Using Spring Roo for the Test-Driven Development of Web Applications

Tim A. Majchrzak  
Department of Information Systems  
University of Münster  
Münster, Germany  
tima@ercis.de

Andreas Simon  
Independent  
Münster, Germany  
research@andreas-simon.info

ABSTRACT
Software should be developed in a way that achieves high external and internal quality. Most programs have to be maintained or are extended. They should not only adhere to their requirements but allow to be efficiently changed. Model-driven software development (MDSD) and test-driven development (TDD) are techniques that help to satisfy this demand. We present a novel approach combining MDSD and TDD while using the Spring Roo Web development framework. Besides discussing existing work and the background, we sketch a test-driven Web development process. We then evaluate its feasibility in an exemplary scenario. Thereby, we contribute to the body of scientific knowledge on Web development frameworks in the field of MDSD and TDD.

Categories and SubjectDescriptors
D.2.2 [SOFTWARE ENGINEERING]: Design Tools and Techniques—Software libraries; K.6.3 [MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS]: Software Management—Software development

General Terms
Design, Experimentation, Performance

Keywords
test-driven Development, TDD, Web application, Web development, Spring, Spring Roo

1. INTRODUCTION
Only rarely, software is programmed and used without changes until it is put out of service. Since defects are found after it is rolled out, requirements change, compatibility issues arise, new platforms have to be supported, and for many other reasons, most programs require maintenance. Software undisputedly ages [43]. Typically, total costs for maintenance greatly exceed development costs [37]. Even if maintenance is not caused by a lack of functional quality—i.e. there has been sufficient testing—costs can be reduced by developing software in a style that aids maintenance.

Two programs with the same functionality and quality as seen from a users’ perspective (external quality) might be different in internal quality. Thus, the effort to alter them differs. Developing Software with future changes in mind prepares it for maintenance and extension. However, what fosters internal quality is not obvious. Typical design decisions are modularity vs. simplicity and reusability vs. specialization; their impact is subject to discussion [7].

Model-driven software development (MDSD) has been proposed as a method to increase internal quality while enabling efficient development. The idea is to create programs from an abstract representation (model). Model-driven development is becoming increasingly popular for Web applications. Spring Roo is an example for a Java-based framework that can be used for MDSD of programs that are deployed to Web servers or application servers. Another technique that seeks to increase external and internal quality of software is test-driven development (TDD). It refrains from an approach in that coding is followed by testing. Code is implemented after the corresponding test case has been written. Therefore, TDD is also called test first development [8]. Programming is finished if the test case can be executed successfully [16].

Our research question is: is it possible to develop Web applications in a model-driven way while working test-driven? Besides this question, we wanted to have a closer look at Spring Roo, which is a novel framework for MDSD and has not been scientifically assessed, yet (see Section 2).

Our work makes several contributions. Firstly, it shows how Spring Roo can be used for TDD. Secondly, findings are generalized to make our work both relevant and rigorous. Thirdly, we comprehensively describe particularities of combining MDSD and TDD for Web development. Fourthly, we integrate our work into the prevalent research and explain different threads of work currently followed.

This paper is structured as follows. In Section 2, we discuss related work. Our work’s background is compiled in Section 3. Section 4 explains our proposal for test-driven development using Spring Roo. A exemplary scenario and an evaluation of our work is given in Section 5. Finally, we draw a conclusion and highlight future work in Section 6.

2. RELATED WORK
There is a myriad of papers on MDSD—which also uses to be called model-driven engineering (MDE) [53]—and TDD. In fact, both approaches are well-studied and even textbooks
have been published [54, 8]. Hence, we will focus our liter-ature review on work that integrates MDSD and TDD, and on work that deals with model-driven or test-driven Web development frameworks. Finally, general papers on Spring Roo can be compared with our work.

The approaches that integrate MDSD and TDD, which we know of, can hardly be compared to our work. ZHANG for example uses a test-driven modeling approach in MDSD [57]. While this is an integration of the two concepts, it is different in aim to our work. Some other approaches are far more specialized such as work on model-transformations based on a test-driven method [28]. There also are approaches that combine TDD with model-based testing such as [56]. However, we deem model-based testing and MDSD two independent approaches even though both rely on models and the same models can be used for testing and coding purposes.

