Fall 2012

Software Requirements As Executable Code

Karen Eileen Wasielewski Morand
Regis University

SOFTWARE REQUIREMENTS AS EXECUTABLE CODE

A THESIS

SUBMITTED ON 31st OF DECEMBER, 2012

TO THE DEPARTMENT OF COMPUTER SCIENCE

OF THE SCHOOL OF COMPUTER & INFORMATION SCIENCES

OF REGIS UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF MASTER OF SCIENCE IN

SOFTWARE ENGINEERING AND DATABASE TECHNOLOGIES

BY

[illegible]

Karen Eileen Wasielewski Morand

APPROVALS

[illegible]

William Masters, Thesis Advisor

Donald J. Ina

Don Ina, Faculty of Record

[illegible]

Nancy Birkenheuer, Ranked Faculty

Revision Date 12/31/2012

Abstract

This project analyzed the effectiveness of using Story Testing frameworks to create an application directly from user specifications. It did this by taking an example business application with “traditional” specifications, and rewriting those specifications in three different Story Testing Frameworks – Cucumber, FitNesse, and JBehave. Analysis of results drew the following conclusions: 1) Story Testing can help prove a project’s completeness, 2) Specifications are still too technical, 3) Implementation is not overly complex, and 4) Story Testing is worth it. It proposed future research around evaluating natural languages and seeking more user-friendly ways of writing specifications in a natural language.

Acknowledgements

Thanks first and foremost to my husband, for supporting me in a thousand different ways.

Thanks to my advisor and other Regis faculty, for guiding me through it all.

Thanks to my employer, for funding my education.

Finally thanks to my parents, for teaching me not to quit.

Table of Contents

List of Figures

List of Tables

Chapter 1 – Introduction

To stay competitive, businesses must be able to rapidly evolve in response to changing markets. To support this evolution, software projects must be able to respond to evolving business requirements – which may very well change in the middle of development. Traditional engineering-based approaches to software development have frequently failed for this reason.

In a traditional engineering project, all of the design and decisions are made before construction begins. This is reasonable when building a skyscraper, but if a software project finalizes all of its design and decisions before any construction begins, then the cost of any change during that construction can be significant. If the business changes a requirement during development, it could mean that the whole project stalls while the initial designs are re-worked. Often the new design means that some of the existing development work must be re-done also, creating cost and wasted effort.

Software Projects Often Fail

Software projects often fail to achieve their goals within the targeted timeframe and budget. According to the Standish CHAOS report, in 1994 only 16% of software projects were successful. Since then, software developers have tried to find ways to enhance their approaches and methodologies. More recent software development innovations and methodologies have helped improve responsiveness to changing requirements, but software project failures persist, costing businesses billions of dollars. A more recent CHAOS report, from in 2009, shows that we are still at only a 32% success rate (Eveleens & Verhoef, 2010). Furthermore, the 1994 Standish report estimated that "…in 1995 American companies and government agencies will spend $81 billion for cancelled software projects. These same organizations will pay an additional $59 billion for software projects that will be completed, but will exceed their original time estimates" (Standish Group, 1995).

What is going wrong? One issue is that there still remains a significant disconnect between the business requirements and the final product. How do we make sure a product matches its requirements? When requirements change, how can we ensure these changes make their way into the final product? Responsiveness to changing requirements will not help if the requirements are not implemented correctly when they change.

Duplicated Software Requirements

Let us consider the classic way a software development project works, as depicted in Figure 1. It starts when a business person has an idea. They describe that idea to an analyst, who then writes up a comprehensive Requirements Document. The requirements document is given to a project team. Next, the programmers read through the requirements, design the software, and implement the code. At the same time, the QA testers read through the requirements and write test cases around them. When the coding is completed, and all of the tests pass, the team declares victory and the application is deployed to production. All too often, however, the application does not match what the business was looking for.

What has gone wrong? First the analyst writes the *requirements*. The developer then creates a *design* from those requirements, and then implements the design as *code*. Finally, the QA Tester writes the requirements in a *test plan*. We now have *four copies* of the requirements. How likely is it that they all match what the business wanted? Furthermore, if the business introduces any change to the original requirements, how likely is it that we will get that change into all four copies?

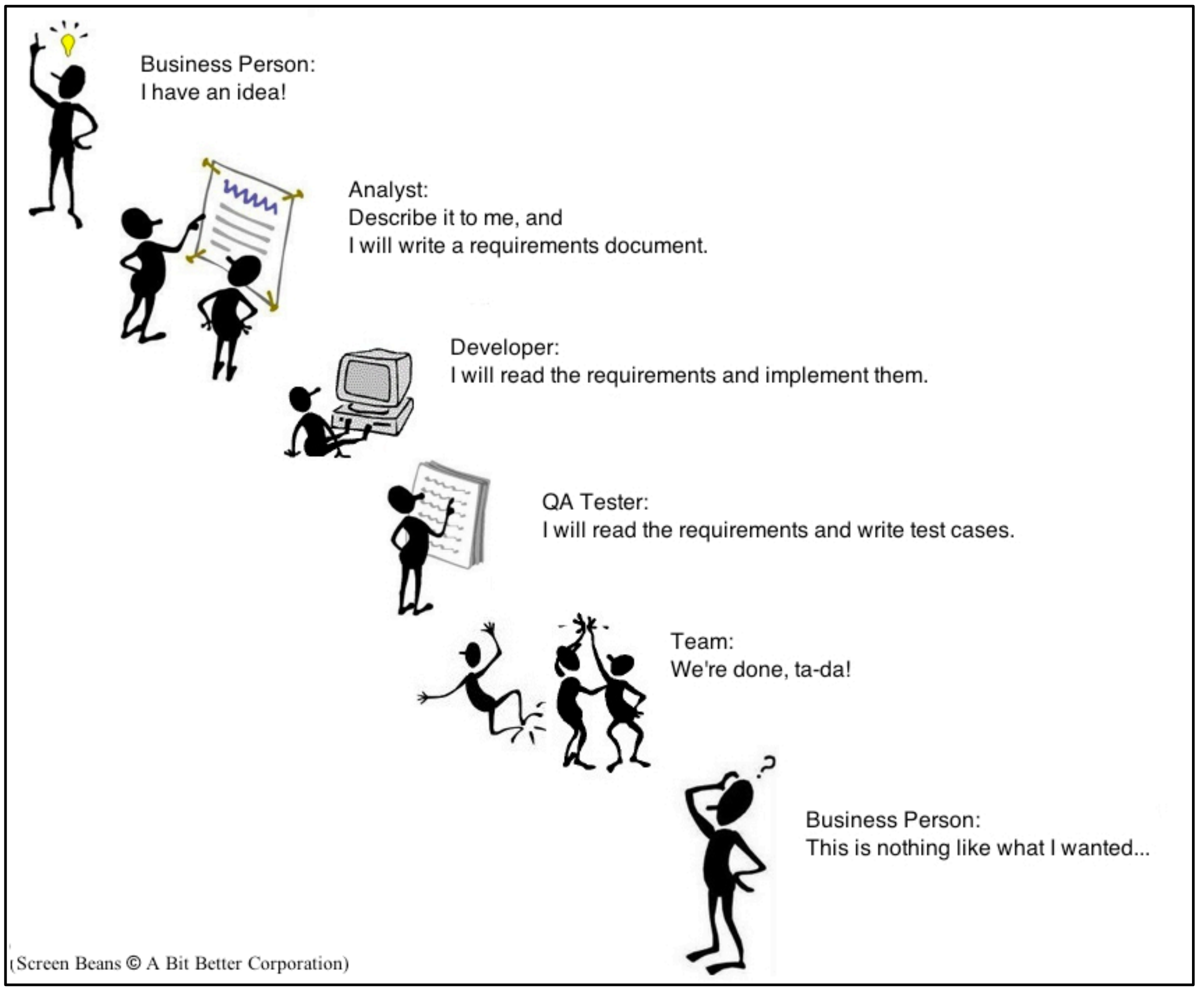

Business Person:
I have an idea!
Analyst:
Describe it to me, and
I will write a requirements document.
Developer:
I will read the requirements and implement them.
QA Tester:
I will read the requirements and write test cases.
Team:
We're done, ta-da!
Business Person:
This is nothing like what I wanted...
(Screen Beans © A Bit Better Corporation)

Figure 1: Does this look familiar?

Business requirements are generally written in unstructured documents, and the software is written in code. This means that there is no way to verify that the application meets the requirements except with rigorous manual testing. Testing can be automated, but this means coding the requirements twice - once for the product, and again for the test suite. Even then, there is no guarantee that the test suite itself matches the requirements. It is subject to human error and misinterpretation.

Could a Story Testing Framework Provide a Solution?

If duplication of requirements is one of the primary sources of failure, then it seems obvious that eliminating duplication will help raise our chances of a successful software project. Alternatively, guaranteeing that duplicate artifacts contain the same content will also raise our chances of success.

A practice sometimes called Story Testing is attempting to do just this. Building on practices begun with the Agile movement's Test Driven Development (Beck, 2003), recently practitioners have started building automated testing frameworks for the purpose of translating software requirements directly into executable code. The executable code can then be run against the software itself, to see if it meets the requirements. This effort, sometimes called Behavior Driven Development (North, 2006) or Acceptance Test Driven Development (Hendrickson, 2008), tries to put software requirements into a framework tightly coupled to the final product. These executable requirements, called Story Tests, are written in a natural language, and are meant to be readable and understandable by the non-technical business people. The argued benefit is that the language you use affects the way you think (Martin, 2008).

The goal of Story Testing is: "…to express functional expectations such that Developers, Testers, and Analysts all can read and understand and agree" (Dinwiddle, 2010). It would be practical to add "customers" to that list as well. Furthermore, Story Tests can define "done" for a particular story, and they can help a team to prevent defects up front, instead of finding than find them later.

Ideally, Story Tests would take the place of unstructured requirements altogether, eliminating redundancy. They should provide a very clear definition of when the requirement is complete. When all of the requirements are satisfied, all tests will pass. If a requirement changes, it is documented by a failing test that indicates that the software does not yet meet the new requirement. Thus, the story tests provide a guarantee that the completed application matches the requirements.

Although several Story Testing frameworks exist, and they have all been used in some form or another, none of them have been widely adopted. They have failed to replace traditional software requirements, despite their potential for reducing the chance of project failure. There are many reasons that might explain this lack of adoption, both from business and technical perspectives. It could be that Story Testing requires too much business involvement. It could be that the writing the requirements is too complicated. It could be that implementing the technical framework is too much effort. It could be something else entirely, or a combination of several factors such as cost, installation, integration, maintenance, and training barriers. Focusing largely on the requirements side of the picture, this research project will explore the usability of Story Testing, and consider why it is not more widely adopted.

Project Statement

Story Testing has the potential to bridge the gap between business requirements and completed software. There have been many efforts to do this via a variety of frameworks, but none have been widely adopted. This project will explore current Story Testing frameworks for creating executable requirements and analyze their effectiveness in a real-world situation.

Significance of the Project

As the CHAOS reports have shown, the cost of a missed or misunderstood requirement can be very high (Standish Group, 1995). Furthermore, the later in the development cycle it is discovered, the higher that cost will be (Ambler, 2003). The traditional "waterfall" model of software involves extensive documentation of requirements, to make certain nothing is missed. However, that documentation still needs to be interpreted and then rewritten multiple times in the form of code and tests, by analysts, developers, and testers. Duplication has a high danger of error. If a requirement is misinterpreted in any step along the way, it can be missed. If a requirement changes, that change has to occur in all the documents, code, and tests that described the original requirement. If one or more location is missed, the change might not make it correctly into software. There is no way of guaranteeing that the final product matches what the requirements stated – let alone what the business really wanted.

Newer "agile" methodologies advocate getting rid of documentation or greatly limiting it, favoring communication over documentation (Beck, Beedle, Van Bennekum, & Cockburn, 2001). This eliminates duplicated documentation – however when communication is not recorded, faulty memories or misinterpretation can result in missed requirements as well. Once again, there is no way of guaranteeing that the final product matches what the requirements stated – let alone what the business really wanted.

Turning business requirements into executable code – Story Tests that can unequivocally pass or fail, and are a part of the code base itself – is a way of guaranteeing that the code will always match the requirements. Doing so will eliminate redundancy and rewriting of specifications, but it can only succeed with buy-in from everyone working on a project – from the business through development and testing. There have been many efforts to do this via a variety of frameworks, but none have been widely adopted. This project will investigate three frameworks, with the goal of researching their effectiveness and appeal.

Project Scope

Using a Design Science approach to research, this project will implement three Story Testing frameworks around the exact same set of business requirements. By implementing and comparing frameworks for the same application, this project will compare and contrast the three different Story Testing frameworks. While the project will mention the technical aspects to implementing the frameworks, it will focus largely on the challenges of writing the requirements themselves, and working with the results.

It will also evaluate each framework based its usability and effectiveness. In doing so, it will point out both advantages and flaws in the current approach to story testing, suggest items for further study, and suggest steps for improvement.

Chapter 2 – Review of Literature and Research

In reviewing the literature around software requirements and Story Testing, this chapter will begin by briefly touching on the history of Software Development processes. As a few main processes are considered, it will note how business requirements are delivered to developers in each process, what they look like, and ways in which they fail to ensure that the system built matches what the business required. The review will also talk about each process's ability to respond to rapidly changing business requirements.

After this short discussion, the review will concentrate on one specific software development methodology: Agile Development. After a brief overview of the methodology, the review will delve more deeply into some Agile techniques relevant to dealing with business requirements. It will start by outlining Test Driven Development, and then address Story Testing as a logical extension of TDD.

Story Testing, as the primary subject of this research paper, requires particular attention. The review will first explore its origins, and then it will discuss many of the current Story Testing frameworks. It will explore where they came from, why there are so many different ones, and what problem each was trying to solve. The review will then discuss the kinds of languages employed by Story Testing frameworks, what they look like, and how they work.

Once the review is complete, this chapter will bring all this history together by describing a step-by-step process of how a single Story Testing framework might be used as part of the development of a new software project. To conclude, the chapter will summarize the specific Story Testing frameworks that will be implemented by this research project in Chapter Three.

Then it will discuss the criteria by which they will be evaluated, in preparation for Chapter Four, the results analysis.

In the beginning…

Software design did not begin as a formalized process. In what Robert Glass calls "The Pioneering Era," from 1955 to 1965, computers were large and complex, and software development was equally complicated. Code was written on punch cards, and engineers could only run their jobs by signing up in advance for machine time. Project scheduling was not even considered. "The field was so new that the idea of management by schedule was non-existent. Making predictions of a project's completion date was almost impossible" (Glass, 1998). Deadlines were not missed, because they did not exist. However without any schedules, the companies relying on the software could not make business plans or budget predictions.

Waterfall Software Development

As the field began stabilizing in the 1970's, and application complexity grew, it became clear that programmers could no longer simply hold the complete abstract of a system in their minds and transfer it into code (De Wille & Vede 2008). Computer programmers began to apply classic engineering practices to software. Software started to be broken down into phases of development, often represented as something like the following: Requirement Definition, System Design, Software Design, Coding, Integration and Verification, then Operation and Maintenance.

These phases are often referred to as "Waterfall Development" (Royce, 1970). Each phase needs to be completed before the next one starts, like water cascading down a series of rocks. Applying standard process to the development of software strengthened the field significantly, bringing order to chaos. It stopped programmers from writing code before they even knew what they were building. It encouraged a logical progression of steps: the business requirements were fully fleshed out; the software was designed based on those requirements; finally, the application would go through a testing phase before it went into production, verifying that the software actually met the requirements. Following these steps in order encouraged programmers to consider their designs before diving into the code, and enforced pre-production testing.

Although the stages of Waterfall helped to create higher quality software, this process did not solve everything. A large challenge remained, and still remains to this day: software requirements frequently change before development is complete. Change wreaks havoc on the waterfall approach. A requirements change that occurs during the "Coding" phase, for example, likely means going back and reworking the Software Design document, and possibly even the System Design document. Once the Systems Design has been reworked, the Software Design must change, then some or all of the code needs to be changed to match the new design… and so on.

At first, software engineers would try to avoid this problem the same way physical engineers did, by forcing the business to sign contracts stating that the requirements fully detailed the expected product. While restrictions like these make sense in physical engineering or computer hardware, where physical space, partially-built structures, and pre-purchased materials create real limitations with regard to change – the fact is that building software is not really like building a skyscraper.

