Evaluating the resulting methodology

The change in the degree of agility is the most important factor which must be evaluated after the application of the mentioned patterns to FDD. In [40], a method is

proposed for quantitatively evaluating the degree of agility in different methodologies. The evaluation process is based on assigning binary values to flexibility (FY), speed (SD), leanness (LS), learning (LG), and responsiveness (RS) for each phase and practice prescribed by the methodology. The values are then averaged across a table, which results in a numerical metric which can be used for comparison. FDD has been evaluated in [40], according to these metrics.

In order to evaluate the extension, the speed of the phases *"Develop an overall model"* and *"Design by feature"* were set to 0 instead of 1. Other values are the same as those used in [40]. In addition, three practices were added to the evaluation matrix. "Domain level testing" refers to the patterns *using domain patterns*, *using architecture to drive testing*, and *using static analysis*. This is the testing counterpart of "Domain object modeling"; therefore, it has been evaluated identically, except for speed. "Feature level tests" refers to the application of *use models in place of behavioral assertion*, *use models as test data*, and *monitoring objects during runtime* at the feature level. This is the testing counterpart of "Developing by feature"; therefore, it has also been evaluated identically, except for speed. "Modeling order of operations" was not considered to have the necessary features to be evaluated as an agile practice, and therefore has been assigned 0 in all fields. Table 2 depicts the details of this evaluation.

It can be observed that the "degree of agility" (as introduced in [40]) drops from 48% to 40% (about 17% decrease) with respect to phases, and it drops from 70% to 62% (about 11% decrease) with respect to practices. Therefore, although there will be a noticeable decrease in agility, the decrease is not drastic.

However, the method discussed in [40], does not consider the relative length of phases with respect to one another, nor does it consider the effect of phases which are repeated iteratively versus the initial, sequential phases of agile methodologies. As discussed in [37], the three initial, sequential phases of FDD are estimated to require 23% of project resources, while 77% of resources are to be assigned to the iterative *"Design by feature"* and *"Build by feature"* phases.

The key to preserving agility in the modified process is considering that the application of the *use models in place of test data* and *use models in place of behavioral assertion* patterns to the *"Design by feature"* and *"Build by feature"* phases is not mandatory. Since these phases are performed iteratively, it can be expected that through post-iteration retrospectives, team members will become efficient in detecting situations in which applying these patterns would be beneficial.

In addition, *"Design by feature"* benefits from the static analysis test cases created during the *"Develop an overall model"* phase. Although maintaining these test cases may seem a hindrance at first, this cost is similar to the maintenance of test cases during TDD, which is usually not considered a hindrance to agile methodologies.

Specification of invariants through the application of the *monitor objects during runtime* will also provide benefits through reducing the number of test cases that need to be coded and maintained.

The three mentioned reasons lead us to expect that after a few iterations, performance in the iterative phases of the modified process should not be any worse than the original FDD process.

The initial, sequential phases, however, will incur a decrease in performance. The additional time required is due to specifying test cases for architecture, domain patterns, and for the models in the initial phase. The other loss of performance is due to specifying the ordering between operations and the data dependencies between them.

Even if the time required for the sequential phases was doubled, the overall time required would only increase by 23%.

It must also be noted that agility is not the only metric that should be used in evaluating agile methods. In [41] a comprehensive framework for evaluating agile processes has been set forth. Although a comprehensive evaluation is out of the scope of this paper, the FDD extension discussed here enhances many of the metrics mentioned in [41]; examples include generic development lifecycle coverage, adequate products, modeling coverage, testability, requirements elicitation, and completeness.

Table 2. Quantitative evaluation of the proposed FDD extension

	Agility Features					
	FY	SD	LS	LG	RS	Total
Phases						
Develop an overall model	1	**0**	0	1	1	3
Build a feature list	0	0	0	0	0	0
Plan by feature	0	0	0	0	0	0
Design by feature	1	**0**	0	1	1	3
Build by feature	1	1	0	1	1	4
Total	3	1	0	3	3	10
Degree of agility	3/5	1/5	0/5	3/5	3/5	10/25
Practices						
Domain object modeling	1	1	0	1	1	4
Developing by feature	0	0	0	0	0	0
Individual class ownership	1	1	0	1	1	4
Feature teams	1	1	0	1	1	4
Inspection	1	1	0	1	1	4
Regular builds	1	1	0	1	1	4
Configuration management	1	1	0	1	1	4
Reporting/visibility of results	1	1	0	1	1	4
Domain level testing	*1*	*0*	*0*	*1*	*1*	*3*
Modeling order of operations	*0*	*0*	*0*	*0*	*0*	*0*
Feature level tests	*1*	*0*	*0*	*1*	*1*	*3*
Total	9	7	0	9	9	34
Degree of agility	9/11	7/11	0/11	9/11	9/11	34/55