In the context of Web development, MDSD [29, 52, 20] and TDD [46, 48, 33] are actively discussed. The work published in recent years is the foundation for our work since it illustrates the feasibility of using MDSD and TDD when developing Web applications. However, neither of these approaches directly fosters our work.

Work on Spring Roo still is relatively limited. This particularly applies to a scientific examination of it. While it is dealt with in two textbooks on Spring from 2010 [36, 23], Roo is not included in older titles. This is not surprising since its first version was released in late 2009 [4]. A practitioner’s book has been published in August 2011 [34] with additional titles announced for winter 2011 [51, 47]. While being mentioned in some scientific papers, we only identified one that uses Roo as a base for own work. CASTREJON et al. present a framework for model-driven Web development that extends Spring Roo [14]. We expect more papers that deal with Roo to appear while our work is under review.

Finally, we identified one approach that has a connection to our work due to dealing with MDSD, TDD, and Web applications. ROBLES LUNA et al. “combine the agile, iterative and incremental style of test-driven development with the formal, transformation-based model-driven Web engineering” [49]. Thereby, their work is process-oriented whereas we focus on a framework directly. Thus, their approach and ours are complementary. In a newer paper, their work is extended to include usability requirements [50].

Additionally, the use of scripting languages such as Groovy [18] and JRuby [41] is notable. Since scripting languages have been ported to the JVM, they become wider recognized [11] in Java-centered companies. Scripting languages allow programming with less syntactical constraints. While they are not directly related by general idea, learning from research on scripting languages might improve our work.

3. BACKGROUND

To motivate TDD of Web applications, we describe the background of technology and methodology.

3.1 Particularities of Web Applications

Web applications deliver functionality through browsers via the World Wide Web. They are constrained by the architecture of the Web and technologies such as HTTP, HTML, and JavaScript that limits the freedom for implementation.

Although the possibilities of Web developers are limited, Web applications apparently have become the most important technological platform for new applications because of lower maintenance cost compared to locally installed applications (typically rich clients). Moreover, Web applications are accessible all over the world. These possibilities found their interim peak in software as a service (SaaS) offers [2]; applications are delivered exclusively via the Web [12].

Users’ expectations in terms of business functionality, usability, and user experience have raised enormously [55]. This makes Web applications very complex. They are hard to maintain and evolve. Some of the requirements are countered by the invention of the Ajax (asynchronous JavaScript and XML) technology that enables asynchronous communication between the browser and the Web server. The HTML document can be updated by means of JavaScript.

The magnitude of available browsers complicates the situation even further. Web applications are written once but have to respect the peculiarity of each of the browsers. The browsers differ—partly dramatically—in the implementation of the Web standards such as HTML and the layout. This led to the so called browser wars [39]. In addition, computers use different screen sizes and the upcoming, Internet-enabled mobile devices make the situation even more complicated. Developers have to ensure in an effective and efficient way that applications are executed correctly by combinations of different browsers, screen resolutions, and devices. That makes regression testing a challenging endeavor.

3.2 Test-Driven Development

KENT BECK proposed Test-Driven Development (TDD) in the course of Extreme Programming [6, 7]. Today, it is a central practice in agile software development [40]. TDD is a micro-process for the implementation of new functionality. It comprises three phases (see Figure 1). The first phase is testing: Write a failing (unit) test. It might be impossible to compile it because invoked methods are not yet implemented. The failing test motivates the incremental implementation of missing functionality. This is similar to an executable specification [8]. Coding follows testing. Add just enough production code to pass the test (and all formerly existing tests) [8]. This might provoke bad design but that is not in focus in this phase [37]. Finally, designing (refactoring) is conducted. Eliminate all the duplication created in the former step until the new code is cleaned up. It is an explicit design phase [8]. Refactorings are small changes to the internal structure of a program without changing their external behavior [42]. Through refactoring, non-functional attributes like simplicity, flexibility, understandability or performance can be improved systematically.

After one cycle through the three phases, a new cycle is started with the next test in order to add more functionality. By cycling over and over again the functionality of the product is built up. The growing number of automated tests avoids regressions. Through refactoring the internal quality of the software is maintained or even improved.