In order to stay competitive in today's dynamic market, businesses have to be able to evolve rapidly. If a company is making software to be used by Widget A, but suddenly massively popular Widget B comes on the market, the software had better support Widget B too, or its customers will go elsewhere. It seemed to everyone, particularly non-technical managers, that these restrictions should not be as necessary in software. Computer hardware was getting cheaper and faster every year, so why couldn't software? However, efforts to introduce change part-way through resulted in a "software crisis" (Randell, 1968), a nightmare of missed schedules, blown budgets, and flawed products (Brooks, 1987). Although it works very well for developing applications where the requirements are stable from the start of the project to its finish, the Waterfall approach to software development is simply not capable of handling changing requirements.

Iterative Software Development

Despite these initial failed efforts, businesses continued to demand the ability to rapidly change in response to changing markets. Any business whose software *can* respond to change will have a competitive edge in the market, since so few can. It seemed that we had to find a way for software development to respond more nimbly to changing requirements. Unified Process, JAD/RAD, and Agile are a few of the development methodologies that tried to address the problems encountered in Waterfall development. Since change is inevitable, they try to find ways to embrace that change rather than resist it.

Unified Software Development Process

The Unified Process (UP) is a generic version of the *Rational Unified Process* trademarked by IBM. It is a software engineering process with three basic axioms: (1) requirements and risk-driven, (2) architecture-centric, (3) iterative and incremental (Arlow, & Neustadt, 2005). "Risk-driven" means that UP addresses the most critical risks close to the beginning on the project. "Architecture-centric" means that the software's architecture is a primary consideration throughout the project. Finally, "iterative and incremental" means that UP tries to make the development process more flexible by introducing the concept of iterative development.

UP is divided into four phases -- Inception, Elaboration, Construction, and Transition. Each phase consists of one or more iterations, and iterations are broken up into five parts -- requirements, analysis, design, implementation, and test. The idea behind this iterative and incremental development process is to be highly responsive to change. By making sure that there are many iterations of the requirement/analysis/design/implementation/test model, it means that the software's requirements and design will be regularly re-evaluated. It also aims to find bugs and requirement changes as early as possible in the process. The sooner a necessary change is detected, the less expensive it will be.

UP heavily uses UML in its design stages. UML, or the Unified Modeling Language, is a means of visually modeling a software system. The Object Management Group accepted UML as a standard Object Oriented modeling notation in 1997 (Weisfeld, 2004). UML consists of tools, rules, and diagrams that can visually represent buildings blocks, common mechanisms, and architecture of a system. Its power comes from the fact that it is not tied to any particular language or methodology.

One artifact produced by UML is a *Use Case*. A Use Case is "…a transition or sequence of related operations that the system performs in response to a user request or event" (Weisfeld). A Use Case is one type of requirements document that might come from the business, or at least be looked at by the business. Here is a Use Case template from Alistair Cockburn (1998).

Use Case: <number> <the name should be the goal as a short active verb phrase>
Goal in Context: <a longer statement of the goal, if needed>

Scope: <what system is being considered black-box under design>
Level: <one of: Summary, Primary task, Subfunction>
Primary Actor: <a role name for the primary actor, or description>
Priority: <how critical to your system / organization>
Frequency: <how often it is expected to happen>

A downside of UML is that it is very documentation-heavy. Even with UP's iterative approach to the development lifecycle, a requirements change in the middle of a project can force many UML document changes. This might generate a good deal of re-work. Furthermore, it is difficult to guarantee that the change makes it into both the UML documents and the software itself. Although there are tools that help with this, keeping the code in sync with the design is a constant challenge, and it reduces the Unified Process's ability to rapidly respond to changing requirements.

JAD and RAD

JAD and RAD are two different methodologies, but in combination they are a powerful way of gathering requirements for development. IBM originally introduced Joint Application Design, or JAD, in the late 1970's (Davis & Yen, 1999). The idea behind JAD is to organize a team of users, sponsors, analysts, designers and developers together in one physical location, to flush out the business requirements and system design. "The facilitated JAD workshop brings key users (stakeholders) and systems professionals together to resolve their differences in a neutral, non-hostile atmosphere" (Jennerich, 1990). The documents produced by a JAD session can be in any format; the JAD process does not specify what the requirements should look like.

JAD is a way of ensuring that the software requirements match the needs of all the stakeholders and are fully understood by the developers as well. It also creates the requirements documents more quickly than if they were passed from team-to-team, like is done in Waterfall or UP. However it is not an iterative process; that is where RAD comes in.

James Martin developed Rapid Application Development, or RAD, in the late 1980's. RAD uses iterative, evolutionary prototyping (Maner, 1990). During a JAD workshop, RAD can be used to iteratively develop exploratory prototypes that enable the JAD team to evolve their requirements design. A prototype will be rapidly built to answer a specific question about design. It is much easier to ask the business "is this what you want?" with a working prototype, than with descriptive text. "The idea behind prototyping or RAD is to expose part of the solution to the end user as early in the development cycle as possible so that critical feedback can be given and reacted to" (Hubbard, n.d.).

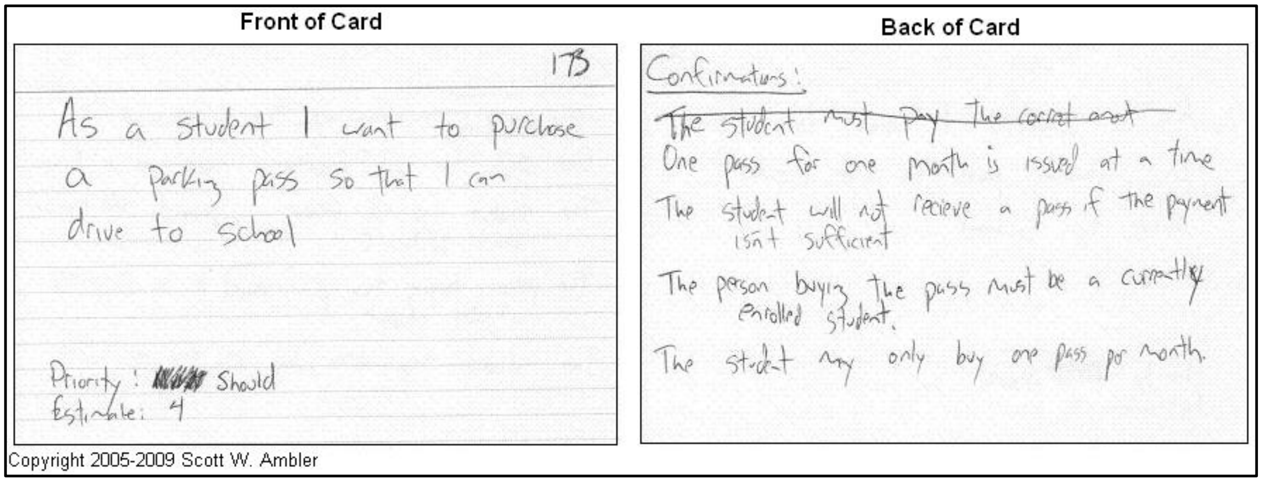
JAD and RAD, when working together, have the ability to generate a set of requirements and working prototypes that meet the customer requirements very well. Their objective is to reduce development time. The less time there is between requirements gathering and the final product, the less likely there are to be significant changes to those requirements. However these processes do not specifically address changing requirements, nor do they provide a way to ensure that the requirements are met.

Agile Development Methodologies

Agile, just like the Unified Process, is an attempt to reduce waste by being able to respond rapidly to changing requirements. Agile development attempts to take things a little farther by being even *more* responsive to change. The goal is to discard "phases" altogether, and do all of these activities in a much tighter cycle - completing all phases in a single iteration, which might be anywhere from a week to a month's worth of time. Additionally, Agile tries to reduce the amount of documentation needed for a project, favoring "Working software over comprehensive documentation" (Beck et al, 2001).

Where UML has Use Cases to deliver requirements, Agile methodologies often use *User Stories*. According to Scott Ambler, "A user story is a very high-level definition of a requirement, containing just enough information so that the developers can produce a reasonable estimate of the effort to implement it" (Ambler, n.d.).

User Stories range from very formal all the way down to informal. The goal of a User Story is to keep it as simple as possible, but no simpler. A user story is generally thought to be the beginning of a conversation, as opposed to a comprehensive requirements document. Practitioners will often write the story on a 3x5 index card, forcing themselves to remain succinct. Here is an example of a story card (Ambler, n.d.).



Front of Card

173

As a student I want to purchase a parking pass so that I can drive to school

Priority: [illegible] Should
Estimate: 4

Back of Card

Confirmations:

~~The student must pay the correct amount~~

One pass for one month is issued at a time

The student will not recieve a pass if the payment isn't sufficient

The person buying the pass must be a currently enrolled student.

The student may only buy one pass per month.

Copyright 2005-2009 Scott W. Ambler

Figure 2: Example Story Card

Agile development strives to be as lightweight as possible. The power in informal requirements like the one show above, and in minimal documentation, is that very little time is spent documenting requirements that may be about to change anyway. When the above User Story is developed, the development team will talk to the business and build precisely what is wanted at the time of development – regardless of what was wanted when the requirement was first conceived. This means a greater responsiveness to changing requirements, however it also means that very few decisions are documented.

A lack of documentation can be dangerous, because then the team might find itself relying solely upon faulty human memories and misinterpretation. Also there is no way to capture and verify changing requirements on a 3x5 index card. Agile techniques try to surmount this by documenting decisions and designs in the source code itself, via executable tests and automation. The rest of this literature review will discuss Agile techniques which are trying to do just that.

Test Driven Development

One of the heavily emphasized Agile development techniques is Test Driven Development, or TDD. Test Driven Development "…is an advanced technique of using automated unit tests to drive the design of software and force decoupling of dependencies" (Palermo, 2006). With TDD, the developer begins development by writing a failing test (in code), and then write just enough code to make it pass. The developer then repeats this process. She writes a new failing test, and then she writes just enough code to make it pass. She repeats these steps regularly, along with regularly removing duplicate code and improving design, until there are no more tests to write.

TDD is practiced in many programming languages. Some examples of popular TDD frameworks would be: JUnit (for Java), NUnit (for .NET), CppUnit (for C++), and Jasmine (for JavaScript). Most TDD frameworks use a "red bar" to indicate a failing test, and a "green bar" to indicate a passing one. Thus the TDD mantra is "Red, Green, Refactor" (Beck, 2003). Figure 3 shows a JUnit screenshot with the green bar displayed.

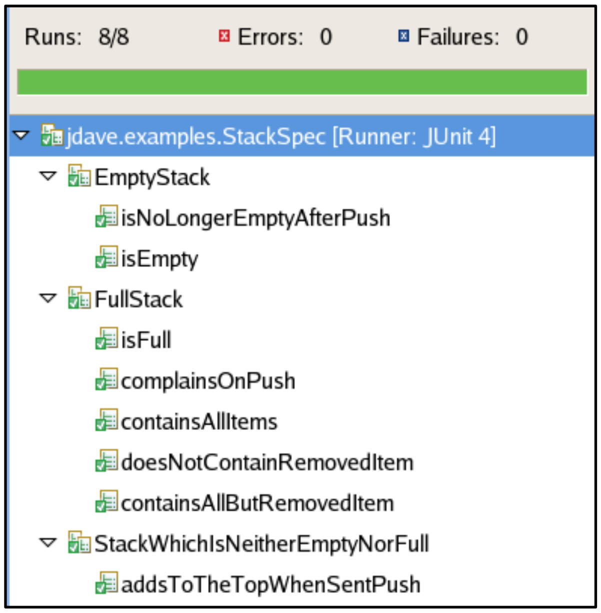

```
Runs: 8/8    Errors: 0    Failures: 0

jdave.examples.StackSpec [Runner: JUnit 4]
  EmptyStack
    isNoLongerEmptyAfterPush
    isEmpty
  FullStack
    isFull
    complainsOnPush
    containsAllItems
    doesNotContainRemovedItem
    containsAllButRemovedItem
  StackWhichIsNeitherEmptyNorFull
    addsToTheTopWhenSentPush
```

Figure 3: JUnit Screenshot

It is important to note that with each step the developer runs *all the tests*, not just the one she is working on, to verify that they all pass. It is also important to note that *the tests are part of the project source code*. The tests are not just used and thrown away during development; the tests comprise a living artifact that is continually updated as each new feature is added. Because the tests are code themselves, running the tests is quick and easy. Generally the rule is: no code is committed to source control unless all tests are green (Beck, 2003).

Many Agile shops use a *Continuous Integration* (CI) environment to protect and integrate the source code. This is a separate job that either monitors the code repository for changes to code, or runs on a timed schedule. Each time the job triggers, a full build is run, then a full test suite is launched against the entire codebase. If any code does not compile, or if any test fails, the whole team is notified that “the build is broken.” CI provides immediate feedback so a developer is aware of not just when he has broken his own code, but when his work has broken anybody else’s code also, so he can fix it right away.

Continuous Integration, like TDD, is not platform-specific. There are many vendors producing CI frameworks, some of which are open source. Some popular CI tools are: Build Forge, CruiseControl, Hudson, Jenkins, and TeamCity.

Story Testing

All of the above history leads us to Story Testing, which is the primary focus of this research paper. It is another Agile development technique. Building on practices begun with TDD, Story Testing takes the whole process to a higher level. As noted earlier, Agile methodologies use User Stories to deliver and document business requirements. The idea of Story Testing is to translate these User Stories (business requirements) into a series of Acceptance Tests. Acceptance Tests are more rigorous than free-form text documents. Ideally these tests can then be translated directly into executable code.

The set of ideas that this paper calls “Story Testing” has many names. Ward Cunningham initially introduced the term “Acceptance Testing” at an XP/Agile conference in 2002 (Cunningham). In 2008 Elizabeth Hendrickson proposed a very similar concept, which she called “Acceptance Test Driven Development” (Hendrickson, 2008). She proposed writing feature-level tests before development, and she stated that the tests should describe expectations of the behavior of the software. “Typically these tests are discussed and captured when the team is working with the business stakeholder(s) to understand a story on the backlog” (Hendrickson).

Another name for the concept came from a paper from Dan North, when he described a process he calls “Behavior Driven Development”, or BDD. He proposes “…a ubiquitous language for analysis” (North, 2006).

Ubiquitous Language

The term “Ubiquitous Language” comes from the concept of Domain Driven Design, laid out by Eric Evans in his book by the same name (Evans, 2004). Domain Driven Design suggests that the language domain of the business should be reflected in the software code itself. As much as possible, the names of objects in the codebase should match the names used by the business. This is in order to “...minimize misunderstandings between users of the software and the developers by establishing and using a common vocabulary” (Farley, 2007). A ubiquitous language can help bridge the gap between technical people and business people on a project, by giving them a common language in which to communicate (Avram, 2008).

This concept was taken further with the idea of a Business Readable DSL, or Business Readable Domain Specific Language. The basic idea of a Domain Specific Language is “…a computer language that's targeted to a particular kind of problem, rather than a general purpose language that's aimed at any kind of software problem” (Fowler, May 2008). When used with regard to software development, a Business Readable DSL is a computer language that a business person can read, understand, and perhaps even write (Fowler, Dec 2008).

The goal of a Story Testing framework is to capture software application requirements in a Business Readable DSL – a single language that can be both understood by the business and verified by a computer. Today's Story Testing frameworks can be loosely categorized into two different groups: Behavior Driven Development frameworks, and Acceptance Test Driven Development frameworks.

Behavior Driven Development Frameworks

When Dan North introduced the concept of BDD, as mentioned earlier, he noted that he was trying to provide “…a ubiquitous language for analysis” (North, 2006). He felt that the word "test" in Test Driven Development was confusing and limiting. Instead he proposed looking at tests in terms of ways to express behavior. A story's behavior is nothing more than the sum of its acceptance criteria. When he wrote the paper, it was already common to write User Stories with the format:

> *As a* [x],
> *I want* [y],
> *So that I can* [z].

For example:

> *As a* student,
> *I want* to see which answers I got wrong on the quiz,
> *So that I can* study these questions before the final.

This format forces the writer to think about not just *what* the software should do, but *why*. If the writer cannot think of a "why", then they might realize the requirement is not so important after all, reducing the scope of the project. The very language can change how you think about the requirement, and force you to think about the business value up front.

In his paper, North proposed extending the above syntax with a loose template for describing Acceptance Criteria in terms of *Features* and *Scenarios*. A feature is a story, and it can be broken down into multiple scenarios. A scenario can take the form:

> *Given* [A],
> *When* [B] occurs,
> *Then* [C] should be true.

