Applying TDD to Web-Application Development – Does Variety Help or Harm?

Wolf-Gideon Bleek, Sebastian Pappert
University of Hamburg
Dept. Informatics, Software Engineering Group
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
bleek@informatik.uni-hamburg.de, Sebastian.pappert@gmail.com

Abstract

This paper presents a case study which covers experiences gained from a Web development project employing agile techniques. Web development involves a number of technologies introducing a variety which is difficult to handle. Beside other agile techniques the project's participants followed test-driven development (TDD) in the project's various technology areas. TDD is used in addition to unit testing to positively influence the design of program artifacts. In doing so, they encountered problems, pitfalls and failures and developed strategies to have an overall test-driven approach. We report on the stony way exercising Test-Driven Development in a typical technology mix of a Web development project to learn about obstacles surfacing while pursuing design through Test-Driven Development. The case study reveals three strategies that programmers follow: avoiding to subclass from a framework, moving code to easy testable classes and keeping hard-to-test artifacts small.

1. Introduction

Testing Web applications is a problem which already gets attention from different directions [1]. A variety of unit testing frameworks exists that address the different technologies to overcome this problem. However, in this paper we focus on Test-Driven Development (TDD) since it not only helps us to write well tested software, reach high test coverage and so on, it also very well influences the design of programming artifacts. While unit testing is standard software development practice we want to utilize the potential of TDD with respect to design.

2. The Context – A Typical Web Development Project

To better understand the context of this problem area we firstly draft the object of this case study, the development project. Secondly, we introduce the underlying software engineering concepts relevant for this research. And thirdly, we will outline the development environment.

2.1. The Project

The experiences reported in this paper centre around a development project of a Web application[1]. This application has been developed since 1999 firstly in PHP. During that period it was not developed using unit tests or explicit agile techniques. Due to the popularity of Java and the lack of architectural tools that can help continuously monitor and refactor the old system we initiated a migration to the Java Web platform.

In this new Java Web development project (called the "migration project") we incrementally develop a Web-based Java application to gradually replace modules of the old system while running concurrently. The development project adopts an agile process which is mostly inspired by Scrum [2] and XP [3] using stories and tasks, stand up meetings, pair programming, shared code etc. and most importantly tries to develop by sticking to the Test-Driven Development rhythm *(see [4], [5])*.

The Java Web development project started out with a group of approximately 20 people. It was set up as a migration project with the long term goal to replace the running PHP system. As an initial decision Java technology was selected to create the new system. Main

[1] The CommSy project can be reached at www.commsy.de.

reasons for this technology decision were, among others, the rich set of available tools to monitor and ensure quality of the code base during development and an already existing knowledge among key project persons.

2.1.1. The Team Structure.

The project is carried out as an open source project [6] that is supported by a number of different groups. Among these groups are students participating in class projects, researchers and paid programmers. All of these meet at least twice a week working together. One key characteristic of this development group is that besides a constant group of about four people there is a constant fluctuation of team members. The typical length of stay in the development project is approx. one year. The alternating participants usually have little to no experience with TDD but do have some experience with agile development methods (e.g. pair programming).

2.1.2. Basis for Evaluation.

This research bases on static code analysis and individual code analysis and is supplemented by the experiences of the project's participants gained through interviews. Data on the Java project has been gathered since 2004. For evaluating the source code we have repositories available that allow retracing the actual source code configuration of each development step. We are therefore able to investigate in detail how the source code developed over time. Measures are calculated according to relevant points in time of the development project (e.g. before or after a major refactoring). The repository is available for both the PHP project and the Java project. However, results in this paper concentrate on the Java development only.

2.2. Related work

This paper focuses on an agile development process and evaluates the relation between Test-Driven Development, its effects on design and the number of technologies involved.

2.2.1. Unit Testing.

Unit testing [7, p. 239], [8] in general is an essential part of a basic quality assuring procedure. For each source code unit developers write a test that runs automatically. This tests all public features of the unit and will break if any feature changes behavior. Each unit test complies with a framework so that all tests can be collected and run sequentially. Due to this automation process we are able to run tests each time required (e.g. before a refactoring [9] or code is added and afterwards).