The foremost goal of TDD is high product quality. New code should never influence existing functionality. Every
accepted test execution improves the quality assessment since a machine always executes the same steps in the same sequence; a human tester occasionally errs. In addition, a stultifying play-back of a test script is not a very satisfying work and leads to a decline of the employee motivation [21]. Furthermore, automated programmer tests advance better design. The better the design is, the easier unit testing becomes. Programmers are motivated to develop highly cohesive, loosely coupled components to simplify their work [8]. The additional activities increase development time by 15% to 35%. In return, the overall product quality is improved; the error rate can be reduced by 40% to 90% [40]. These observations have been verified experimentally [13].

Development efficiency can be improved through automation [21]. More tests can be executed more often because test execution is much cheaper compared to manual testing. Especially regression tests [45] become more economic. Frequent execution in test cases advance the trust in the system stability. By a frequent, almost continuous test execution errors are detected very soon after their creation, and fixing them is relatively cheap. This lowers the cost for removing defects [8]. Finally, automated tests can serve as a communication medium. Well written tests express the functionality intended by the developer. They serve as an automated specification for the respective component. Other developers can get a quick grasp of the intents of the primary developer. Tests save and transport knowledge within the team [13].

Test-driven development is constrained to implementation and design activities carried out by the developers. There is no integration of the customer. Through acceptance-test-driven development (ATDD) the principles of TDD are extended to the whole development process [31].

In agile projects, requirements typically are captured as user stories. They define only the scope of a requirement, not the actual specification. The user story with the highest priority is picked and specified in the form of one or more acceptance tests [31]. Acceptance tests are owned by the customer, written together by customer, developer, and tester, address the what and not the how, expressed in the language of the problem domain, and concise, precise and unambiguous [17]. It is vital to write readable and understandable acceptance tests. Since customers shall be actively involved in requirements analysis, they must be able to at least understand the acceptance tests. Acceptance tests are the central communication medium in ATDD.

Eventually, acceptance tests are automated. All automated acceptance tests form the executable product definition” [31]. Often acceptance tests are realized as black box tests going end to end. The system as a whole is configured almost like in production. It is executed through the top layer (usually the user interface) and involves each of the application’s components. In effect, the acceptance tests are likely to harness most of the technical and logical problems. However, executing functionality through the user interface is relatively slow, which deteriorates the feedback speed. Because automation requires a very high level of concretion, there is lower risk for misinterpretation and misunderstanding between customer and developer.

Finally, the specified functionality is implemented in order to pass the acceptance criteria. Programmers benefit from the unambiguous semantics of automated acceptance tests when they implement functionality. Working from the

![Figure 2: Manual and automatic code generation](image)

outside in also is one of the goals of **behavior-driven development (BDD)** [15].

### 3.3 Spring Roo

Spring Roo (or just Roo) is a round-trip code generator framework [44] that generates most of the artifacts needed in Spring-based Web applications [32]. It is divided into a core and several extensions. The core provides a meta model for the project artifacts and tools for code generation. Based on formal models, namely the entity classes annotated with meta data for the code generators, the application is generated. Thus, Spring Roo is one implementation of MDSD.

While the core provides only the basic infrastructure, additional useful functionality is provided by **add-ons**. There are **base** add-ons and **third-party** add-ons. Base add-ons are delivered together with the Roo distribution. They support the generation of model classes, persistence components and Web application components. They facilitate the quick development of prototypes based on scaffolding [45]. Third-party add-ons can be installed from any URI with a special command via the Roo shell (see below).

The user is guided by the **Roo shell**, a custom command line interface (CLI). By means of Roo shell commands the user can generate the annotated domain model classes and necessary XML settings. This is called passive code generation. The alternative is called active code generation. Roo constantly monitors the class path. When a model class is created, changed or deleted, the corresponding meta data is updated and the depending artifacts (e.g. Inter-Type Declarations or XML files) are synchronized. This comprises aspects, controllers, and views. If representations change, Roo reserializes the artifacts and ensures their consistency. The two ways of code generation [5] can also be called **manual** and **automatic**. They are visualized in Figure 2.