For example:

> *Given* a student has grades of 80%, 75%, and 85%,
> *When* the student gets a grade of 100% on a test,
> *Then* the student has an average grade of 85%.

See Figure 6 for a full example that includes a feature and scenarios.

Also in this paper, North stated that Acceptance Criteria should be executable. The template provides fine-grained enough expectations that it should be possible to translate this into code. To prove this claim, he built and introduced a framework that he called *JBehave* (“JBehave,” n.d.). JBehave is a Java implementation of a Behavior Driven Development environment. Initially begun in 2003, JBehave is the one of the first examples of an effort to write a framework for executable story tests. We will discuss JBehave as a framework in greater detail shortly.

JBehave is no longer the only BDD-style framework. Dan North’s paper, along with JBehave, produced a flurry of interest and activity in BDD. Dave Astels built a popular BDD framework called *RSpec* (“RSpec,” n.d.), written in Ruby. RSpec is aimed more at the code level (closer to TDD than BDD), but it inspired *RBehave*, a Ruby implementation of JBehave. RBehave is aimed at the application level – this makes it more granular and thus more behavior-driven (North, 2007). All of these frameworks ultimately inspired a framework called *Cucumber* (“Cucumber,” n.d.). To address flaws found in the RSpec “Story runner”, Aslak Hellesøy began the Cucumber framework in 2008. It quickly gained strong support, with over 250 developers contributing to the project when 1.0.0 was released in 2011 (Hellesøy, 2011). We will discuss Cucumber in greater detail shortly.

For an interesting comparison of the popularity of the BDD frameworks listed above, we can run a search using Google Trends ("Google trends," n.d.). Google Trends is a handy tool for graphing the volume of Google searches on a particular word or phrase. For example, Figure 4 shows the comparative popularity of searches referencing BDD terms over the last eight years.

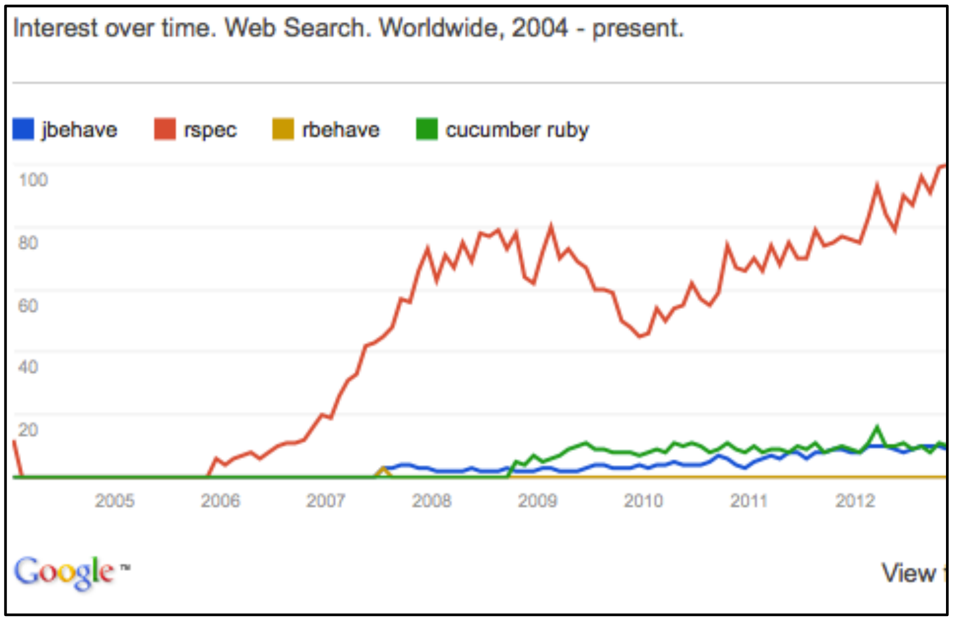


Figure 4: Google Trends for BDD search terms

While interesting, the validity of the data is challenging to certify, because all kinds of irrelevant search terms could muddy the results. For example while it looks like *RSpec* has a very high lead, there is concern that some other searches might be making it look more popular than it really is. Still, it does appear that RSpec is significantly more popular than its predecessor, JBehave. Because "Cucumber" is a common noun, we enhanced the search with the word "ruby," the language in which it is developed. Unfortunately that will have masked some valid searches, affecting the data. It is unfortunate that Cucumber does not have a more unique name, which would make the graph more relevant. Clearly the graph is interesting to consider, but not as precise as one would desire.

Acceptance Test Driven Development Frameworks

In Elizabeth's paper on Acceptance Test Driven Development (ATDD) mentioned earlier, she describes the ATDD Cycle: "Discuss the requirements, Distill tests in a framework-friendly format, Develop the code (and hook up the tests), Demo the feature" (Hendrickson, 2008). She notes that the "develop" part of the cycle is complex, and should employ traditional

TDD. There is also an exploratory testing step squeezed in before the "demo" step, and she notes: issues which came up during the exploratory testing should be brought with the product owner during the demo.

While BDD was growing in popularity in some circles, other circles were developing ATDD Story Testing efforts along a parallel path. Similar though not precisely the same, ATDD frameworks allow a user to define the test in advance of implementation. These frameworks included: FIT, FitNesse, Concordian, and Robot Framework.

Ward Cunningham initially introduced the term "Acceptance Testing" (Cunningham, 2002). He detailed three necessary artifacts:

- Test Suite Repository, where tests could be preserved and protected as code,
- Test Suite Browser, where tests could be written, run, and failures observed, and
- Test Fixture, for code objects suitable for performing a test.

Cunningham described "…the act of acceptance test development as one of joint authorship where the customer and developer collaborate in the writing of satisfied tests" (Cunningham 2002).

Shortly after the publication of the above paper, Cunningham built and introduced a framework called *FIT*, or Framework For Integrated Tests (Cunningham, 2007). FIT is a tool meant to enhance communication and collaboration among customers and programmers. As he says, "Fit gives customers and programmers a way to communicate precisely about their software." Fit uses MS Word to build HTML tables. These documents are then interpreted by a "fixture," which is a piece of code that retrieves the specifications from the document and runs them against the actual application (Cunningham, 2005). FIT then confirms these behaviors against the actual working code and provides feedback – "…thus building a simple and powerful bridge between the business and software engineering worlds" (Cunningham, 2007).

FitNesse, begun by Robert Martin in 2004, is built to be a wiki front-end on top of the Fit framework ("Fitnesse history," n.d.). Martin describes FitNesse as a "Software Development Collaboration Tool" (Martin, n.d.). While Fit is a strong framework for running test tables, it does not provide an easy means of creating those tables or displaying the results of those tests. This is what FitNesse adds. However, like Fit, it also "…enables customers, testers, and programmers to learn what their software should do, and to automatically compare that to what it actually does do" (Martin).

The FitNesse documentation says that it is three things: a software testing tool, a wiki, and a web server. This makes it unique from any of the frameworks described so far, because it is an "all-in-one" package. A user can go to the FitNesse wiki, write a test, run it, and see the results, all on the same web page. We will discuss FitNesse in greater detail shortly.

Concordion was also inspired by Fit. David Peterson created it in 2006 (Peterson, n.d.). Concordion uses what Peterson calls "Active Specifications", which live in the markup of its test pages. It creates very clean test pages, all written in HTML.

Robot Framework ("Robot framework," n.d.) is another similar framework. It began in 2006 as the master's thesis of Pekka Klärck (Laukkanen, 2006). It is "…a Python-based keyword-driven test automation framework for acceptance level testing and Acceptance Test Driven Development" (Michalak & Laukkanen, 2008). Test cases are written in tabular format, and they can be saved in either HTML or in TSV (tab separated value) format. Robot has grown a following, which has helped the framework to grow more robust. It has its own Robot Framework IDE, and it integrates with another popular test automation framework: Selenium.

Once again, to see if it showed any interesting results, we put together a search using Google Trends. Figure 5 shows the comparative popularity of searches referencing ATDD terms over the last eight years.

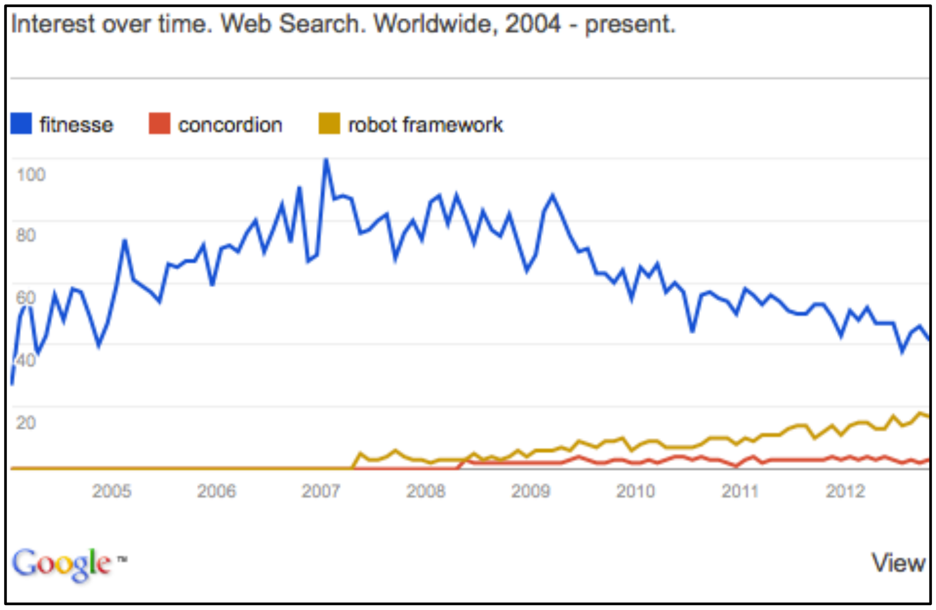


Figure 5: Google Trends for ATDD terms

This graph data is still not reliable enough for research or analysis, because all kinds of irrelevant search terms could muddy the results. For example, while it looks like *FitNesse* has a very high lead, it is probable that some searches for the word "fitness" might be making it look more popular than it really is. It still provides some interesting food for thought.

Types of Natural Language Requirements

One of the primary goals of Story Testing is to write requirements in a natural language. By "natural language" we do not necessarily mean a person's natural speech. Instead the goal is to create a Business Readable Domain Specific Language – a language that feels natural to a non-technical person, but is precise enough for a computer to interpret.

Given this goal to provide a framework in which requirements can be written in a natural language, it will be useful to discuss some natural languages. There are a variety of approaches to writing natural language requirements. Two of the most commonly used in the frameworks described above are Gherkin (also known as Given/When/Then), and Decision Tables (also known as Truth Tables).

Given/When/Then and Gherkin

The Given/When/Then template that North suggested in his original paper grew to be a specification and language in its own right with the introduction of the Cucumber framework. Cucumber's specifications are written in *Gherkin*. A gherkin is a savory pickled cucumber, but it is also both the name of a language and a piece of software that interprets the Gherkin language. Since Gherkin is a stand-alone language in its own right, other frameworks are also able to use it, and they do – it is no longer only in use by Cucumber.

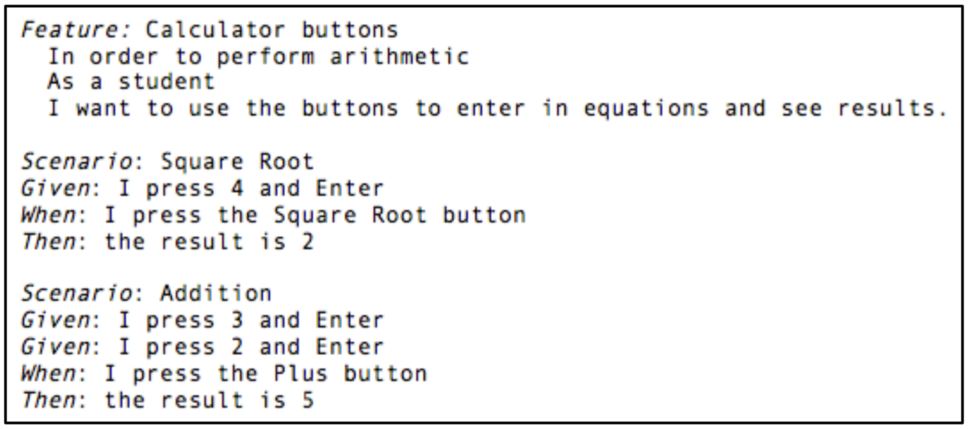

```
Feature: Calculator buttons
  In order to perform arithmetic
  As a student
  I want to use the buttons to enter in equations and see results.

Scenario: Square Root
Given: I press 4 and Enter
When: I press the Square Root button
Then: the result is 2

Scenario: Addition
Given: I press 3 and Enter
Given: I press 2 and Enter
When: I press the Plus button
Then: the result is 5
```

Figure 6: Gherkin in action

The Gherkin documentation describes Gherkin as "…a Business Readable, Domain Specific Language that lets you describe software's behavior without detailing how that behavior is implemented" ("Gherkin", n.d.). Gherkin uses *Features* as top-level elements, which are broken down into *Scenarios* ("Behat – Writing Features", n.d.). A scenario follows the given/when/then model. Example scenarios for a simple calculator application can be seen in Figure 6.

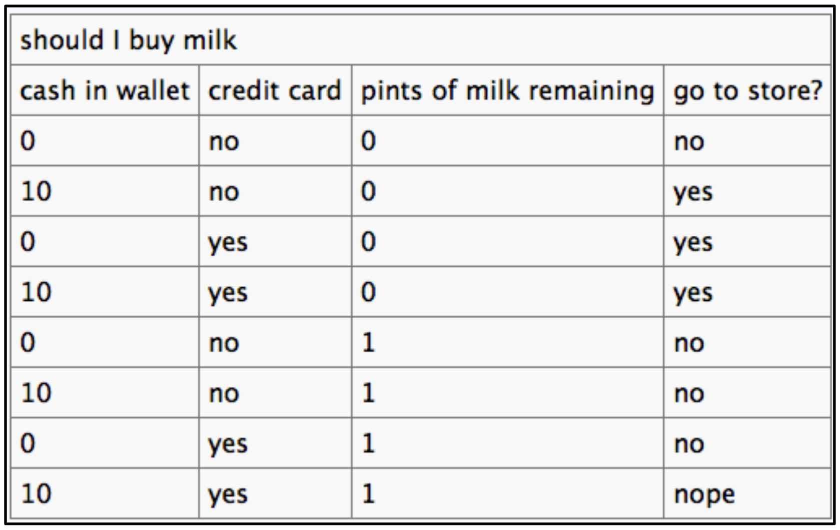
Decision Tables

The ATDD frameworks generally use tables for their natural language. For example, FitNesse and FIT use Decision Tables – also known as Truth Tables. The concept of using decision tables in computing has been around for decades. Lew and Tamanaha wrote about it in their 1976 paper, "Decision table programming and reliability." They describe the tool thus: "A *decision table* is commonly viewed as a functional description, which maps inputs (conditions) to outputs (actions) without necessarily specifying the manner in which the mapping is to be implemented" (Lew & Tamanaha, 1976). They also note that decision tables may be thought of as computer programs written in a high-level language, with straightforward syntax and formal mapping. They state that any Turing Machine program, and any flow-chart, can be emulated with a decision table.

Decision tables, being multi-dimensional, are more readable than mathematical notation or computer code (Janicki & Wassyng 2003). This, along with their precision, makes them good candidates for natural language specifications.

In FitNesse, a decision table begins with a title, which asks a question. It then contains a series of inputs and outputs, indicating the answer to that question. Figure 7 shows an example taken from the FitNesse User Guide ("FitNesse Decision Table," n.d.).

The output column is indicated by a "?" at the end of its title. Each row answers the table's question based on its variables. Row 1 says: if you have $0 in cash, no credit cards, and no milk, do not go to the store. Row 2 says: if you have $10 cash, no credit card, and no milk, then yes, you should go to the store and buy some milk.



should I buy milk			
cash in wallet	credit card	pints of milk remaining	go to store?
0	no	0	no
10	no	0	yes
0	yes	0	yes
10	yes	0	yes
0	no	1	no
10	no	1	no
0	yes	1	no
10	yes	1	nope

Figure 7: FitNesse Decision Table

The table basically indicates that the reader should buy milk only if he has no milk, and can afford to buy some. It is a silly example, but we can see how a table that lists all the variable combinations will help in defining software requirements. Although this is not a "natural" language exactly, a human being can easily parse the logic, and all of the rules are clearly laid out. Decision tables are useful in cases like the above example, where there are several variable inputs that will affect the output.