In accordance with Beck the following set of general criteria applies [10]:

- Isolation: A unit must be tested solely without using other operative code. No side-effects must occur. The order of testing must not matter.
- Simplicity: Writing tests must not complicate things in terms of programming language or technology involved. The development environment must (seamlessly) integrate writing and executing tests.
- Automation: Tests must run automated without user intervention.
- Speed: Tests must not consume much time in order to be executed frequently.
- Learnability: Developers must easily understand tests and their frameworks.
- Design: The technology must to a large extent allow test-driven development to design the interface of an artifact.

2.2.2. Test-Driven Development.

Test-Driven Development is an agile practice that goes beyond unit testing. As the name suggests it focuses on a process where tests drive the development. But despite its name, TDD is rather a design technique than a testing technique. This is accomplished by writing the test first and reacting by implementing the required operations in the artifact [11]. In consequence one designs the interface of that artifact reflecting the requirements of the test. In a typical development cycle, at first the test requires functionality that does not exist. The test cannot compile at this time and after fulfilling the syntactical requirements the test will fail during run. After that the implementation is written to suffice the test. Finally refactoring cleans up the artifact under development (see [4], [5] for an example of the TDD cycle here called the "TDD rhythm")

We favor TDD because it designs the interface of a program artifact from the test's perspective, reduces or avoids YAGNI[2] [3, p. 42], implements and tests code in small steps and the project's repository consists of well tested classes. In a complex project or an environment where new people are introduced on a regular basis it is of importance to have interfaces that are small and/or easy to understand. TDD supports achieving these kinds of interfaces. Furthermore, we feel that the overall design of the artifacts' interfaces has a bet-

[2] YAGNI: You Aren't Gonna Need It

ter quality [11] in terms of cohesion (high) and coupling (loose).

TDD can be practiced at initial development as well as during further development of a system. It should be well applicable for Web applications. Pursuing TDD only needs a unit testing framework that supports running tests repeatedly at the developers' workstation.

2.3. A Typical Variety of Technology

Web development projects tend to cover a significant number of technologies. This is not in contrast to other development projects. However, with the introduction of Web technologies the variety spans on multiple levels. Not only exist different formal languages for program and data description (Java and XML), there are also different programming languages (i.e. Java, JSP, EL), different control flow and state mechanisms (i.e. Servlets, Java objects) and the complex standards for visual presentation (i.e. HTML, XHTML, JavaScript). While on the one hand this variety helps to cope with ambitious requirements it also demands many qualifications from all project participants and poses a problem for a uniform testing strategy.

Taking aside technologies for special requirements and of course XHTML which is standard in today's Web applications, the Java platform alone introduces Servlets and JavaServer Pages (JSP), the Expression Language (EL) within the JSP and Tags (which utilize JSP and XML). All of these can be referred back to pure Java code but they certainly have their reason to be available.

2.3.1. Servlets.

Servlets[3] are Java's way of directing the flow of control into the hands of the application. Therefore, the application's entry point is an overridden method in a subclass of the Servlet class. This is straightforward for an experienced programmer of object oriented languages but has its pitfalls. Although you can implement this subclass stateful it is not recommended to do so because the container re-uses the Servlet objects in different contexts. Therefore, a set of objects handed over through the method's parameter called *request*, *response* and *session* represent the state. These objects model the state of the application since multiple instances can run at the same time utilizing the same Servlet object.

[3] JSR 154: Java Servlet 2.4 Specification (Final Release), 2003
http://jcp.org/aboutJava/communityprocess/final/jsr154/index.html

2.3.2. JavaServer Pages.

While JSPs[4] are written in an XML style language they may also contain segments of Java code. This mixture makes these documents hard to understand since one segment can reference variables that have been introduced in a previous segment at the beginning of the document (we call it the "mixture problem"). These properties of the technology can be better understood when looking at what happens behind the scenes: The Servlet container translates the JSPs into pure Java code implementing a Servlet subclass. Then it compiles the code in the background and the control flow continues after that in the compiled subclass. Therefore, JSPs behave like Servlets. This has to be kept in mind when writing JSPs since each Java snippet embedded in the XHTML document can access a common namespace and through it the Web application's context.

2.3.3. Expression Language.

The Expression Language (EL) is an enhancement of the JSP environment that helps resolve the "mixture problem" stated above. Using the EL avoids introducing Java code snippets into the XHTML document. The EL gives access to Java objects and Java operation calls. However, its syntax is cryptic (so that it complies with XML requirements) and EL implies that data transfer between code in the Servlet and the JSP is handled by Java Beans[5] (but this does not pose a problem since it is best practice). However, the naming scheme and identifying the corresponding elements is not obvious.