4. Conclusions and Future Work

In this paper, we first provided a summary of MBT literature representative of techniques that we found feasible to apply to agile/lightweight methodologies. Methods were selected which were based on tangible models that can be created and maintained in such a methodology. Next, an analysis of several problems that testing currently faces were presented. The problems discussed included:

- Low level of abstraction of test cases leading to costly maintenance, low reuse, and high fragility;
- TDD's tendency to generate suboptimal designs;
- The inability of agile/lightweight methodologies to prevent deviation from architecture;
- Ad-hoc tests being developed without a meaningful progress indicator;
- The problem of relying on code coverage as the sole test-suite quality metric.

For each category of problems, a set of patterns were proposed based on the body of knowledge reviewed. These patterns each represent a category of MBT methods, and can serve as guidelines for choosing or devising new task-specific methods. These patterns have been summarized in Table 1.

Finally, it was demonstrated how to apply these patterns to FDD, one of the best known agile methodologies, as an example of how these patterns can be fused into a methodology. The modified methodology was then evaluated with respect to the metrics introduced in [40] and [41]. The evaluation leads us to the conclusion that the prejudice of agile methodologies towards modeling notwithstanding, MBT may actually be able to contribute to software quality and its important metrics while having only a moderate impact on their agility, if applied properly.

Further research should be carried out on the use of MBT in agile/lightweight methodologies. Some of the impediments which limit the use of MBT in these fields are listed below, which should be addressed in future research:

- Configuration management and version control systems are essential to agile development. Unfortunately, support for managing model artifacts is rather limited in the agile development tools currently employed.
- Integration of models with popular IDEs, and tool support for the various methods proposed are essential to the adaptation of MBT to agile contexts.
- A programming language interface should be designed (such as EMF [39]) which allows access to models from scripted tests, and vice-versa.
- Although empirical evidence on the efficacy of MBT exists, industrial research has yet to prove its symbiosis with agile/lightweight methodologies.
- MBT techniques are not usually designed with agile/lightweight methodologies in mind; exploring the ability to design specific lightweight MBT techniques may thus prove to be an interesting field of research.

References

[1] K. Beck and C. Andres, Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional, 2004.
[2] K. Beck, Test Driven Development: By Example. Addison-Wesley Professional, 2002.
[3] S. W. Ambler, Agile modeling: Effective Practices for Extreme Programming and the Unified Process. Wiley, 2001.
[4] S. Kollanus, "Test-driven development - still a promising approach?" International Conference on the Quality of Information and Communications Technology, QUATIC, pp. 403–408, 2010.
[5] A. C. D. Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: a systematic review," in ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies, WEASELTech. ACM, 2007, pp. 31–36.
[6] D. Faragó, "Model-based testing in agile software development," in 30. Treffen der GI-Fachgruppe Test, Analyse & Verifikation von Software (TAV), Testing meets Agility, ser.Softwaretechnik-Trends, 2010.
[7] M. Katara and A. Kervinen, "Making model-based testing more agile: A use case driven approach." in Haifa Verification Conference, ser. Lecture Notes in Computer Science, vol. 4383. Springer, 2006, pp. 219–234.
[8] B. Rumpe, "Agile test-based modeling." in Software Engineering Research and Practice, CSREA Press, 2006, pp. 10–15.
[9] M. Utting and B. Legeard, Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, 2007.
[10] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
[11] B. Rumpe, "Model-based testing of object-oriented systems." in Formal Methods for Components and Objects, International Symposium, FMCO, ser. Lecture Notes in Computer Science, vol. 2852. Springer, 2002, pp. 380–402.