Other Story Testing Frameworks

Although this research has focused primarily on the most popular frameworks, many other "one-off" types of frameworks are being developed and used. Each framework is an example of developer or group of developers who appreciated the Story Testing concept, but had a need that the existing frameworks did not support. For example, *Specflow* uses the Gherkin language in a .NET environment ("Specflow," n.d.), while *Behat* uses Gherkin in a PHP environment ("Behat," n.d.).

StoryQ and *BDDfy* do not use Gherkin, but they each also run in a .NET environment. StoryQ is meant to be portable, running in a single .dll file ("StoryQ," n.d.), while BDDfy was created to be used by non-agile teams. Finally *GivWenZen* is a framework that uses the given/when/then approach to BDD while running in the FitNesse environment (Williams, 2011). This list is not comprehensive, because new frameworks appear all the time.

How to Use a Story Testing Framework

To summarize all of the history and details described above, we will walk through the steps a project team would take to actually use a Story Testing framework to develop a software application. These steps do not need to reference a specific framework. Although each framework works slightly differently, they all share the following common functionality and approaches.

Write the Story Tests

The first step is to write the Story Tests, or product requirements, in whatever natural language is available with the framework. This involves going to the place where the tests will be stored and accessed, using either a standard text editor or an HTML/wiki page. As the requirements are defined, somebody will write one or more Story Tests for each requirement. That "somebody" should be either a business stakeholder, or an analyst working directly with a business stakeholder, to make sure that they match what the business needs.

Run the Story Tests Regularly

The next step is to run the tests, in order to see what tests pass and what do not. Most of the Story Testing frameworks utilize some form of HTML reporting. The reports must be accessible in some form or another, whether it be a web page to visit, a regular email or report, or a chart on the wall. Once the reports are accessible, the next step is to use them. This feedback loop is at the heart of Story Testing: at any time, a business person should be able to use the Story Test reports to get a full understanding of how many of the requirements are working, and what is still left to be done.

Generally the running of the tests should be automated in some fashion, although they should also be accessible for ad-hoc runs. This automation can be done in the same *Continuous Integration* environment discussed in the TDD section above. If the tests are run automatically run every night, for example, the reports can be the pulse of the project.

Develop the Application Code

The development team then uses its standard practices to develop the application code, whatever those practices may be. As each story is completed, the above-mentioned reports will show that the percentage of completed requirements has gone up. The development team continues to work until all of the requirements show as "passing" in the reports.

Story Test Reports Reveal Project Status

Each framework provides its own flavor of reports, but at a minimum every framework reports whether a story is “passing” or “failing”. Some frameworks offer other statuses as well, such as “pending”. An application will have many requirements. It is the team’s goal to move each requirement from “not started”, to “in progress”, “complete”. The framework’s reports indicate the status of each requirement, and thus the status of the project as a whole.

Suppose a team has two completed requirements, “A” and “B”. As the team is working on requirement “C”, suppose they accidentally change the behavior of a feature that was needed for requirement “B”. When the automated tests run that night, the team will be alerted that “B” is no longer in a passing state. The team will see that previously-implemented functionality is no longer working, and the product owner will also be alerted that “B” has gone from “passing” to “failing”. At this point the team and product owner can look at the Story Tests for both features, figure out what the functionality ought to be, and correct it. The change might involve a change to the code in requirement C, because it may have been a bug. However it might instead produce a change to the Story Tests for requirement B, because the work on requirement C gave the business a deeper understanding of what the application as a whole is meant to do.

Managing Changing Requirements

The scenario above leads smoothly into a discussion of managing changing requirements. Story Testing is touted as a way of mitigating the risk of changed requirements. It can do this in the same way it protects the code from broken features. If a requirement changes, somebody (an analyst or business representative) records this change by changing, rewriting, or adding to the existing Story Tests. The next time the reports run, one or more features will show up as “failing”. Once this happens, there is no way for the requirement to be missed or dropped. The team will have to examine the features and the code that supports them, and get everything to a “passing” state before the project is declared complete.

Story Tests are executable requirements. Because the requirements are executable, and because they run against the application code itself, they guarantee that the code implements each requirement. Once every test passes, we know that the application meets the requirements.

What This Research Will Explore

As the literature review has revealed: many Story Testing frameworks are competing in the market right now. However despite the wealth of frameworks out there, Story Testing is still not very widely adopted. This paper will explore why this might be, by implementing three different Story Testing frameworks with the same set of business requirements. The paper will also provide three different end-to-end examples, while reducing as many variables as possible. The frameworks selected for this research are: Cucumber, FitNesse, and JBehave.

Cucumber

As we noted earlier, Cucumber has a large following and a wide developer network. It is relatively modern, works with many programming languages, and it uses the Gherkin language for writing requirements. As this researcher is primarily a Java developer, the Methodology section will use Cucumber-JVM, which is Cucumber's Java implementation.

FitNesse

FitNesse was selected because it is the most popular among the ATDD frameworks, and it also has a large following. FitNesse uses Decision Tables for writing requirements, and all of the work is done in a wiki, which makes it a very different flow than Cucumber. It supports Java and .NET programming languages.

JBehave

The final framework selected is JBehave. JBehave was selected because it is the original BDD framework. It has been around a long time, and also has a strong following. It only supports the Java programming language, but that works perfectly for this research, since the other two frameworks are also Java-based.

Criteria for Judging Story Testing Frameworks

As we implement the three frameworks, in order to compare them with one another we need a set of criteria for judging them. Based on the review of Story Testing frameworks thus far, we see that new frameworks keep popping up. Each new framework is meant to fix a perceived lack in the existing frameworks. The same issues that have necessitated the creation of new frameworks can also be used as criteria for judging a framework. Based on the literature reviewed so far, this paper will use the following criteria.

- *Usability* of the framework – How easy is it to find and work with the Story Tests? Can the reports be run ad-hoc, by a non-technical person?
- *Readability* of the specifications - Can a non-technical person understand what the specifications are telling them? Can a non-technical person write one, or at least modify an existing Story Test?
- *Understandability* of the results - How easy is it to tell if the requirements have been met? How easy is it to tell what went wrong? Do the results provide just pass/fail, or do they have more states, like "pending"?
- *Develop-ability* - How easy is it to develop the fixtures/hooks to the application itself? How easy is it to back up the Story Tests?
- *Expandability* - How easy is it to add a new requirement? How easy is it to change an existing one?

In Chapter Three, *Methodology*, we will implement each framework. Then in Chapter Four, *Results and Analysis*, this set of criteria will be used as one means of judging the three frameworks.

Chapter 3 – Methodology

The goal of this research was to implement and explore three current Story Testing frameworks, in order to analyze their effectiveness in a real-world situation. This exploration and evaluation should enable us to better understand how to use Story Testing frameworks, and also should allow us to consider reasons why Story Testing is not more widely adopted in software projects. It should also allow us to discuss whether Story Testing is a viable solution to requirements dilemmas.

The best way to compare complex items is to eliminate as many irrelevant differences between them as possible. By fixing most of the variables, the differences and similarities between the items will become more obvious. If all three frameworks can be implemented from the same original application requirements, each framework implementation will have to deal with the same amount of complexity, the same number of rules, the same number of scenarios, and the same size of data. The frameworks chosen also all use Java for their back-end implementation, and the same developer implemented them all, eliminating two more variables. Reducing unrelated differences allowed true differences – and true similarities – to appear.

Design Science

The approach the paper took to achieve this goal was a form of Constructive research called Design Science. Design Science "specifically focuses on tackling ill-structured problems in a systematic manner" (Holmstrom, Ketokivi, & Hameri, 2009). It emphasizes the process of "exploration through design: design science is research that seeks (i) to explore new solution alternatives to solve problems, (ii) to explain this explorative process, and (iii) to improve the problem-solving process" (Holmstrom et al.).

Design Science was particularly helpful to this research, because it enabled the design and construction of comparable artifacts. The fundamental principle of Design Science is that "…knowledge and understanding of a design problem and its solution are acquired in the building of an application or an artifact" (Hevner, March, Park, & Ram, 2004). The process of implementing the identical problem in three different frameworks should heighten our understanding of each of them, allowing us to explore the differences and similarities between them.

The Sample Application

The sample application has a fairly simple user interface, but a complex set of business rules. Imagine a web application that is a tool for addressing envelopes for formal communication, such as wedding invitations. A user can upload a spreadsheet of the names and addresses of their guests, and the application will format the names and addresses according to style and etiquette guidelines.

The formal rules for addressing envelopes are complicated and archaic, so it is helpful to have a tool to do it for you. The sample application will format names only. To keep things simple for the sake of the example, addresses will be kept out of scope.

This application is a good candidate for Story Testing, because there is a clear separation between the User Interface and the Business Rules. While the User Interface might involve logging in, uploading files, and displaying results, that has nothing to do with the actual Business Rules – the way to format different types of names.

Business Rules

The following guidelines, pulled from the *Martha Stewart Weddings* website (Stewart), provide a nicely complex set of rules for formatting names on envelopes.

> **Names, formal**
> Your guests' names should be written in full on outer envelopes -- no nicknames or initials. Use the appropriate social titles as well, such as addressing married couples as "Mr. and Mrs." If a man's name has a suffix, write "Mr. Joseph Morales, Jr.," or "Mr. Joseph Morales IV"; "Junior" can be spelled out on a more formal invitation. It gets a little tricky when husband, wife, or both have different professional titles. If the husband is a doctor, for example, the titles will appear as "Doctor and Mrs."; if the wife is a doctor, her full name would come first, as in "Doctor Sally Carter and Mr. John Carter." If both are doctors, write "The Doctors Carter." If they have different professional titles, list the wife first: "The Honorable Pamela Patel and Lieutenant Jonathan Patel, U.S. Navy." If a wife has kept her maiden name, her name should appear first and be joined with her husband's using "and."
>
> **Names, Informal**
> To some couples, omitting wives' first names feels too old-fashioned; including the first names of both husband and wife after their titles is appropriate. The house number, even though it is less than 20, can be written as a numeral for a less-formal feeling.
>
> **Different Last Names**
> When a husband and wife have different last names, the wife's name is traditionally written first. Connecting the couple's names by the word "and" implies marriage. For an unmarried couple that lives together, names should be written on separate lines without the word "and." On the inner envelope, both are addressed by their titles and respective last names.

This is a great example of how software requirements might traditionally come to a development team. The requirements contain multiple paragraphs, with complex rules strung together. Although the document contains some examples, there are not nearly as many examples as there are rules. There are gaps in the details that are left to the imagination, or assumed to be obvious. For example, in the formal section it does not actually say that the wife's name should be omitted. That does not become clear until we get to the informal section, when the document notes, "omitting wives' first names feels too old-fashioned."

To explore Story Testing frameworks, this research implemented the above "free-form paragraph" requirements in three different popular frameworks.

Cucumber Implementation

The first Story Testing framework implemented was *Cucumber JVM* ("Cucumber," n.d.). As we saw in the literature review, Cucumber uses Gherkin's "given-when-then" approach to writing requirements. It also allows a tabular approach to adding data, which worked very neatly in this context.

Cucumber Requirements

The Cucumber requirements began by defining a *feature* – in this case, "Names for Wedding Labels". Then all of the requirements above were carefully analyzed, and a number of *scenarios* – cases for which the results may differ – were identified. In essence, each scenario provides an example of a single rule, although we also broke each rule down as either "informal" or "formal". Consider the very first rule listed in the Martha Stewart requirements:

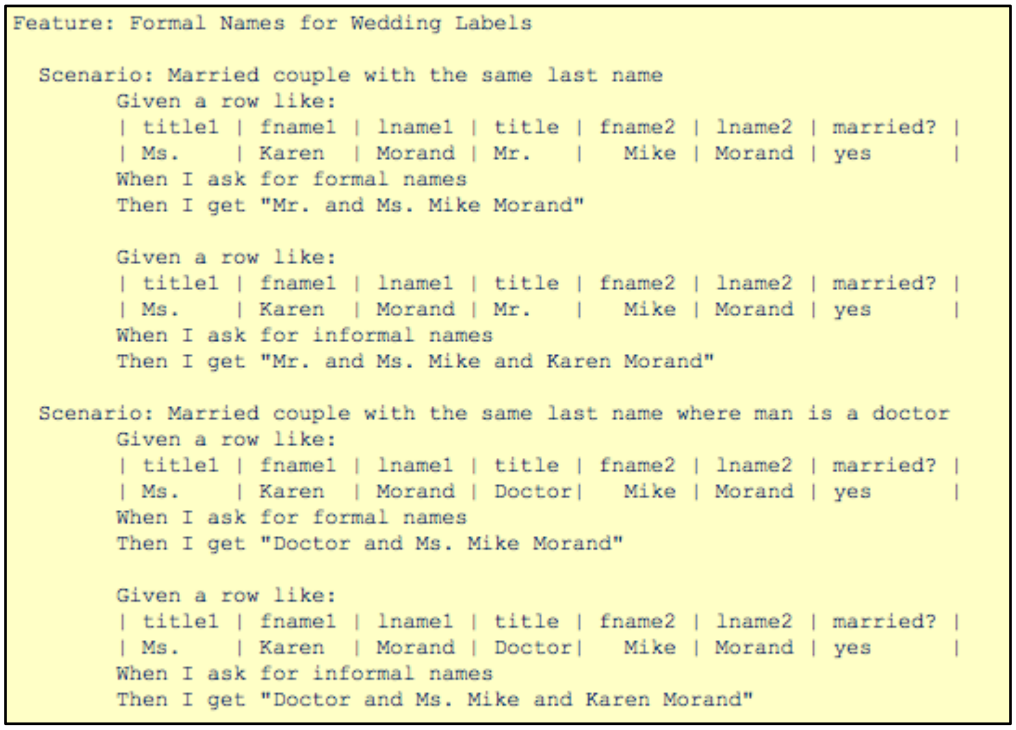
> Your guests' names should be written in full on outer envelopes -- no nicknames or initials. Use the appropriate social titles as well, such as addressing married couples as "Mr. and Mrs." (Stewart).

To write a Cucumber scenario for this single rule, example data had to be created to represent the rule. The "Given" part of the scenario must describe the initial state of all the data for the example. The input data will be full names for two married guests. The guests share a last name. The following data (with examples filled in) is needed:

First Name 1	Karen
Last Name 1	Morand
Title 1	Ms.
First Name 2	Michael
Last Name 2	Morand
Title 2	Mr.

Now that the “given” clause is identified, the next step is the “when” clause. The “when” clause is a user-requested action, or a system-generated action. In this case, “When I want formal names” would be a good “when”. “When I want informal names” would be another good “when”.

According to Martha Stewart’s rules, the result for this data would be formally written: “Mr. and Ms. Michael Morand.” An informal representation would be, “Mr. and Ms. Michael and Karen Morand.” Those are the “then” clauses.