2.3.4. JSP Tag Libraries.

The tags introduced into the JSPs are a way of reducing duplicated code. They resemble subroutines and can be written using JSP syntax and EL or they can be provided as Java code. These tags are listed in an XML document to be available for reference.

2.4. The Development Environment

The integrated development environment (IDE) used for the case study's project is Eclipse[6]. This allows programmers to work effectively with the tech-

[4] JSR 152: JavaServer Pages 2.0 Specification (Final Release), 2003
http://jcp.org/aboutJava/communityprocess/final/jsr152/index.html

[5] Hamilton, Graham (eds.): Java Beans, Sun Microsystems, Version 1.01-A, 1997
http://java.sun.com/products/javabeans/docs/spec.html

[6] Eclipse IDE: http://www.eclipse.org

nology mix outlined above. Plug-ins exist for the chosen technologies and support working with them.

The project's integration server[7] [12] not only offers a shared code repository but also serves as a build and test server. Each check-in triggers a complete compile and test cycle and installs the new version as a test system. Reports on each build cycle (regarding build steps, tests and test coverage[8]) are available through a central development Web server.

3. Applying the TDD-Rhythm

In the case study, project development follows agile development practices, especially unit testing / test infected programming and aims at implementing the TDD rhythm as depicted by [4], [5]. This cycle enables people to quickly generate functional code and they achieve a high quality design. But as pointed out in Section 2.3 in this project participants have to deal with a number of technologies (program artifacts) typical for Web development projects besides pure Java classes. The most prominent among these were:

- Database Mapping
- Servlets and Filters
- Servlet Container Classes
- JavaServer Pages
- Custom Actions (Tags)

These different technologies hinder the team members implementing the TDD rhythm. The first problem is that unit testing frameworks are not as easily available for these technologies as for pure Java code. Secondly, the different unit testing frameworks do not integrate seamlessly into the development environment. Thirdly, different handling mechanisms make it cumbersome to run tests as often as necessary for TDD. And finally, even though testing frameworks are ultimately available, they may not directly support a TDD strategy. In the following subsections we will systematically investigate how the criteria listed above (section 2.2.1) apply to the different kinds of technologies that we encountered in the project.

These different technologies hinder the team members implementing the TDD rhythm. The first problem is that unit testing frameworks are not as easily available for these technologies as for pure Java code. Secondly, the different unit testing frameworks do not integrate seamlessly into the development environment. Thirdly, different handling mechanisms make it cumbersome to run tests as often as necessary for TDD. And finally, even though testing frameworks are ultimately available, they may not directly support a TDD strategy. In the following subsections we will systematically investigate how the criteria listed above (section 2.2.1) apply to the different kinds of technologies that we encountered in the project.

[7] In this case CruiseControl is used: http://cruisecontrol.sourceforge.net/

[8] Test coverage is metered by Cobertura: http://cobertura.sourceforge.net.

Table 1. Artifacts and their basic properties in the project (current)

Type of artifact	LoC	# artifacts	tested
Java classes			
Pure Java classes	11072	128	57.8%
Beans	2177	31	0.0%
Database mapping	2084	58	69.8%
Servlets	433	6	5.1%
Filters	589	4	21.2%
Servlet Container	5578	87	15.4%
JavaServer Pages	2863	40	n/a
Custom Actions (Tags)			
Java classes	581	8	43.6%
Tag files	858	21	n/a

3.1. Pure Java Classes

Pure Java classes are a well supported area of TDD and provide the basic reference line for other technologies and their artifacts. This type of artifact forms approx. 50% of the system in this case (not considering JavaServer Pages, see below in section 3.4) and is mainly used to implement domain values, data items, and utility classes. In this project, the test coverage for this type of technology is nearly 58%. However, not all tests have been created within a TDD rhythm.

All of these classes can be tested in isolation. Tests are written in the same language within the development environment using the JUnit framework [8], [15]. JUnit is fully integrated into the Eclipse IDE. This provides basic automation at the workstation and it is quick enough to be run repeatedly. Except for the usage of JUnit, developers had nothing additional to learn. The interface of classes can be designed by applying the TDD rhythm.

3.1.1. Experiences.

The type of technology and the type of classes written facilitate identifying the corresponding test. It is also clear what to test and how it is to be tested.