The architecture of applications that are generated by Spring Roo can be described by two orthogonal axes (Figure 3). Vertically, the application is structured into two tiers [25]. The entity layer comprises data access functionality and business logic. The presentation layer is responsible for rendering Web pages. Horizontally, each tier is partitioned into a annotated class stub and multiple aspects [47]. These aspects provide additional functionality for the base classes, e.g. JavaBeans accessors or data access functionality. Because business logic and data access logic is combined in the entity classes, they can be said to implement the Active Record design pattern [25]. Data constraints for domain objects are enforced by JSR 303 (Bean Validation) annotations [10]. An example is given in Listing 1; attribute title of class Book is required to have at least a length of 1.
4. TDD WITH SPRING ROO

In the following sections, steps towards a test-driven usage of Roo are explained.

4.1 Process-Oriented Analysis

We have introduced three techniques: ATDD, TDD and MDSD. We now analyze the potential for combining them. Both TDD and MDSD are techniques for implementing new functionality. The order of the steps taken out to implement code are almost inverted (Figure 4). In TDD the first step is writing the test, followed by implementing the desired functionality and eventually cleaning up both the test code and the production code by applying refactorings. In MDSD, the first step is designing an abstract model and generators, which produce the executable production code. This step implies the design of the resulting system because it is incorporated in the generators. Thereafter, the domain is modeled and the corresponding system is generated. The last step might be generating tests for the production code to build a test harness that protects against unintended effects when changing the production system.

Obviously, these two approaches are mutually exclusive because their processes are too different. But to guarantee the quality of the code generator one can use TDD for their implementation. While TDD and MDSD cannot be used together, there is a natural fit of ATDD and MDSD. Both are used in different development phases. One can record the customer’s requirements using automated acceptance tests. Subsequently, these tests drive the implementation of the desired functionality.

4.2 Integrating TDD and MDSD

Our analysis yields potential for integrating ATDD, TDD and MDSD. Preliminarily, we denote our test- and model-driven development approach with the term TMDD. The outward, incremental cycle is determined by ATDD. Automated acceptance tests capture new requirements for the current iteration. Depending on the complexity of the new functionality, you can choose between three methods for implementing the new feature (Figure 5).

Less complex features for which a generator already exists (or a generator is worth investing resources to build it) can be implemented cheaply in a model-driven manner. This approach is very cost-effective and with proven generators the quality of the generated code will be adequately high.

More complex features are seldom worth building a generator. For example, a very complex search functionality has exactly one implementation. There is no cost advantage of model-driven “mass production”. Rather, it is more efficient to build this component directly. To ensure good internal quality and thereby low maintenance costs, a test-driven approach should be chosen. Obviously, unique components require additional resources but they provide extraordinary business value and competitive advantage.

Finally, there are features in between the two extreme cases. The desired functionality is similar to other generated components but there are subtle differences in the specification. This is a valid case for combining MDSD and TDD. First of all, the supportive code is generated. This builds a stable basis for the new component. Generated tests might now come in handy as a harness against unwanted changes. Afterwards, the component is changed through new (unit) tests. During the change process, some of the generated tests might break. It is up to the developer to decide whether this failure is unwanted and the former change must be reverted or if the specification of the unit differs from the generated component and the failing test can be deleted safely.

4.3 Supportive Tools and Techniques

Test- and model-driven development approach (TMDD) can be supported with tools and techniques.

4.3.1 A Java Acceptance Test Library

Currently, there is a lack of support for writing automated acceptance tests in a Java-centered Web project. There are well established libraries for automated execution of user interactions inside the browser, e.g., Selenium WebDriver [1]. However, they are technologically-oriented: The respective DOM elements are located by CSS or XPath expressions, which are hard to read, especially for non-technical customers. Hence, we developed a dedicated library that aids in abstracting from technical details towards a business-centered form of acceptance tests. Our library integrates with JUnit 4, the de-facto standard in Java test frameworks.

The first benefit is the decoupling of logical UI elements and technical DOM element IDs. The main entry point to our library is the AbstractAcceptanceTestGroup class. It provides several methods, which compute DOM elements from their logical caption. For instance, the method textfieldWithLabel(String label) takes the label of a text...
field as input parameter, looks for the corresponding `<label>` tag on the current Web page, extracts the ID of the text field from the `for` attribute of the tag and executes an XPath expression on the DOM to obtain the `<input type="text">` element. This makes the tests robust against changes in the computation of element IDs. Some Web frameworks—e.g. Java Server Faces (JSF)—compute element IDs automatically based on the tree structure of the JSF components and the IDs of the parent elements. Both the structure and the element IDs are subject to frequent change that would break the acceptance tests without reason—the underlying functionality does not usually change.