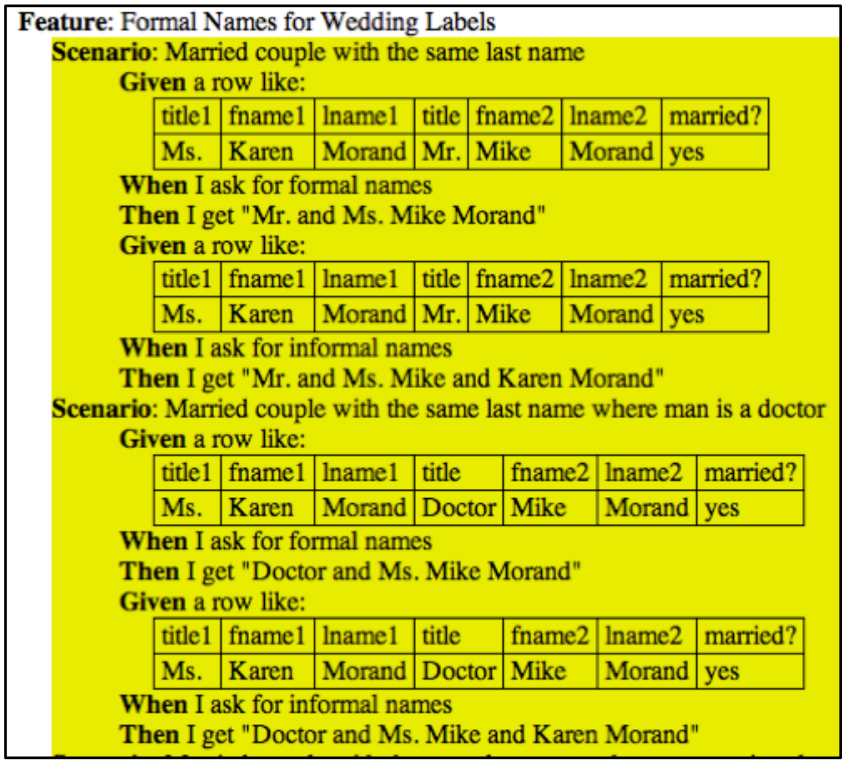
```
Feature: Formal Names for Wedding Labels

  Scenario: Married couple with the same last name
        Given a row like:
        | title1 | fname1 | lname1 | title | fname2 | lname2 | married? |
        | Ms.    | Karen  | Morand | Mr.   |   Mike | Morand | yes      |
        When I ask for formal names
        Then I get "Mr. and Ms. Mike Morand"

        Given a row like:
        | title1 | fname1 | lname1 | title | fname2 | lname2 | married? |
        | Ms.    | Karen  | Morand | Mr.   |   Mike | Morand | yes      |
        When I ask for informal names
        Then I get "Mr. and Ms. Mike and Karen Morand"

  Scenario: Married couple with the same last name where man is a doctor
        Given a row like:
        | title1 | fname1 | lname1 | title | fname2 | lname2 | married? |
        | Ms.    | Karen  | Morand | Doctor|   Mike | Morand | yes      |
        When I ask for formal names
        Then I get "Doctor and Ms. Mike Morand"

        Given a row like:
        | title1 | fname1 | lname1 | title | fname2 | lname2 | married? |
        | Ms.    | Karen  | Morand | Doctor|   Mike | Morand | yes      |
        When I ask for informal names
        Then I get "Doctor and Ms. Mike and Karen Morand"
```

Figure 8: Cucumber requirements code

Now that each piece of the test has been identified, how is it written in a way that Cucumber can understand it? The easiest way to organize this many variables is in a tabular format of some sort. Cucumber provides a “data table” format that will serve this purpose. To create a data table in Cucumber, a sort of “wiki markup” is used, where table cells are separated by the “|” character. Figure 8 is a code snippet from the Cucumber scenarios written for the above requirements. Note each test is an example for a single rule. It contains test data that depict the example, within the "given" section. The "when" describes the action taken by a user, and the "then" describes the application's expected output for that particular set of test data. Also note that the "Given" section contains table markup – one table cell for each field of the data. Please see Appendix A for all of the code artifacts from the Cucumber implementation.

Cucumber Reports

Cucumber produces an HTML report whenever it is run. As discussed in the literature review: Story Test reports will be run on a regular basis (daily, for example), and then evaluated to see how well the code is meeting the requirements.



Feature: Formal Names for Wedding Labels

Scenario: Married couple with the same last name

Given a row like:

title1	fname1	lname1	title	fname2	lname2	married?
Ms.	Karen	Morand	Mr.	Mike	Morand	yes

When I ask for formal names

Then I get "Mr. and Ms. Mike Morand"

Given a row like:

title1	fname1	lname1	title	fname2	lname2	married?
Ms.	Karen	Morand	Mr.	Mike	Morand	yes

When I ask for informal names

Then I get "Mr. and Ms. Mike and Karen Morand"

Scenario: Married couple with the same last name where man is a doctor

Given a row like:

title1	fname1	lname1	title	fname2	lname2	married?
Ms.	Karen	Morand	Doctor	Mike	Morand	yes

When I ask for formal names

Then I get "Doctor and Ms. Mike Morand"

Given a row like:

title1	fname1	lname1	title	fname2	lname2	married?
Ms.	Karen	Morand	Doctor	Mike	Morand	yes

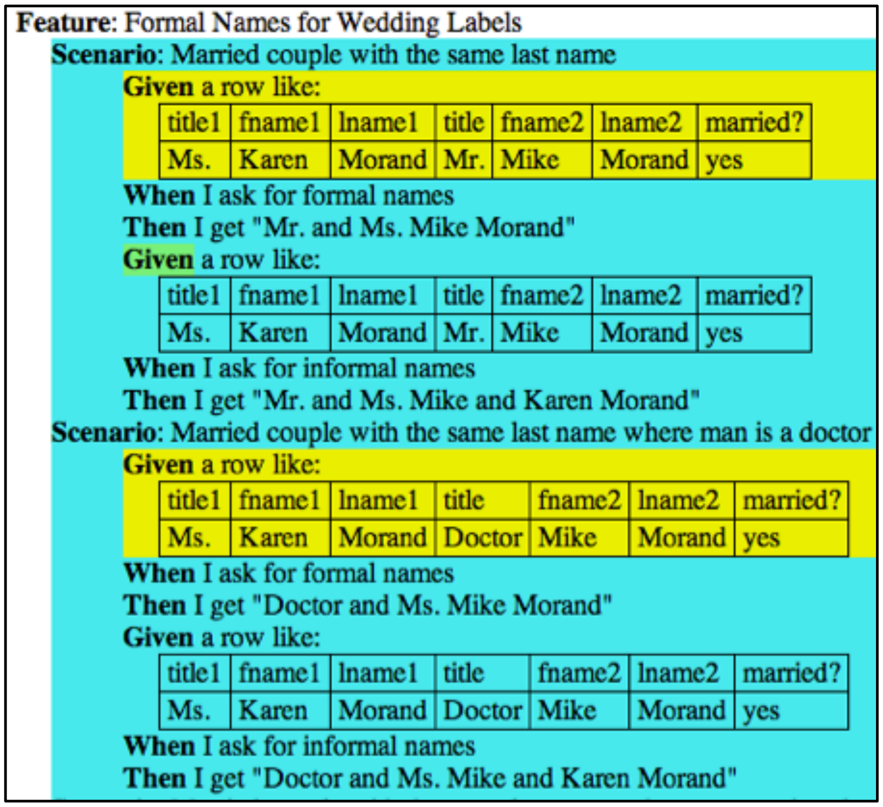
When I ask for informal names

Then I get "Doctor and Ms. Mike and Karen Morand"

Figure 9: Cucumber results when no code is written

When the Cucumber report for the above tests was run without having written any code, the HTML report gave the results seen in Figure 9. Note that the wiki markup has been translated to actual HTML tables in the report, and the keywords have been made bold.

The yellow background indicates that none of the scenarios have been implemented – meaning that Cucumber could not find code for any of the scenarios. This indicates an unimplemented framework, and it is Cucumber's standard starting state. The requirements have been built, but no code has been written. Cucumber JVM relies on Java code with which it can connect the Story Tests above to an actual running application.



Feature: Formal Names for Wedding Labels

Scenario: Married couple with the same last name

Given a row like:

title1	fname1	lname1	title	fname2	lname2	married?
Ms.	Karen	Morand	Mr.	Mike	Morand	yes

When I ask for formal names

Then I get "Mr. and Ms. Mike Morand"

Given a row like:

title1	fname1	lname1	title	fname2	lname2	married?
Ms.	Karen	Morand	Mr.	Mike	Morand	yes

When I ask for informal names

Then I get "Mr. and Ms. Mike and Karen Morand"

Scenario: Married couple with the same last name where man is a doctor

Given a row like:

title1	fname1	lname1	title	fname2	lname2	married?
Ms.	Karen	Morand	Doctor	Mike	Morand	yes

When I ask for formal names

Then I get "Doctor and Ms. Mike Morand"

Given a row like:

title1	fname1	lname1	title	fname2	lname2	married?
Ms.	Karen	Morand	Doctor	Mike	Morand	yes

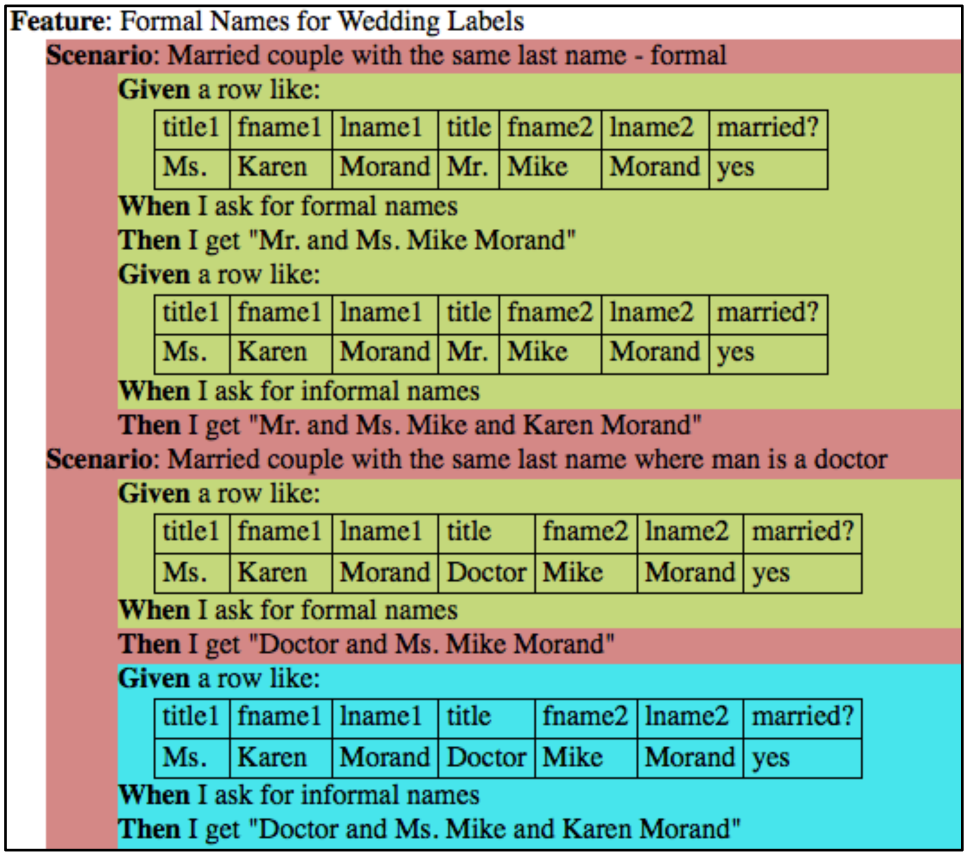
When I ask for informal names

Then I get "Doctor and Ms. Mike and Karen Morand"

Figure 10: Cucumber results with pending tests

The next step was to create "stubs" for the framework code itself. Essentially the coding framework was set up, but no application code was written. Everything was marked as "pending". This is a way of indicating a requirement that is understood, ready to implement, but it has not yet been implemented. "Pending" means incomplete, and it is a different state than "failing". A pending test is not expected to work yet, while a failing test is something which is broken. Figure 10 shows what pending tests look like: the pending tests have a blue background.

Things get more interesting when development begins. First the developer wrote just enough code to get the first test to pass, but all of the others were allowed to fail. None of the pending tests were left, so this mimicked a "broken" or "bug introduced" state. When the tests were run, they produced the report in Figure 11.


Feature: Formal Names for Wedding Labels

Scenario: Married couple with the same last name - formal

Given a row like:

title1	fname1	lname1	title	fname2	lname2	married?
Ms.	Karen	Morand	Mr.	Mike	Morand	yes

When I ask for formal names
Then I get "Mr. and Ms. Mike Morand"
Given a row like:

title1	fname1	lname1	title	fname2	lname2	married?
Ms.	Karen	Morand	Mr.	Mike	Morand	yes

When I ask for informal names
Then I get "Mr. and Ms. Mike and Karen Morand"

Scenario: Married couple with the same last name where man is a doctor

Given a row like:

title1	fname1	lname1	title	fname2	lname2	married?
Ms.	Karen	Morand	Doctor	Mike	Morand	yes

When I ask for formal names
Then I get "Doctor and Ms. Mike Morand"
Given a row like:

title1	fname1	lname1	title	fname2	lname2	married?
Ms.	Karen	Morand	Doctor	Mike	Morand	yes

When I ask for informal names
Then I get "Doctor and Ms. Mike and Karen Morand"

Figure 11: Cucumber results with failing test

The parts that passed are color-coded green. The first "then" statement is green, but the second "then" statement is color-coded red. The text that has a blue background is being skipped, because the test in front of it failed. At this point, if this were a real development project, the developer would continue to write code until all of the Cucumber scenarios turned green.

Cucumber Code

The Cucumber JVM framework is built to interact with Java code. It operates by using annotations and regular expressions to find the correct piece of code for each Given/When/Then assertion.

Cucumber offers a useful feature for developers: when it is run and cannot find the code implementing a scenario, along with producing an "undefined" message, Cucumber (via Gherkin) offers empty implementations of the methods it could not find. This allowed the developer to quickly put together the just enough code to flip the tests to "pending".

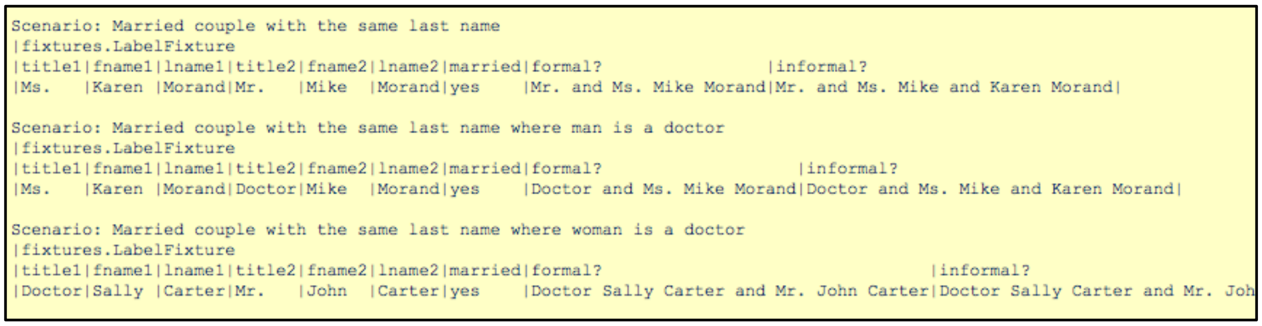
Once all the tests were pending, the developer implemented the first one. Only the tests were written for this study, not the actual wedding label generator, because that was not the goal of this research. Instead just enough code was written to make the first test pass, by hard-coding the correct result. All of the Java code for the project is in Appendix A.

FitNesse Implementation

The next Story Testing framework implemented was *FitNesse* ("Fitnesse history," n.d.). While Cucumber operates on a given-when-then approach, FitNesse uses Decision Tables – sometimes known as a Truth Table. As we saw in the literature review, a decision table consists of a name, some number of inputs, and some number of expected outputs.