It is imperative that tests are easy and quick to run in order to keep programmers performing the rhythm. If tests are an obstacle during the development process these tests are usually the first thing to be dropped.

The project's experience is that TDD can be applied smoothly but new team members only reluctantly adhere to it.

3.1.2. Reflection and assessment.

The analysis of these classes' tests shows that even though a significant number of JUnit tests exists and the test coverage is good some tests which were intended as pure unit tests were in fact integration and acceptance tests. One significant problem here is that tests are not written for isolated classes (e.g. by utilizing mock frameworks).

Besides the technical qualification for writing a test with JUnit, programmers also need training in how to write a meaningful test. Neither TDD nor the unit testing framework can assure this.

Regarding the design aspect of TDD, programmers can make use of the full potential of TDD. There were no restrictions on the interface of classes.

3.2. Servlets and Filters

Servlets are the centre of Java Web applications and control the interaction with the user. They consist of two parts: a class file implementing the behavior and a configuration that makes the Servlet accessible through the Servlet container and configures the application. This configuration depends on the Servlet container. For the Tomcat Servlet container[9] the configuration is stored in XML files. Filters are a light-weight realization of the *Intercepting Filter* pattern [16, pp. 144-165] and very similar to Servlets in implementation and configuration.

In this particular case, Servlets and Filters are less than 5% of the system. Of these classes the coverage is only 20%. Tests are written using the Cactus testing framework [17] following a JUnit mechanism but allowing the object's execution in a Servlet container environment.

3.2.1. Experiences.

A Servlet object relies on a request, a response, and a context object which the container provides. This implies a substantial initialized object structure to perform the test. Either that object structure needs to be replaced by mock objects or the tests have to be executed in a Servlet container using a framework like Cactus. The project follows the latter approach.

Regarding simplicity tests can be written in the same language in which Servlets are programmed. However, it involves introducing a new testing framework which should comply with established standards: writing and executing the tests needs to be integrated in the development environment. In this case, tests can be automated because they are also JUnit tests making them part of the project's test suite.

Due to the fact that part of the test is executed on "the client side" and another part is executed on "the server side" tests get more complex but are still understandable.

There are multiple levels of problems testing the Servlets: the container characteristics of the request and response objects, the sequence of operations in the main Servlet, and the session representing the status of the calling context.

3.2.2. Reflection and assessment.

This particular project adopts a Servlet structure to work as a single entry point into the Web application serving as a collecting instance that hands over to other (non Servlet) parts of the system. This complies with the MVC2 pattern using a so-called "front controller" [16, pp. 166-180]. The goal is to reduce the Servlet's function to collecting the necessary information, hand them over to a transforming instance and collect the result and output for transfer to the user.

Even though Servlets are a centre point of Web application development a common approach for Web applications is to avoid them. The strategy is to keep the number of Servlets of an application to a minimum.

Most of the operations of a Servlet are inherited. Application specific actions take place in one or two methods. Testing these methods is an obstacle because there is a lot of context to initialize and reproduce in order to perform the test. Moreover, the operation usually gets complex during development.

Because the interface of Servlets and Filters is predefined, in this respect TDD cannot contribute to the design of the interface but to the quality of the code. The Servlet's interface does not evolve through development since there are only dedicated hook methods to specify. A good strategy is to move code from these classes to places where it can be tested properly and classes' interfaces do matter and can be designed.

Instances of the Servlet classes are not first-class objects. They are not allowed to have instance variables since the Servlet container may re-use them, they run non-deterministic in different threads, and are challenging to test in full isolation. The latter is due to the fact that complex request, session, and response objects have to be initialized to enter a Servlet method. The major drawback is the single method to which control is handed over when the Servlet is entered. This method may degenerate to a huge main method of the application (the code repository of the project shows this effect in older versions) encapsulating most

[9] http://tomcat.apache.org/

of the functionality and hiding it from dedicated testing.

Servlet programming requires access to the requestor session object which hands over vital information about the current status of the application as well as the requested action and its parameter (they replace the instance variables one would typically find in classic objects). The processing in the Servlet's method retrieves and evaluates elements and stores new elements. While this is normal programming style within a Servlet environment it poses a significant burden on testing because it obfuscates the interrelation of these elements. Implicit preconditions and interdependencies occur: e.g. a value needs to be available in the session object before another can be calculated. While in classic code this can be determined and resolved at compile time in a Servlet the situation will arise at runtime. This does not support a TDD style because dependencies are rather in the sequence than reflected by the class' interface. In consequence, programmers should avoid putting application specific code into these classes.