The second benefit is improved readability and understandability of the acceptance tests. This reduces problems arising from communication problems between customer and developer. It is our goal to enable automated acceptance tests that are as comprehensible as manual tests. When the customer is able to read the tests, this might allow skipping “prosaic” acceptance test thus reducing development costs. We foster the readability of acceptance test code by using `intent-revealing names` [9, 37]. We deal conciseness for readability. The purpose of the method parameters is described by the method name. To give an example: the methods `buttonWithCaption("OK")` and `buttonWithAltText("OK")` reveal the different purpose of the `String` argument.

To work around distracting and irritating keywords of Java, we use `factory methods` [27], which hide syntactical complexity from the reader. Instead of e.g. creating a new instance of `AbstractTable` directly, one uses the `tableWithColumns(String... columnNames)` factory method.

### 4.3.2 Custom Roo Add-On

To facilitate the use of our library in Roo projects, we developed a small add-on. Executing the command `acceptancetest --story [--storyGroup]` on the Roo shell creates a new test class that inherits from our `AbstractAcceptanceTestGroup` class. The `--storyGroup` option eases the logical grouping of stories. Stories that belong together get a common superclass forming a group of tests.

Issuing the `perform acceptancetests` command on the Roo shell applies the necessary changes to the Maven build process and directly executes all acceptance tests.

#### 4.3.3 Building a Maintainable Acceptance Test Suite

Although our library significantly raises the level of abstraction, it is still bound to the `solution domain` (i.e. Web applications). By systematic refactoring and smart code design, one can raise the level of abstraction even further and approach the language of the `problem domain`. The goal is to formulate acceptance tests in terms of the `domain language`. We show how to apply well-known techniques on acceptance test code. These techniques require programming skills, thus address developers. Domain experts will not be able to use them without extensive training.

If you want to evolve a comprehensive specification from your acceptance test suite, appropriate arrangement and meta data become crucial. To improve the searchability and maintainability, descriptive title for features, user stories, and test cases should be used [3]. A short summary at the beginning of each feature class helps readers understanding the purpose. To ease orientation within an acceptance test, you can leverage approaches from BDD, which proposes a standard form for tests: `Given—When—Then`. `Given` expresses the preconditions that must be fulfilled to be able to execute the test. There might be multiple given-expressions. They should address domain concepts; technical premises should be dealt with in other parts of the automation code. The `When` part describes the action that changes the system state. There is only one such step. Otherwise, the test context changes multiple times; that makes it hard to analyze, why a test failed [3]. The `Then` steps define the postconditions that have to be fulfilled. They tell the tester how to check whether the specified feature works correctly.

In order to promote sustainability of the test suite, all tests should be executable in any order at any time. Test `interactions` through side effects [38] can be hard to detect. If the test suite becomes unstable, confidence in the tests declines and benefits will probably vanish. The simplest mean is setting up the application state from the baseline. In JUnit, methods annotated with `org.junit.Before` can clean up the database. This way every test asserts that the state is consistent and deterministic for every execution. Employing `@Before` instead of `@After` has another advantage: After the tests ran, the state of the application can be inspected.

To improve readability of test code, it should be refactored constantly. With careless development, much data will be duplicated. Think of a base URL of an application. It complicates testing the application in a different environment. Besides, a URL as it is usual in a development environment (e.g. `http://localhost:8080/`) is not easy to understand for business users. The solution is at hand: extracting a constant [24] for the base URL such as `HOMEPAGE`.

Some user actions are the same in multiple tests. To give an example: users have to log in for session-based applications. Instead of repeating the steps for login, these concepts should be abstracted by extracting methods [24] capturing the whole login procedure. With a growing number of acceptance tests, high-level concepts might be scattered among test classes. It is useful to create a common super class and pull up general helper methods (Figure 6).

A major hindrance to readability are `incidental details` [3]. This is information that is not relevant to the specified business rule but still part of the acceptance test. Moving this information to other parts of the code is crucial. Methods can be extracted and parameters replaced by constants.