FitNesse Requirements

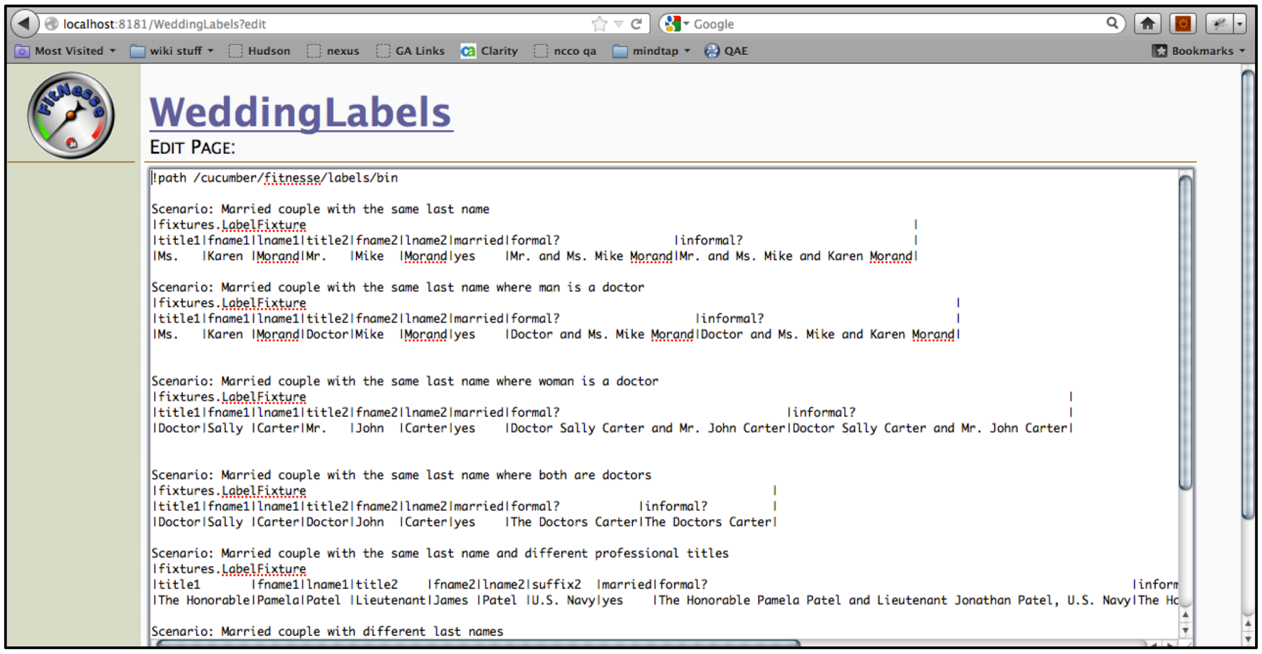
To implement the requirements in FitNesse, the developer created very wide decision tables, because both the inputs and outputs all reside in one table. FitNesse uses the pipe character to build tables, just like Cucumber. Each input field is a separate column, as well as the two expected outputs. Output columns can be recognized because the column title ends with a question mark. Figure 12 shows a snippet of the FitNesse requirements built, while all of the FitNesse implementation code can be found in Appendix B.



```
Scenario: Married couple with the same last name
|fixtures.LabelFixture
|title1|fname1|lname1|title2|fname2|lname2|married|formal?                 |informal?
|Ms.   |Karen |Morand|Mr.   |Mike  |Morand|yes    |Mr. and Ms. Mike Morand|Mr. and Ms. Mike and Karen Morand|

Scenario: Married couple with the same last name where man is a doctor
|fixtures.LabelFixture
|title1|fname1|lname1|title2|fname2|lname2|married|formal?                     |informal?
|Ms.   |Karen |Morand|Doctor|Mike  |Morand|yes    |Doctor and Ms. Mike Morand|Doctor and Ms. Mike and Karen Morand|

Scenario: Married couple with the same last name where woman is a doctor
|fixtures.LabelFixture
|title1|fname1|lname1|title2|fname2|lname2|married|formal?                                 |informal?
|Doctor|Sally |Carter|Mr.   |John  |Carter|yes    |Doctor Sally Carter and Mr. John Carter|Doctor Sally Carter and Mr. Joh
```

Figure 12: FitNesse requirements code

Rather than putting the formal and informal tests one above the other, like with Cucumber, the developer put them both in the same table. This is because they use the same setup data.



WeddingLabels

EDIT PAGE:

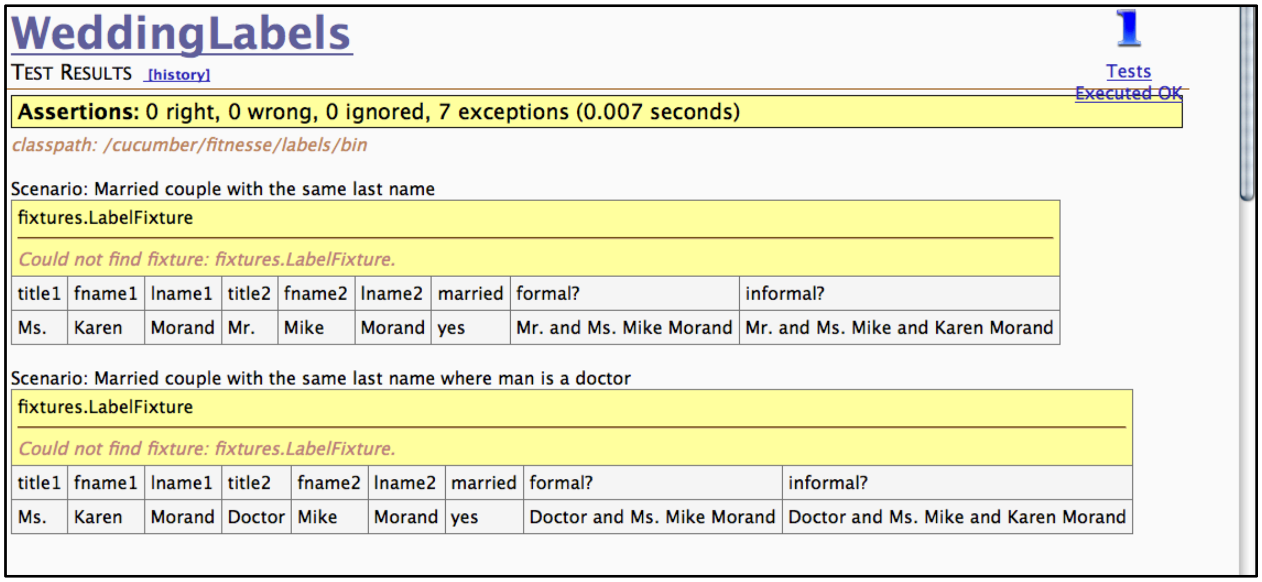
```
!path /cucumber/fitnesse/labels/bin

Scenario: Married couple with the same last name
|fixtures.LabelFixture                                                          |
|title1|fname1|lname1|title2|fname2|lname2|married|formal?                 |informal?                        |
|Ms.   |Karen |Morand|Mr.   |Mike  |Morand|yes    |Mr. and Ms. Mike Morand|Mr. and Ms. Mike and Karen Morand|

Scenario: Married couple with the same last name where man is a doctor
|fixtures.LabelFixture                                                              |
|title1|fname1|lname1|title2|fname2|lname2|married|formal?                     |informal?                          |
|Ms.   |Karen |Morand|Doctor|Mike  |Morand|yes    |Doctor and Ms. Mike Morand|Doctor and Ms. Mike and Karen Morand|


Scenario: Married couple with the same last name where woman is a doctor
|fixtures.LabelFixture                                                                         |
|title1|fname1|lname1|title2|fname2|lname2|married|formal?                                 |informal?                              |
|Doctor|Sally |Carter|Mr.   |John  |Carter|yes    |Doctor Sally Carter and Mr. John Carter|Doctor Sally Carter and Mr. John Carter|


Scenario: Married couple with the same last name where both are doctors
|fixtures.LabelFixture                                         |
|title1|fname1|lname1|title2|fname2|lname2|married|formal?           |informal?         |
|Doctor|Sally |Carter|Doctor|John  |Carter|yes    |The Doctors Carter|The Doctors Carter|

Scenario: Married couple with the same last name and different professional titles
|fixtures.LabelFixture
|title1       |fname1|lname1|title2    |fname2|lname2|suffix2  |married|formal?                                                          |inform
|The Honorable|Pamela|Patel |Lieutenant|James |Patel |U.S. Navy|yes    |The Honorable Pamela Patel and Lieutenant Jonathan Patel, U.S. Navy|The Ho

Scenario: Married couple with different last names
```

Figure 13: FitNesse requirements being edited in wiki

FitNesse Reports

An interesting feature of FitNesse is that it does not just *use* wiki-markup; it actually runs and is developed within the context of a wiki. This means that a user can access, read, write, and run story tests all from an Internet browser. Figure 13 shows the above tests being written in the FitNesse wiki.

When the above tests were written, the developer clicked "Save", then immediately selected the "Test" button to see the results. Figure 14 shows the results before having written any code.



Assertions: 0 right, 0 wrong, 0 ignored, 7 exceptions (0.007 seconds)

classpath: /cucumber/fitnesse/labels/bin

Scenario: Married couple with the same last name

fixtures.LabelFixture

Could not find fixture: fixtures.LabelFixture.

title1	fname1	lname1	title2	fname2	lname2	married	formal?	informal?
Ms.	Karen	Morand	Mr.	Mike	Morand	yes	Mr. and Ms. Mike Morand	Mr. and Ms. Mike and Karen Morand

Scenario: Married couple with the same last name where man is a doctor

fixtures.LabelFixture

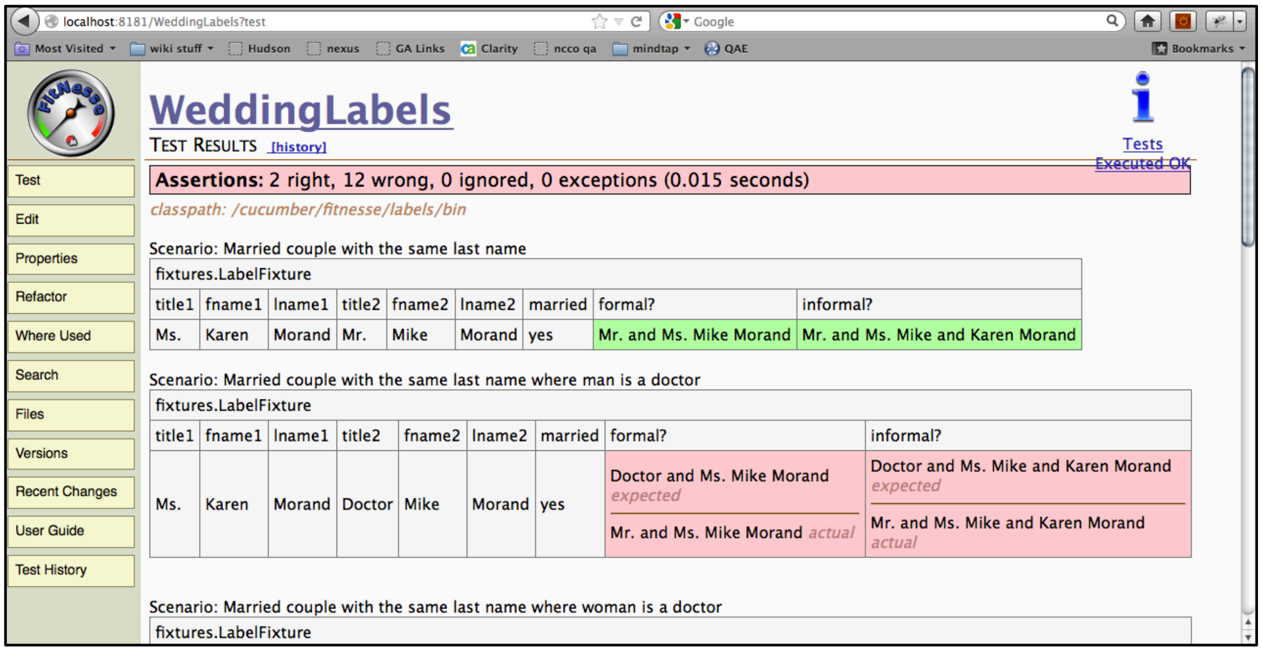
Could not find fixture: fixtures.LabelFixture.

title1	fname1	lname1	title2	fname2	lname2	married	formal?	informal?
Ms.	Karen	Morand	Doctor	Mike	Morand	yes	Doctor and Ms. Mike Morand	Doctor and Ms. Mike and Karen Morand

Figure 14: FitNesse results when no code is written

The yellow color indicates an exception occurred. Unlike Cucumber, when recording the test results FitNesse tells the user what error has occurred. Here we see that the error is: "Could not find fixture". A "fixture" is a Java object that links a specific test to the code that it is testing.

Things get more interesting when development begins. Once a fixture was written, the developer hard-coded it to return the correct data for the first formal and the informal tests. The very first two tests should pass, but all of the others should fail. This mimics a “broken” or “bug introduced” state. When we run, we get the following result in the wiki:



WeddingLabels

TEST RESULTS [history]

Assertions: 2 right, 12 wrong, 0 ignored, 0 exceptions (0.015 seconds)

classpath: /cucumber/fitnesse/labels/bin

Scenario: Married couple with the same last name

fixtures.LabelFixture								
title1	fname1	lname1	title2	fname2	lname2	married	formal?	informal?
Ms.	Karen	Morand	Mr.	Mike	Morand	yes	Mr. and Ms. Mike Morand	Mr. and Ms. Mike and Karen Morand

Scenario: Married couple with the same last name where man is a doctor

fixtures.LabelFixture								
title1	fname1	lname1	title2	fname2	lname2	married	formal?	informal?
Ms.	Karen	Morand	Doctor	Mike	Morand	yes	Doctor and Ms. Mike Morand *expected* Mr. and Ms. Mike Morand *actual*	Doctor and Ms. Mike and Karen Morand *expected* Mr. and Ms. Mike and Karen Morand *actual*

Scenario: Married couple with the same last name where woman is a doctor

fixtures.LabelFixture

Figure 15: FitNesse results with failing test

The results state that there were 2 right assertions, and 12 wrong. The successes are colored green, and the failure cases are colored red. Along with the red highlight to indicate failure, the FitNesse framework tells the user what the code actually returned, enabling the user to diagnose what is wrong or missing. This is different from Cucumber, where it is necessary to look in another place to find out what the error message was. Another difference between Cucumber and FitNesse is that there is no “out of the box” way to indicate that a story is “not yet implemented”.

FitNesse Code

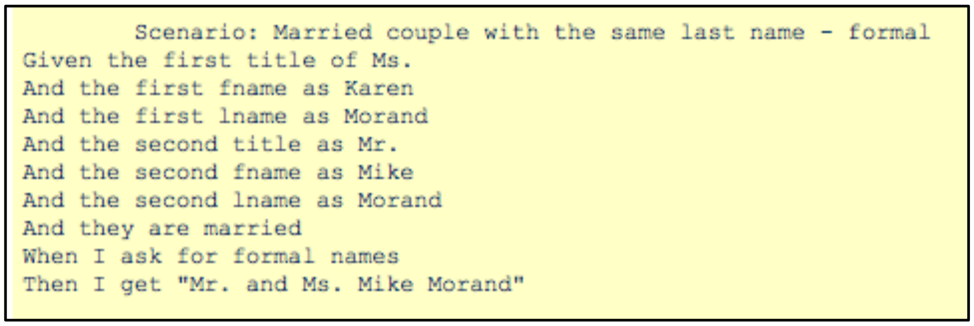
Unlike Cucumber, FitNesse does not provide a code-snippet for the developer – it has to be written by hand. It did provide the name of the class it was looking for though, because the error was “Could not find fixture: labels.LabelFixture”. FitNesse uses a combination of inheritance and convention to find the code. A class was created called `LabelFixture`, which extended the built-in `ColumnFixture`. Next methods were implemented with the same name as each of the table columns. This was enough for FitNesse to find its way to the correct code. Once again no wedding label generator was written, and the Java code is fairly rudimentary, because that was not the goal of this research. Instead the developer wrote just enough code to make the first test pass, by hard-coding the correct result. All of the Java code is in Appendix B.

JBehave Implementation

The final Story Testing framework implemented was JBehave ("JBehave," n.d.). As seen in the literature review, JBehave inspired what Cucumber is today. It is thus not surprising that the two frameworks are very similar at first glance, nor that it also uses a given-when-then approach to Story Tests.

A digression: JBehave possible implementation not taken

An astute reader might ask at this point, "Why don't you use an approach without tables for your third framework?" This was considered, and then it was rejected. The standard approach to both JBehave and Cucumber (given/when/then Story Tests) is to write them in step-by-step format, instead of tables. Figure 16 is an example of a single test written in traditional step-by-step implementation.



```
        Scenario: Married couple with the same last name - formal
Given the first title of Ms.
And the first fname as Karen
And the first lname as Morand
And the second title as Mr.
And the second fname as Mike
And the second lname as Morand
And they are married
When I ask for formal names
Then I get "Mr. and Ms. Mike Morand"
```

Figure 16: JBehave requirements (step-by-step version)

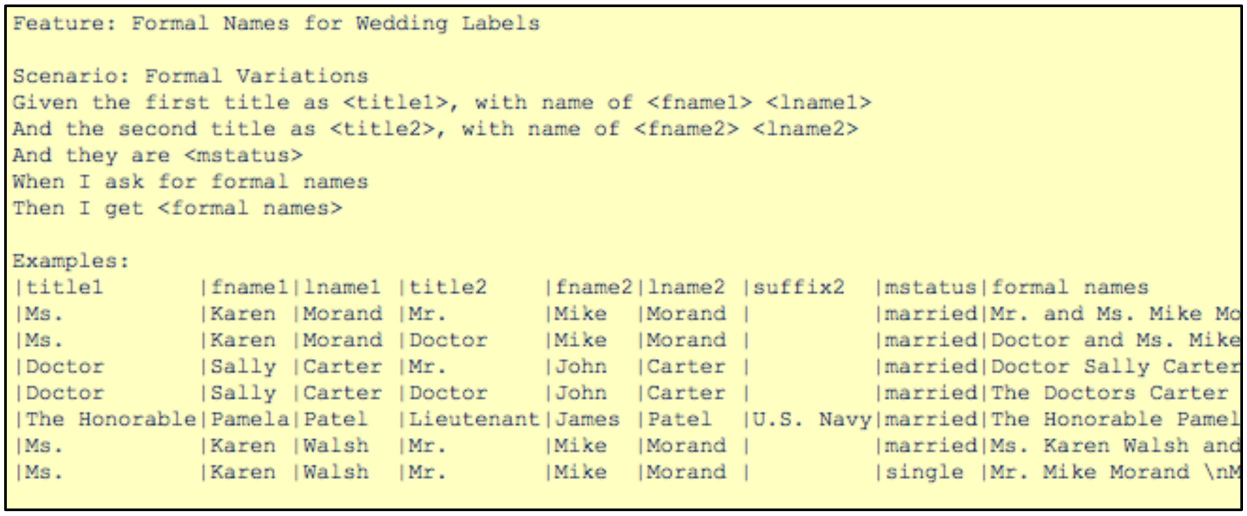
This paper did not use the step-by-step approach, because the researcher believes the approach to be too wordy – particularly in the case of large sets of setup data, like the test application has. Each field requires a separate line, making the test more challenging to digest.