3.3. Servlet Container Classes

Servlet container classes (SCC) are essentially pure Java classes that read or manipulate the Servlet container's context objects request, response, and session. These classes serve as a mediator between the Servlet classes and the pure Java classes. The coupling with the Servlet API is existent but not as extensive as the Servlet classes are coupled with the Servlet API [18]. SCC make use of types introduced by the Servlet API but they do not rely on inheriting classes or interfaces of the Servlet API. Approx. 30% of the system is in this category with coverage similar to the Servlets.

3.3.1. Experiences.

Since the syntactical design of SCC is equivalent to pure Java classes all experiences of the developing process of pure Java classes apply to SCC. The testability of these classes solely depends on the design. Unlike Servlets and Filters, SCC are not bound to be run inside a Servlet container. However, the usage of context objects requires at least a minimal implementation of the context classes that the function under test operates on. The context classes are rather complex and therefore the simplest solution is to use the already introduced framework Cactus. In this case study the response object is never manipulated in SCC which reduces the complexity of SCC compared to Servlets. The request and the session objects can be seen as a secondary interface. Without limitations for type and amount of the data in the context objects, tests describe the secondary interface. Developers can reuse the previously learned knowledge of writing tests for Servlets and the tests are seamlessly integrated into the JUnit framework.

3.4. JavaServer Pages

JSPs are basically XHTML documents with special tokens that allow insertion of programming constructs. During run-time these JSPs are transformed to Servlets. While the XHTML text is translated to println commands on the response stream, the inserted statements and code are directly passed through. From the programmers' point of view JSPs look like a completely different programming environment since they have to utilize a different formal language and have a (prepared but implicit) environment which they can (or must) use to access data (e.g. for outputting values).

3.4.1. Experiences.

In terms of testing we need to investigate the question whether a JSP is a proper unit for testing. We need to identify the public operations, to find out what interface actually is developed or, with respect to TDD, designed. JSPs certainly need testing and can be seen as a unit of the whole application. However, the artifact has no explicit interface or – if seen as a Servlet – there is a predefined and fix interface.

Moreover, a JSP can hardly be tested in isolation. Like Servlets it needs an environment which is provided by request, response and session objects. Bean objects need to be available at compile time.

Interaction with JPSs is possible through a Web request which triggers the compilation and returns an XHTML stream. This stream is either an error report that indicates problems during compilation or runtime or it is the generated XHTML which can then be checked for syntax and content.

Due to the request, the compilation, and the stream result testing JSPs is significantly complex. The test may break for various reasons and unit tests cannot easily be integrated into the overall testing framework. A major question – as with Servlets – is whether JSPs fall in the area of artifacts that are reasonably developable through TDD. Since there is no interface to design most things to test are standard with all JSPs (see above).

Rudimentary tests for JSPs can be automated. The test can request a page described by a JSP and evaluate the XHTML stream. If errors exist the test may fail. However, this kind of test is significantly slower and requires an application server to run. It is also hard to extend for tests that go beyond these characteristics.

If testing is performed there are multiple levels of what can be tested: there is XHTML testing regarding the syntax, testing of the control flow (e.g. are iterations over collections performed correctly?), and whether all necessary data are available for processing (e.g. is a specific element necessary for display provided by the session object?).

Due to the nature of JSPs we need to investigate if it is at all reasonable to pursue a TDD approach. How could TDD help during development, since the strong sides of TDD are the design of a units interface and to ensure correctness against the test? We do not see how TDD can contribute to e.g. testing the initialization of a variable. Typical problem categories comprise the existence of getter operations and whether a Bean has been properly initialized.

Since we have a hard time formulating preconditions for a JSP we are hardly able to design tests in a unit test fashion. In the evaluated project a design rule to keep algorithmic code away from the JSP (and from Beans and tags also) has been developed.

3.4.2. Reflection and Assessment.

As we have shown above, JSPs do not have an explicit interface that can be tested or even designed. Still, these units of the application need testing on various levels but a test-driven approach is not the adequate way of development for these elements. Since JSPs do not offer separate methods for which we can assign pre- and post-conditions the cause for or location of an error cannot be identified easily. Unit testing beyond syntax errors is hard to achieve (while integration and acceptance tests are easy). To reach a maximum level of unit testable elements it is important to not let algorithmic code get into JSPs. This should be handled by classic Java classes and elements that have a testable interface.