---

**Figure 5:** Integrated process model for TDD with Spring Roo
Our approach is demonstrated using the example of a simple (book) library management. It has a reasonable complexity e.g. complicated validation rules. We create a new Roo project by executing project --topLevelPackage www.library --projectName Book-Library. The initial requirement was captured as a user story: As a librarian I want to enter the meta data of a book to make it borrowable.

This story is illustrated with acceptance tests. Starting with the happy path we create a new test with the command acceptancetest --story "Happy-Path" --storyGroup EnterBookMetadata. Listing 2 shows the result.

```
public class EnterBookMetadataStories {
    @Test public void happyPath() {
        open(HOMEPAGE);
        clickClickableAndWaitForNextPage(
            linkWithCaption("New book record"));
        typeTextIntoTextfield("Design Patterns",
            textfieldWithLabel("Title"));
        typeTextIntoTextfield("Gamma et al.",
            textfieldWithLabel("Author"));
        buttonWithCaption("Save")
            assertTextIsPresent("Design Patterns"),
            assertTextIsPresent("Gamma et al.");
    }
}
```

Listing 2: Library code example

Executing the test with perform acceptancetests fails. So far there is no production code offering the intended Web application. It is relatively easy to pass this test because it merely deals with simple CRUD functionality. We set up the persistence layer and create a book entity: entity --class ~.domain.Book. Based on this entity we scaffold the book management functionality: controller scaffold --class ~.web.BookController --entity ~.domain.Book. Despite the page for recording a book being implemented, the test still fails; the domain object does not have the required fields. We add the title attribute with field string --fieldName title --class ~.domain.Book. Likewise we add the fields author, publisher, year, and isbn. Afterwards, the first acceptance test passes. The functionality was implemented by using model-driven means.

Refactoring follows. A failing test during this activity signals that we broke the test logic. We extract descriptive constants from the test data, and extract helper methods. This yields the code shown in Listing 3.

```
public class EnterBookMetadataStories {
    String VALID_TITLE = "Design Patterns";
    String VALID_AUTHOR = "Gamma et al.";
    [...] public class EnterBookMetadataStories {
    String VALID_TITLE = "Design Patterns";
    String VALID_AUTHOR = "Gamma et al.";
    [...] @Test public void happyPath() {
        whenIAddNewBookMetadata(
            VALID_TITLE, VALID_AUTHOR, [...] );
        thenIShouldSeeTheRecordedBookData(
            VALID_TITLE, VALID_AUTHOR, [...] );
    }

    private void whenIAddNewBookMetadata(
        String title, String author, [...] ) {
        open(HOMEPAGE);
        clickClickableAndWaitForNextPage(
            linkWithCaption("New book record"));
        typeTextIntoTextfield(title,
            textfieldWithLabel("Title"));
        typeTextIntoTextfield(author,
            textfieldWithLabel("Author"));
        buttonWithCaption("Save")
            assertTextIsPresent("Design Patterns"),
            assertTextIsPresent("Gamma et al.");
    }
}
```

Listing 3: Refactored test code

The next step is to specify requirements for the input data. We execute acceptancetest --story "Invalid ISBN number" --storyGroup EnterBookMetadata on the Roo shelf to create a new test method. We adopt the first acceptance test, but expect to see an error message (Listing 4).

```
@Test public void invalidISBNNumber() {
    whenIAddNewBookMetadata(
        VALID_TITLE, [...] , TOO_SHORT_ISBN);
    assertTextIsPresent("The ISBN contains less than 13 digits!");
    [...] public class EnterBookMetadataStories {
    String TOO_SHORT_ISBN = "978-020163361";
    [...] }
```
ConstraintValidatorContext context =
IsbnValidator isbnValidator =
   new IsbnValidator();

@Before public void initMocks() {
   cnstrVltBldr =
      mock(ConstraintViolationBuilder.class);
   context =
      mock(ConstraintValidatorContext.class);
   when(context
      .buildConstraintViolationWithTemplate(
         (String) any()));
   thenReturn(cnstrVltBldr); }

@Test public void acceptsValidIsbn() {
   boolean actual =
      isbnValidator.isValid(VALID_ISBN, context);
   assertTrue(actual);
   verify(context)
      .buildConstraintViolationWithTemplate(
         isbnValidator.LESS_THAN_13_DIGITS);
   verify(cnstrVltBldr)
      .addConstraintViolation(); }