JBehave Requirements

JBehave story tests are written in plain text, and JBehave has tabular options just like Cucumber. In fact, the interpreter is similar enough that the same stories used in the Cucumber implementation could be run in JBehave with just some minor reformatting.

However, JBehave also offers an interesting option called “Parameterized Scenarios”. Parameterized Scenarios allow the Story-Writer to create the scenario once, and provide many examples to be used with it. The JBehave stories were written using this approach.

With parameterized scenarios, it was possible to write the expectations just once, and then provide a table full of data examples. This kept the JBehave tests thus fairly short, as can be seen in the following snippet. All of the code for the JBehave implementation is found in Appendix C.



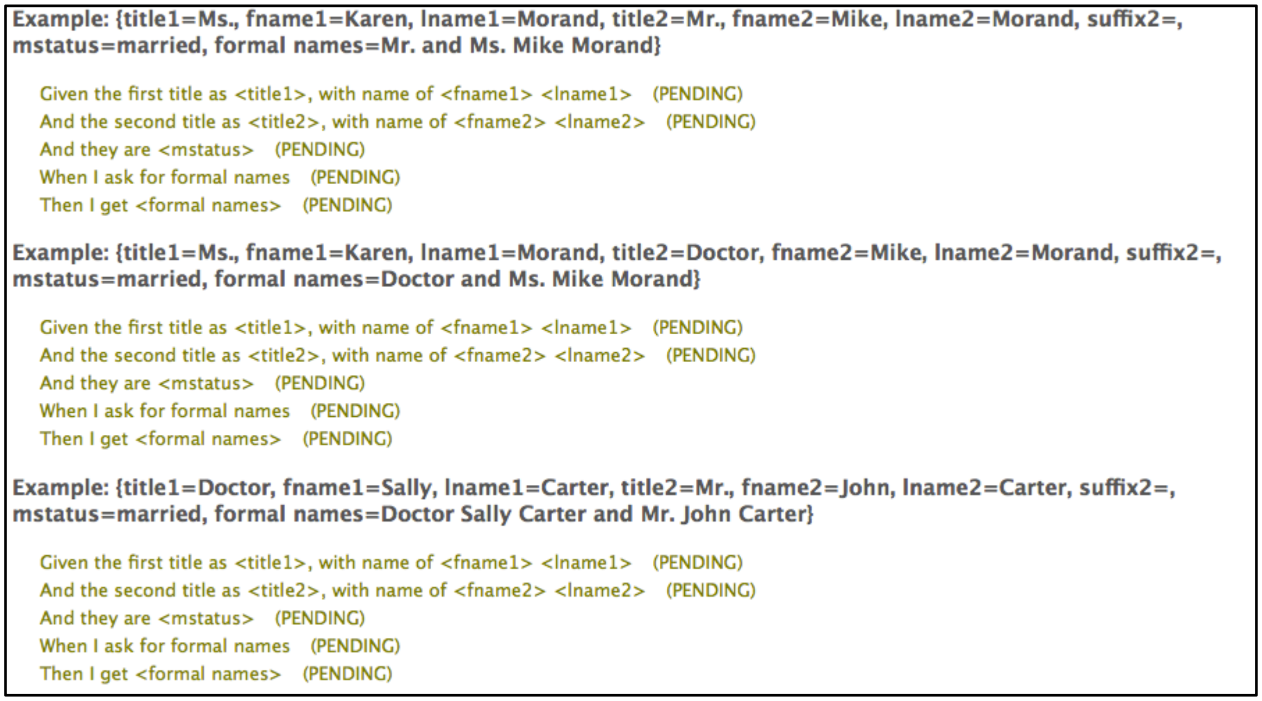
```
Feature: Formal Names for Wedding Labels

Scenario: Formal Variations
Given the first title as <title1>, with name of <fname1> <lname1>
And the second title as <title2>, with name of <fname2> <lname2>
And they are <mstatus>
When I ask for formal names
Then I get <formal names>

Examples:
|title1       |fname1|lname1 |title2    |fname2|lname2 |suffix2  |mstatus|formal names
|Ms.          |Karen |Morand |Mr.       |Mike  |Morand |         |married|Mr. and Ms. Mike Mo
|Ms.          |Karen |Morand |Doctor    |Mike  |Morand |         |married|Doctor and Ms. Mike
|Doctor       |Sally |Carter |Mr.       |John  |Carter |         |married|Doctor Sally Carter
|Doctor       |Sally |Carter |Doctor    |John  |Carter |         |married|The Doctors Carter
|The Honorable|Pamela|Patel  |Lieutenant|James |Patel  |U.S. Navy|married|The Honorable Pamel
|Ms.          |Karen |Walsh  |Mr.       |Mike  |Morand |         |married|Ms. Karen Walsh and
|Ms.          |Karen |Walsh  |Mr.       |Mike  |Morand |         |single |Mr. Mike Morand \nM
```

Figure 17: JBehave requirements code

A downside of writing all the examples in one table is that the English descriptions of each scenario are lost (the text that said, for example: *Scenario: Married couple with the same last name where woman is a doctor*). This could potentially be put back in as an extra column in the table, or as a comment after the table, if it was decided the scenario was not readable.

JBehave Reports

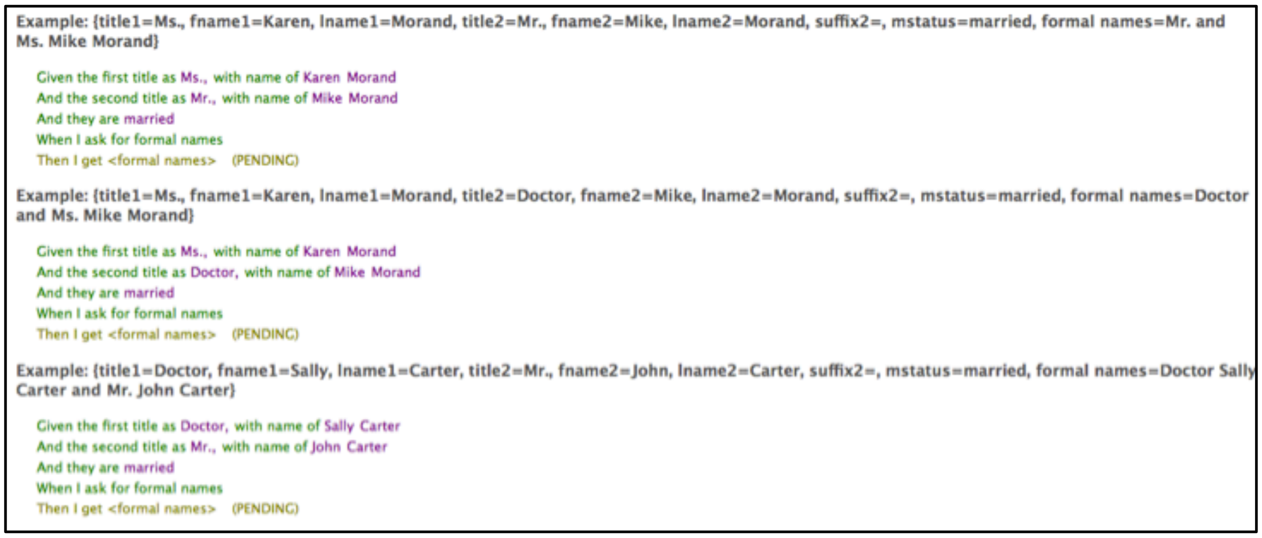
Like Cucumber, JBehave produces an HTML report each time it is run. The report begins with a nicely formatted table describing the tests to run. Then it details each test along with the results. When the developer ran it before writing any code, the following results were produced.



```
Example: {title1=Ms., fname1=Karen, lname1=Morand, title2=Mr., fname2=Mike, lname2=Morand, suffix2=,
mstatus=married, formal names=Mr. and Ms. Mike Morand}

    Given the first title as <title1>, with name of <fname1> <lname1>   (PENDING)
    And the second title as <title2>, with name of <fname2> <lname2>   (PENDING)
    And they are <mstatus>   (PENDING)
    When I ask for formal names   (PENDING)
    Then I get <formal names>   (PENDING)

Example: {title1=Ms., fname1=Karen, lname1=Morand, title2=Doctor, fname2=Mike, lname2=Morand, suffix2=,
mstatus=married, formal names=Doctor and Ms. Mike Morand}

    Given the first title as <title1>, with name of <fname1> <lname1>   (PENDING)
    And the second title as <title2>, with name of <fname2> <lname2>   (PENDING)
    And they are <mstatus>   (PENDING)
    When I ask for formal names   (PENDING)
    Then I get <formal names>   (PENDING)

Example: {title1=Doctor, fname1=Sally, lname1=Carter, title2=Mr., fname2=John, lname2=Carter, suffix2=,
mstatus=married, formal names=Doctor Sally Carter and Mr. John Carter}

    Given the first title as <title1>, with name of <fname1> <lname1>   (PENDING)
    And the second title as <title2>, with name of <fname2> <lname2>   (PENDING)
    And they are <mstatus>   (PENDING)
    When I ask for formal names   (PENDING)
    Then I get <formal names>   (PENDING)
```

Figure 18: JBehave results when no code is written

Here each example fills in the parameterized scenario with one row of data. It is also clear how JBehave displays unimplemented code. Unlike Cucumber, there is no “skipped” status; it just marks every line of every test as “PENDING”. The next thing done was to implement the framework code, but mark the actual application code as “pending”. This gave the following result.



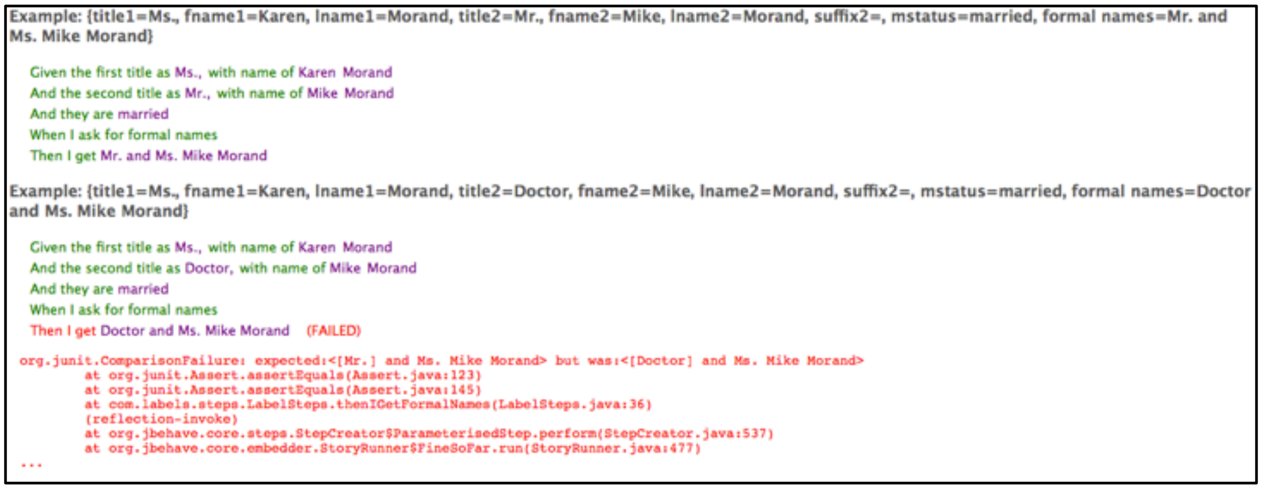
```
Example: {title1=Ms., fname1=Karen, lname1=Morand, title2=Mr., fname2=Mike, lname2=Morand, suffix2=, mstatus=married, formal names=Mr. and
Ms. Mike Morand}

  Given the first title as Ms., with name of Karen Morand
  And the second title as Mr., with name of Mike Morand
  And they are married
  When I ask for formal names
  Then I get <formal names>   (PENDING)

Example: {title1=Ms., fname1=Karen, lname1=Morand, title2=Doctor, fname2=Mike, lname2=Morand, suffix2=, mstatus=married, formal names=Doctor
and Ms. Mike Morand}

  Given the first title as Ms., with name of Karen Morand
  And the second title as Doctor, with name of Mike Morand
  And they are married
  When I ask for formal names
  Then I get <formal names>   (PENDING)

Example: {title1=Doctor, fname1=Sally, lname1=Carter, title2=Mr., fname2=John, lname2=Carter, suffix2=, mstatus=married, formal names=Doctor Sally
Carter and Mr. John Carter}

  Given the first title as Doctor, with name of Sally Carter
  And the second title as Mr., with name of John Carter
  And they are married
  When I ask for formal names
  Then I get <formal names>   (PENDING)
```

Figure 19: JBehave results with pending tests

The text of the scenario is colored green, while the variables are colored purple. The actual “then” statements are all marked “PENDING”, because the implementation was marked as pending. There is no distinction between “unimplemented” and “pending” code.

For the final step, like in the other two frameworks, the developer wrote just enough code to make the first test pass, leaving the other tests failing, to mimic a “broken” or “bug introduced” state. The result in Figure 20 was produced.

The first test is all green and purple, with no errors. The second test has red text, and the text “FAILED” next to it. JBehave also included information about the difference between expected results and the actual results, and it included a Java stack trace pointing to the place where the error occurred.



```
Example: {title1=Ms., fname1=Karen, lname1=Morand, title2=Mr., fname2=Mike, lname2=Morand, suffix2=, mstatus=married, formal names=Mr. and Ms. Mike Morand}

  Given the first title as Ms., with name of Karen Morand
  And the second title as Mr., with name of Mike Morand
  And they are married
  When I ask for formal names
  Then I get Mr. and Ms. Mike Morand

Example: {title1=Ms., fname1=Karen, lname1=Morand, title2=Doctor, fname2=Mike, lname2=Morand, suffix2=, mstatus=married, formal names=Doctor and Ms. Mike Morand}

  Given the first title as Ms., with name of Karen Morand
  And the second title as Doctor, with name of Mike Morand
  And they are married
  When I ask for formal names
  Then I get Doctor and Ms. Mike Morand   (FAILED)

 org.junit.ComparisonFailure: expected:<[Mr.] and Ms. Mike Morand> but was:<[Doctor] and Ms. Mike Morand>
	at org.junit.Assert.assertEquals(Assert.java:123)
	at org.junit.Assert.assertEquals(Assert.java:145)
	at com.labels.steps.LabelSteps.thenIGetFormalNames(LabelSteps.java:36)
	(reflection-invoke)
	at org.jbehave.core.steps.StepCreator$ParameterisedStep.perform(StepCreator.java:537)
	at org.jbehave.core.embedder.StoryRunner$FineSoFar.run(StoryRunner.java:477)
 ...
```

Figure 20: JBehave results with failing test

JBehave Code

The code for JBehave is written in Java, and like Cucumber, it uses annotations to find the correct code to run. Although JBehave does provide sample code when it cannot find any code that matches, none of the code it provided matched the needed code for the parameterized scenario. It was written by hand. The Java implementation can be found in Appendix C.