3.5. Custom Actions

The JSP specification provides developers with a basic set of standard actions to enhance JSP pages with logic hidden behind XML syntax. These standard actions can be extended by custom actions either by creating a Java class that follows the tag extension mechanisms of the JSP specification or by writing tag files in a JSP syntax.

3.5.1. Experiences.

In particular, the implementation of custom actions as Java classes confronts developers with two points of view for the interface. First, the Java interface for the class file is only of interest to the Servlet container and bears no possibilities for design decisions. Second, the XML interface is important for the actual user of the action. The Cactus framework provides mechanisms similar to the one for Servlets to test the Java interface of custom actions. It is also possible to test the XML interface by embedding the action in a simple JSP and utilizing the testing mechanisms for that type of artifacts. This approach integrates easily in the development process of the other units tested with Cactus but comes with the cost of splitting test cases into two separate files. The input for a test case is defined in the JSP whereas the output is verified in a Java file. A different approach would be to use a framework that allows writing test cases for custom actions completely in the JSP syntax, e.g. TagUnit[10]. TagUnit does not work together with the JUnit framework and therefore does not fit into the established development process. It can be executed automatically in an Ant script but unlike Cactus TagUnit is not supported by IDEs and the integration into a build server would need manual adjustments.

3.5.2. Reflection and Assessment.

As we have shown above, similar to Servlets, custom actions do not offer any degree of freedom with respect to the interface that is to be fulfilled. Therefore, the major strength of TDD is not applicable. This only leaves the question of how the best testing approach can be achieved. While implementing custom actions in JSP syntax is compact and easy to write it requires discipline and understanding when it comes to realizing the test. Additionally, test coverage cannot be measured easily. Therefore we would strongly suggest implementing custom tags in Java.

4. Evaluation

In this section we will generalize the experiences of the different technologies collected in the empirical study and take a look at the concepts. After that we distinguish different levels of testing and finally explicate the contribution of the different testing approaches to these levels.

4.1. General Problems

Web applications introduce a set of technology that complicates testing and in particular TDD thus not allowing taking advantage of the design contribution TDD provides. As shown above, each technology requires a different view on testing, a different unit test-

[10] http://tagunit.sourceforge.net

ing framework and poses new challenges on how to develop code.

On top of that, even with the proper testing framework for each technology the approach to testing is slightly different. Programmers are required to learn the differences and have to implement different strategies of testing (local unit, client-side, server-side).

For example testing Servlets and Filters has not been well understood by programmers in this project in comparison to pure Java classes because:

- the designation of Servlets and Filters is not clear,
- programmers are confused by the control flow and the change of means of expression (application server, Filter, Servlet, JSP, Java class),
- in contrast to "normal" unit tests the number of test methods is increased (up to seven),
- Servlets and Filters make use of implicit objects that are handled differently in each of them,
- execution during tests takes place on the client and on the server side (people have to know that while writing the test and analyzing a broken test),
- testing "normal" Java classes can be run by a keystroke or mouse click while testing Servlets and Filters requires a fully set up and running environment.

As a consequence, these tests are reluctantly implemented and operated.

Not all testing frameworks allow testing the unit in isolation. As shown by the JSPs we are not able to mock all objects of the JSP's environment (e.g. other JSPs included, custom actions).

With respect to simplicity it is ambiguous to introduce a number of additional testing frameworks. While on the one hand they enable testing of other technologies they on the other hand demand additional learning and may confuse due to (slightly) different handling.

At least the aspect of test automation can be accomplished for all technologies discussed. This is currently not an open field. Also, speed is not a relevant topic even though testing JSPs imposes additional compiling.

Design with respect to the interface of an artifact is a weak topic in the field of Web application development and TDD because most of the interfaces of the technologies introduced are fixed. To access functionality interfaces have to be met and developing them further is counterproductive.

4.2. Levels of Testing

Even though all testing frameworks make use of the unit testing approach they do not – in fact – have the same level of testing. The least common denominator of all frameworks simply is that they test a single artifact. For this artifact test methods can be specified. Whether these test methods provide tests for each method of a class or offer different request types for a Web page is not specified further.

In this project we have identified at least three different levels of testing:

Method by Method Test. Java classes allow tests that provide a test method for each (relevant/public) method of the unit under test. TDD can govern the process in a way that influences what methods emerge and how their signature is designed.