@Test public void rejectsShortIsbn() {
   boolean actual =
     isbnValidator.isValid(VALID_ISBN, context);
   assertFalse(actual);
   verify(context)
      .buildConstraintViolationWithTemplate(
         isbnValidator.LESS_THAN_13_DIGITS);
   verify(cnstrVltBldr)
      .addConstraintViolation(); }

Listing 5: Test to reject too short ISBNs

class IsbnValidator
   implements ConstraintValidator<Isbn, String> {
   String LESS_THAN_13_DIGITS =
      "The ISBN contains less than 13 digits!";

   public void initialize(Isbn annotation) {
   }

   public boolean isValid(String value,
      ConstraintValidatorContext context) {
      if (countDigits(value) < 13) {
         context
            .buildConstraintViolationWithTemplate(
               LESS_THAN_13_DIGITS)
            .addConstraintViolation();
      return false;
   return true;
   }

   private int countDigits(String text) {
      return text.replaceAll("[^0-9]", "").length();
   }

Listing 6: Implementation of IsbnValidator

Likewise, we could drive checks for more than 13 digits,
for null values, and for advanced, algorithmic ISBN con-
straints. Finally, we write an annotation (Listing 7) and
stick it to the isbn attribute in class Book. After all this
low-level work, our acceptance test passes.

@Target({ElementType.METHOD, ElementType.FIELD,
   ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy=IsbnValidator.class)
@Documented
public @interface Isbn {
   String message() default "Enter a valid ISBN!";
   Class<?>[] groups() default {};
   Class<? extends Payload>[] payload() default {};
}

Listing 7: The Isbn annotation

The example illustrates how ATDD, MDSD with Spring
Roo and TDD can be combined. Automated acceptance
tests form an unambiguous specification for implementing
new functionality. Simultaneously, the growing set of accep-
tance tests prevents regressions. Our Roo add-on facilitates
the adoption and use of automated acceptance testing. How-
ever, since the latter are still plain JUnit tests, syntax and
semantics come natural to Java developers. The investment
and resistance for adoption will be low, and the team can
concentrate on adjusting their development process. Tests
can be run without Roo. There is no vendor lock-in.

6. CONCLUSION AND FUTURE WORK

We presented work on the combination of model-driven
and test-driven development of Web applications. It in-
cluded evaluating Spring Roo as a development framework.
We sketched related work; while many papers exist on the
distinctive approaches, only few combine them. After ex-
plaining basics, we explained our development process. We
also highlighted the connection to acceptance testing. The
feasibility of our work was shown in an exemplary scenario.
To conclude, using Roo for code generation and incorpo-
rating it into a MDSD process driven by acceptance testing
is advisable. The combination with TDD is not only possi-
bility but a viable option. MDSD increases implementation
efficiency. This is aided by the fact that learning to work
with Spring Roo is straightforward since merely knowledge
in Java is required. Using ATDD for regression can increase
overall (external) quality. Finally, TDD increased the inter-
code quality. Therefore, a combination of the approches
in the improves both effectivity and efficiency.

Our work is not finished. Findings should be evaluated
in industrial software development projects. Ideally, they
should be verified by a quantitative study. A future target
could be to describe best practices for combining various
development paradigms to provide better development pro-
cesses. It would be valuable to know whether our findings
can be transferred to other forms of development. Targets
of future research could be other frameworks besides Spring
Roo as well as other platforms. For example, developing
mobile applications (apps) could be a target of future re-
search. Furthermore, we will consider the inclusion of script-
ing languages to e.g. increase readability of our tests. Ide-
ally, measuring the success of our approach e.g. regarding
cost-effectiveness should become possible.

7. REFERENCES

shifting businesses to the cloud. In Proc. ICWET ’11,
New York, NY, USA, 2011. ACM.
http://blog.springsource.com/2009/12/31/spring-roo-
1-0-0-released/.
[5] B. Alex, S. Schmidt, A. Stewart, J. Tyrrell, and
A. Swan. Spring Roo - Reference Documentation
1.2.0.SNAPSHOT. SpringSource, Inc., 2011.