[12] G. Meszaros, XUnit Test Patterns: Refactoring Test Code. Addison-Wesley, 2007.
[13] P. Baker, Z. R. Dai, O. Haugen, and I. Schieferdecker, Model-Driven Testing: Using the UML Testing Profile. Springer, 2008.
[14] G. Engels, B. Gldali, and M. Lohmann, "Towards model-driven unit testing." in MoDELS Workshops, ser. Lecture Notes in Computer Science, vol. 4364. Springer, 2006, pp. 182–192.
[15] B. Meyer, "Applying design by contract," IEEE Computer, vol. 25, pp. 40–51, 1992.
[16] The Java modeling language (JML). [Online]. Available: http://www.cs.ucf.edu/~leavens/JML/
[17] K. Beck, "Aim, fire," Software, IEEE, vol. 18, no. 5, pp. 87–89, 2001.
[18] S. Maoz, J. O. Ringert, and B. Rumpe, "Modal object diagrams." in European Conference on Object-Oriented Programming, ECOOP, ser. Lecture Notes in Computer Science,vol. 6813. Springer, 2011, pp. 281–305.
[19] A. A. Andrews, R. B. France, S. Ghosh, and G. Craig, "Test adequacy criteria for UML design models." Software Testing, Verification and Reliability, vol. 13, no. 2, pp. 95–127, 2003.
[20] J. Steel and M. Lawley, "Model-based test driven development of the Tefkat model-transformation engine." in International Symposium on Software Reliability Engineering, ISSRE. IEEE Computer Society, 2004, pp. 151–160.
[21] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. L. Traon, "Metamodel-based test generation for model transformations: an algorithm and a tool." in International Symposium on Software Reliability Engineering, ISSRE. IEEE Computer Society, 2006, pp. 85–94.
[22] J.-M. Mottu, B. Baudry, and Y. L. Traon, "Mutation analysis testing for model transformations." in European Conference on Model Driven Architecture, ECMDA-FA, ser. Lecture Notes in Computer Science, vol. 4066. Springer, 2006, pp. 376–390.
[23] L. C. Briand and Y. Labiche, "A UML-based approach to system testing." in UML, ser. Lecture Notes in Computer Science, vol. 2185. Springer, 2001, pp. 194–208.
[24] D. Amyot, L. Logrippo, and M. Weiss, "Generation of test purposes from use case maps." Computer Networks, vol. 49, no. 5, pp. 643–660, 2005.
[25] M. H. Pakinam N. Boghdady, Nagwa L. Badr and M. F. Tolba, "A proposed test case generation technique based on activity diagrams," International Journal of Engineering and Technology, IJET-IJENS, vol. 11, no. 3, 2011.
[26] Z. Jin and J. Offutt, "Deriving tests from software architectures." in International Symposium on Software Reliability Engineering, ISSRE. IEEE Computer Society, 2001, pp. 308–313.
[27] H. Reza and S. Lande, "Model based testing using software architecture." in International Conference on Information Technology: New Generations, ITNG. IEEE Computer Society, 2010, pp. 188–193.
[28] D. Kim, R. France, and S. Ghosh, "A UML based language for specifying domain-specific patterns," Journal of Visual Languages and Computing, vol. 15, no. 3-4, 2004.
[29] A. Causevic, D. Sundmark, and S. Punnekkat, "Factors limiting industrial adoption of test driven development: A systematic review." in IEEE International Conference on Software Testing, Verification and Validation, ICST. IEEE Computer Society, 2011, pp. 337–346.
[30] A. Z. Javed, P. A. Strooper, and G. Watson, "Automated generation of test cases using model-driven architecture." in International Workshop on Automation of Software Test, AST. IEEE, 2007, pp. 3–9.
[31] M. F. Aniche and M. A. Gerosa, "Most common mistakes in test-driven development practice: Results from an online survey with developers." in ICST Workshops. IEEE Computer Society, 2010, pp. 469–478.
[32] jMock – an expressive mock object library for Java. [Online]. Available: http://www.jmock.org
[33] Fitnesse acceptance testing framework. [Online]. Available: http://fitnesse.org
[34] S. W. Ambler, "Agile testing and quality strategies: discipline over rhetoric," [Online]. Available: http://www.ambysoft.com/essays/agileTesting.html
[35] M. Fowler, Analysis Patterns: Reusable Object Models. Addison-Wesley, 1996.
[36] D. E. Turk, R. France, and B. Rumpe, "Assumptions underlying agile software-development processes," Journal of Database Management, vol. 16, no. 4, pp. 62–87, 2005.
[37] S. R. Palmer and J. M. Felsing, A Practical Guide to the Feature Driven Development. Prentice Hall, 2002.
[38] R. Ramsin and R. F. Paige, "Process-centered review of object oriented software development methodologies," ACM Computing Surveys, vol. 40, no. 1, pp. 1–89, February 2008.
[39] Eclipse modeling framework. [Online]. Available: http://www.eclipse.org/modeling/emf/
[40] A. Qumer and B. Henderson-Sellers, "An evaluation of the degree of agility in six agile methods and its applicability for method engineering," Information & Software Technology, vol. 50, no. 4, pp. 280–295, 2008.
[41] M. Taromirad and R. Ramsin, "CEFAM: Comprehensive Evaluation Framework for Agile Methodologies," in Annual IEEE Software Engineering Workshop, SEW, pp. 195–204, 2008.