Single Entry Test. Some tests, like e.g. JSP tests, simply request the result of that specific unit to evaluate the result. In some instances the request has parameters that may influence the result. A common theme to these tests is that there is only one entry point.

Disguised *-Test. While many of the so called tests come as unit tests they aren't in fact unit tests. They bear the name to signal that they can be integrated in a JUnit run. HttpUnit, for example, is not a unit test in comparison to JUnit simply because it does not test a unit in isolation.

4.3. Coping with Web Technology – Overcoming the Major Problems

Since the case study identified a number of problematic characteristics of the technologies involved in terms of TDD it also outlines strategies to minimize or overcome these problems.

Strategy 1: Avoid Subclassing the Framework. Keep the number of classes that subtype from a framework's class as low as possible because the interface of such classes is pre-defined and designing it further is not intended. Since testing Servlets is not easy and TDD mostly not applicable they should – from the standpoint of TDD – be avoided. This has been achieved in the project by reducing the number of Servlets to a minimum. Currently the project contains five classes that inherit the Servlet interface. This will be reduced to three leaving one actual Servlet and two super classes.

Strategy 2: Move Code to Testable Classes. The second and corresponding strategy is to introduce objects of Java classes that are not joined with the Servlet class. Move as much code as possible out of artifacts that pose a testing problem and implement using pure Java classes. Delegate from the special classes as early as possible. In this way, full TDD can be applied: the object has a relevant interface, the object has a state and basic test principles apply.

Strategy 3: Keep Hard-To-Test Artifacts Small. Keep the size of artifacts that cannot be tested small (in contrast to a few large ones). As discussed above, testing JSPs is a similar problem as with Servlets but with the added handicap that JSPs are typically not segmented into methods. While for Servlets a valid strategy can be to avoid them this may be more complicated for JSPs because they are a good way for formatting output. Other solutions, like transformation frameworks introduce an ample amount of learning and complexity.

Therefore, the project's strategy is to keep JSPs as small as possible to have the unit under test manageable. In this case study there were 15 JPSs with an average size of 149 and a peak of 310 LoC before refactoring. After following this strategy the number increased to 40 with an average size of 72 and a peak of 190 LoC.

5. Discussion

Analyzing the project has shown that each new technology introduced into the development process is likely to require a specifically tailored test framework. For those technologies where frameworks are available they have at first been used to implement tests but not all have proven an optimal solution. For some technologies sufficient testing support simply is not achievable with acceptable effort. In any case, special knowledge is necessary not only for handling the technology but also for writing the tests.

For a project team whose goal is to have sufficiently tested artifacts for all technology areas and establish a good design for all artifacts strategies have to be established that can be used during software development. The strategies presented above were identified through the project's analysis. Their common theme is to reduce the code written in technology that is hard to test and move code to technology that offers an easy to use and established testing framework.

One deficiency of the depicted collection of strategies is that one cannot get rid of the different testing frameworks completely. The team still needs to have the knowledge about the frameworks and their testing strategy available in the project.

What has been achieved? Rather than pursuing a way to make TDD available for all technology areas the study shows that it is advisable to carefully investigate whether new technology needs to be introduced. If so, it needs to be determined how much must be written using this technology and how much can be moved to "normal" code.

In this respect this research has identified in what areas new technology hinders designing the artifact through TDD. It also presents a viable strategy to elegantly reduce the restraining effects of new technology. In total, by concentrating on the artifacts that can be designed by TDD developers can make the most of it.

Testing is not the only domain in which moving to a well established base is helpful. There are a number of tools available for pure Java code which is hardly found for the other technologies. Examples are, to start with, test coverage, static code analysis (measuring, identifying bugs), dynamic code analysis (profiling). Even though some areas may be covered in the future it will always be easier to work with a main technology in contrast to a set of different technologies.

The approach outlined in the evaluation can possibly be transferred to other programming languages and to other areas of software development. However, this research is limited to the Java environment and a single Web development project.

6. Conclusion

In this paper we have outlined a case study of a Web development project in Java. This project makes use of the standard Java technologies like Servlets, JavaServer Pages, the Expression Language, Tags and Custom Actions for implementing a Web application. With a test-driven approach one needs special support in the area of unit testing. The case description shows that each technology besides pure Java classes has its drawbacks and requires special support (in the form of a framework) to implement tests and in consequence profit of the design capabilities of TDD. Some drawbacks hinder a TDD approach and some unit testing frameworks do not allow literal unit testing.

The case study shows approaches on how to cope with these deficiencies. The first approach centers on avoiding special Web related technology. Servlets are an example where reducing the number of classes and reducing the lines of code has effectively allowed for more testing. This is, in part, achieved by the second strategy that accompanies the first. Moving code to separate testable classes. JSPs are an example of another strategy. They can practically only be tested as a single entity. To make testing more precise these entities have been reduced in length. The number of artifacts more than doubled but it had a positive effect on modularization and testability and has had hardly any effect on the total number of lines.

We can conclude that the key message of this case is a most obvious one: reduce the number of technologies to simplify testing during development. In addition, both strategies, avoiding problematic technology and splitting hard to test artifacts in small pieces, are part of a feasible way that leads to a better testable architecture. This also allows circumventing the problem of driving the development of artifacts that do not have an interface (in this case JSPs). Since designing the interface in terms of methods is not applicable, at least small artifacts stand in.

With regard to the design of artifacts the message is to focus on established artifact types and avoid elements with the characteristics shown here.

The identified approach combining these three strategies leaves the problem that by utilizing a set of problematic technologies there is always the chance of growing artifacts of these types that evade an established and practical testing strategy. Further research on this needs to be done.

References

[1] E. Hieatt and R. Mee, “Going faster: Testing the web application,” IEEE SOFTWARE, pp. 60–65, March/April 2002.

[2] K. Schwaber and M. Beedle, Agile Software Development with SCRUM. Prentice Hall, 2001.

[3] K. Beck, Extreme programming explained: embrace change. Amsterdam: Addison-Wesley Longman, 2005.

[4] D. Astels, Test-Driven Development: A Practical Guide. Upper Saddle River, New Jersey, USA: Prentice Hall, 2003.

[5] G. Doshi, “Test-driven development rhythm,” Instrumental Services Inc., http://www.testdriven.com/modules/mydownloads/singlefile.php?cid=3&lid=3, Tech. Rep., 2005, last visited 12-oct-2007.

[6] W.-G. Bleek and M. Finck, “Ensuring transparency – migrating a closed software development to an open source software project,” in Proceedings of the 28. Conference on Information Systems Research in Scandinavia. Kristiansand, Norway: IRIS, 6-9 August 2005.

[7] A. Hunt and D. Thomas, The Pragmatic Programmer – From Journeyman to Master, ser. The pragmatic programmers. Boston: Addison-Wesley, 2000.

[8] ——, Pragmatic unit testing: in Java with JUnit, 2nd ed., ser. The pragmatic programmers. Raleigh, NC: Pragmatic Bookshelf, 2004.

[9] M. Fowler, Refactoring: improving the design of existing code. Boston, MA. USA: Addison-Wesley Longman, Inc., 1999.

[10] K. Beck, Test-Driven Development by Example. Boston, MA: Addison-Wesley, 2003.

[11] ——, “Aim, fire [test-first coding],” IEEE Software, vol. 18, no. 5, pp. 87–89, Sep/Oct 2001.

[12] M. Fowler, “Continuous integration,” Thought Works, Tech. Rep., May 2006, http://www.martinfowler.com/articles/continuousIntegration.html, last visited 12-oct-2007.

[13] D. Willmor and S. M. Embury, “An intensional approach to the specification of test cases for database applications,” in ICSE ’06: Proceeding of the 28th international conference on Software engineering. New York, NY, USA: ACM Press, 2006, pp. 102–111.

[14] ——, “A safe regression test selection technique for database-driven applications,” in ICSM ’05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM’05). Washington, DC, USA: IEEE Computer Society, 2005, pp. 421–430.

[15] E. Meade, R. C. Martin, O. Mentors, and J. Kohnke, “Junit project website,” http://www.junit.org, 2007, last visited 13-jun-2007.

[16] D. Alur, J. Crupi, and D. Malks, Core J2EE Patterns: Best Practices and Design Strategies, 2nd ed., ser. Core Design Series. Prentice Hall / Sun Microsystems Press, 2003.

[17] A. S. Foundation, “Cactus project website,” http://jakarta.apache.org/cactus, 2005, last visited 13-jun-2007.

[18] C. Szyperski, “Independently extensible systems. Software engineering potential and challenges,” in Proceedings of the 19th Australasian Computer Science-Conference, Melbourne, Australia, February 1996.