Chapter 4 –Project Analysis and Results

In order to compare and contrast the three BDD frameworks, it was necessary to start with a set of criteria with which to judge them. As noted in the Literature Review, the same reasons that cause new frameworks to be written can also be used for judging existing frameworks.

Criteria for Data Analysis

As a reminder, the criteria selected are:

- *Usability* of the framework – How easy is it to find and work with the Story Tests? Can the reports be run ad-hoc, by a non-technical person?
- *Readability* of the specifications - Can a non-technical person understand what the specifications are telling them? Can a non-technical person write one, or at least modify an existing Story Test?
- *Understandability* of the results - How easy is it to tell if the requirements have been met? How easy is it to tell what went wrong? Do the results provide just pass/fail, or do they have more states, like "pending"?
- *Develop-ability* - How easy is it to develop the fixtures/hooks to the application itself? How easy is it to back up the Story Tests?
- *Expandability* - How easy is it to add a new requirement? How easy is it to change an existing one?

This chapter will discuss each framework implementation in detail, paying close attention to how well it achieved each of the criteria listed above. After a discussion of each framework, the framework will be rated according to each criterion on a scale of 1-5 (where “5” means that the framework fully met the criterion).

The ratings themselves will be fairly subjective, since they will represent the opinion of just one researcher. They are provided primarily as a means of quickly visualizing the results. More important than the ratings are the discoveries made through exploration and evaluation of the same problem in each framework tool. These discoveries will be discussed in more detail in Chapter Five.

Cucumber

The analysis will begin with Cucumber, because it is the most newly developed of the three frameworks being reviewed.

Technical Evaluation

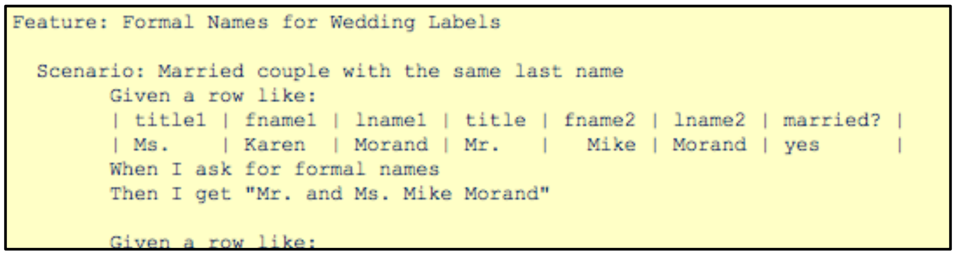
The Cucumber JVM framework was easy to download and install. It was well-documented, and it provided helpful code snippets when needed. All of the Story Tests and fixtures live in the same code-base as the application code itself. This means that they can be integrated with the application code’s source control and backup processes.

A downside of having the Story Tests live with the code is that they are not necessarily very accessible by non-technical people. Somebody who wanted to change or add a Story Test would need to use the team’s source control tool to download the latest tests, and to commit the changes to the code repository. Source control tools are often not very user-friendly.

Although not implemented in this study, Cucumber integrates well with a variety of Continuous Integration environments (“Cucumber Continuous Integration,” n.d.). This means that a development team can automate the report generation at any interval desired.

Business Evaluation

As far as reading and writing the requirements themselves, the tabular presentation of data was challenging to work in. The wiki markup approach, using a pipe (|) symbol to separate table columns, made it easy to write but hard to read. If one added spaces to make all of the pipes line up vertically, like in Figure 21**,** it was easier to read but hard to maintain. Changing a piece of data meant going through the whole table and reformatting the spaces.

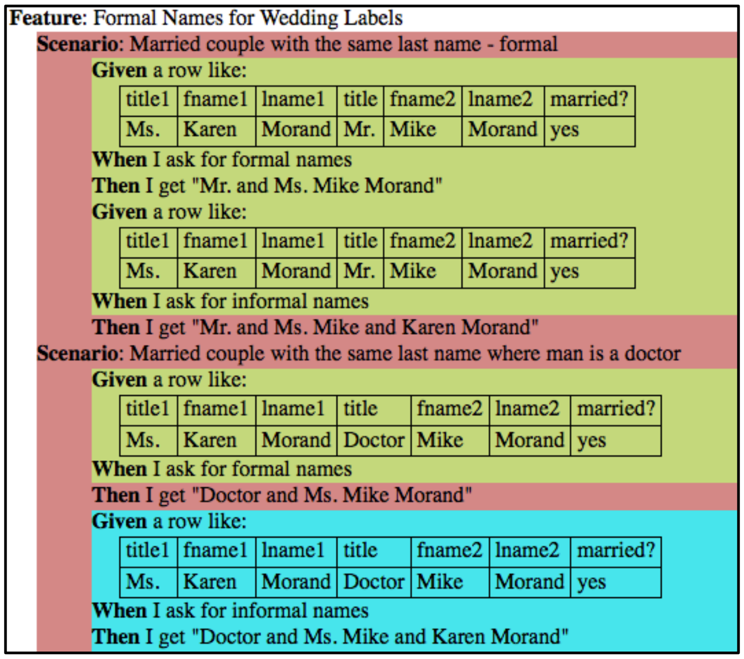


```
Feature: Formal Names for Wedding Labels

  Scenario: Married couple with the same last name
        Given a row like:
        | title1 | fname1 | lname1 | title | fname2 | lname2 | married? |
        | Ms.    | Karen  | Morand | Mr.   |   Mike | Morand | yes      |
        When I ask for formal names
        Then I get "Mr. and Ms. Mike Morand"

        Given a row like:
```

Figure 21: Example Cucumber Code

On the other hand, aside from the readability of the specifications, it was not difficult to *add* a new requirement. The data was easy to set up, and the expectations were easy to write. Making the pipes line up was not difficult when adding a new requirement.

Regarding the reports, an example of which can be seen in Figure 22: one downside of Cucumber JVM is that the reports are unattractive "out of the box," and not very readable. However, the results can be made more attractive with some CSS help. The example seen in Figure 22, like the examples in the Methodology chapter, uses a style-sheet downloaded from the Internet, making it a bit more readable. A proper web designer could make them even better, making the framework more appealing.


Feature: Formal Names for Wedding Labels
Scenario: Married couple with the same last name - formal
Given a row like:

title1	fname1	lname1	title	fname2	lname2	married?
Ms.	Karen	Morand	Mr.	Mike	Morand	yes

When I ask for formal names
Then I get "Mr. and Ms. Mike Morand"
Given a row like:

title1	fname1	lname1	title	fname2	lname2	married?
Ms.	Karen	Morand	Mr.	Mike	Morand	yes

When I ask for informal names
Then I get "Mr. and Ms. Mike and Karen Morand"
Scenario: Married couple with the same last name where man is a doctor
Given a row like:

title1	fname1	lname1	title	fname2	lname2	married?
Ms.	Karen	Morand	Doctor	Mike	Morand	yes

When I ask for formal names
Then I get "Doctor and Ms. Mike Morand"
Given a row like:

title1	fname1	lname1	title	fname2	lname2	married?
Ms.	Karen	Morand	Doctor	Mike	Morand	yes

When I ask for informal names
Then I get "Doctor and Ms. Mike and Karen Morand"

Figure 22: Example Cucumber Report

A more frustrating problem is that when a test fails, the report indicates a failure but does not say why. A person reading the reports might want more details when a test has failed. For example, did it fail due to a null pointer exception, or due to the word "Doctor" showing up when it should show the guest's first name? Those are two very different situations, and the framework does not provide a quick way of finding out which was the case.

Based on the above discussion, this research assigned Cucumber the following ratings:

Table 1: Cucumber metrics

Criteria	Cucumber
Develop-ability	5
Usability of framework	4
Readability of specifications	2
Understandability of results	3
Expandability	4

FitNesse

FitNesse is quite different from the other two frameworks, because it is written, maintained, and run all in the same wiki page or pages. This comes with both advantages and challenges.

Technical Evaluation

From a development perspective, FitNesse was quite easy to set up and write fixtures for. However having all of the specifications live inside an entire wiki does create some maintenance problems with regard to source control, version management, and backups. The FitNesse wiki has a rudimentary versioning system, but it is also easy to accidentally lose all of your changes with a few accidental keystrokes. Trying to check the wiki pages in and out of source control defeats the advantages of having them in a wiki to begin with.

While the tests are easy to run interactively, because that is what they were designed to do, FitNesse does create some challenges running the tests in an automated fashion. However as noted in the "How to Use a Story Testing Framework" section of Chapter Two, running the tests regularly is an essential part of the feedback loop. FitNesse runs on its own web server, which means that the Continuous Integration machine needs to launch the tests remotely, then collect the results and interpret them. All of this is feasible, but it is not straightforward.

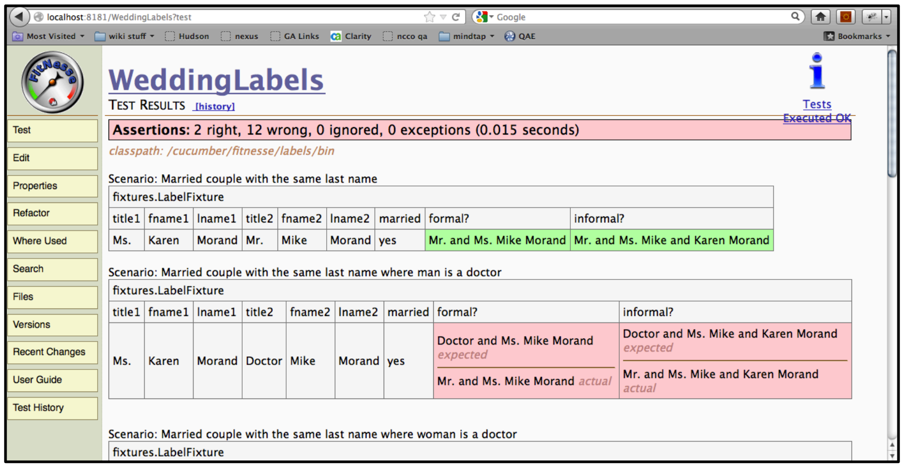
Business Evaluation

The advantage of running tests in a wiki is that multiple users can access and run the tests at the same time. Multiple users can also edit test pages and add new ones, all at the same time.

The readability and flexibility of FitNesse is a mix. FitNesse uses wiki markup and pipes to create tables, just like Cucumber. This makes each test not very readable. However since the user writes the tests within a wiki itself, there are some advantages. For one thing, the wiki editor has a “format” button on the bottom of the screen, which will automatically adjust all of the whitespace in the editor so that the pipes in the tables will line up vertically. This is a big time-saver. Also when the page is saved, the user see can see the table properly formatted right away, so formatting mistakes are immediately apparent. Being able to immediately view the formatted table takes away a good deal of the struggle with readability.

However, when in the FitNesse editor, the tables are still hard to read. This makes them hard to change, especially if a user is looking for one particular piece of data. Being able to format the pipes and whitespace helps, but it is still a challenge to make changes.

One big advantage to writing tests in a wiki is immediate feedback. If a user adds a new test to an already-built framework, she can save it and instantly find out if it passes or fails. This is very handy when requirements expand or change.



WeddingLabels

TEST RESULTS [history]

Assertions: 2 right, 12 wrong, 0 ignored, 0 exceptions (0.015 seconds)

classpath: /cucumber/fitnesse/labels/bin

Scenario: Married couple with the same last name

fixtures.LabelFixture								
title1	fname1	lname1	title2	fname2	lname2	married	formal?	informal?
Ms.	Karen	Morand	Mr.	Mike	Morand	yes	Mr. and Ms. Mike Morand	Mr. and Ms. Mike and Karen Morand

Scenario: Married couple with the same last name where man is a doctor

fixtures.LabelFixture								
title1	fname1	lname1	title2	fname2	lname2	married	formal?	informal?
Ms.	Karen	Morand	Doctor	Mike	Morand	yes	Doctor and Ms. Mike Morand *expected* / Mr. and Ms. Mike Morand *actual*	Doctor and Ms. Mike and Karen Morand *expected* / Mr. and Ms. Mike and Karen Morand *actual*

Scenario: Married couple with the same last name where woman is a doctor

fixtures.LabelFixture

Figure 23: Example FitNesse Report

The results page for FitNesse is, as noted earlier, the same wiki page. A reminder of that wiki page can be seen in Figure 23. This is useful in some ways, because the user should already be familiar with the way the specifications look, and the way they work. However it does mean that the user does not get any extra formatting or special presentation of the test results. Also FitNesse does not offer a way of indicating that a test is "pending," to indicate work not yet in development – it can only be passing or failing. The red/green highlighting works, and the results are understandable. It is also helpful that the user is presented with both "expected" and "actual" results.

Based on the above discussion, this research assigned FitNesse the following ratings:

Table 2: FitNesse Metrics

Criteria	FitNesse
Develop-ability	3
Usability of framework	4
Readability of specifications	4
Understandability of results	2
Expandability	4

JBehave

Technical Evaluation

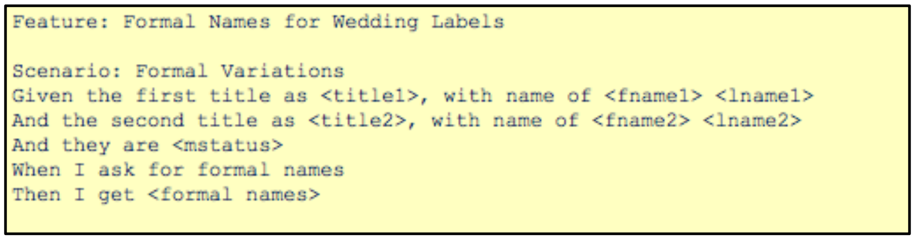
Surprisingly JBehave proved the most frustrating of the three frameworks to set up. Given its popularity and high usage, the difficulty getting it downloaded, installed, and running was unexpected. There is documentation, but it is example-driven instead of just providing steps and instructions. It also turned out that certain key components, such as the default style-sheet, were missing and had to be retrieved separately. All of these issues were "one-time" issues though, not the kind of issues a developer would be dealing with on a daily basis. Once all the pieces were put together, the actual coding of the fixtures was fairly straightforward, but there were many frustrations along the way.

Just like Cucumber, all of the Story Tests and fixtures live in the same code-base as the application code itself. This means that they can be integrated with the application code's source control and backup processes, which is a positive. A negative is that, just as in with Cucumber, they are not necessarily very accessible by non-technical people. Somebody who wanted to change or add a Story Test would need to use the team's source control tool to download the latest tests, and to commit the changes to the code repository. Source control tools are often not very user-friendly.

Another positive feature is that, although not implemented in this study, JBehave uses a build tool called Maven and therefore integrates well with a variety of Continuous Integration environments (Jaspers, 2011). This means that a development team can automate the report generation at any interval desired.

Business Evaluation

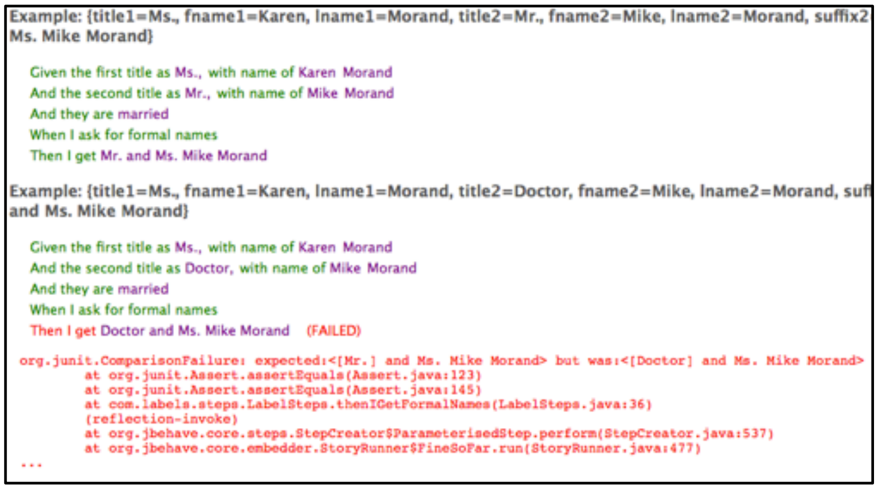
Once the installation hurdles were overcome, JBehave proved to have some nice features. The Parameterized Scenario approach, reproduced in Figure 24, greatly helped with making each test easier to read, and making each test easier for another reader to understand. The only downside is that, as noted in the previous chapter, the scenario descriptions were lost with this approach. However they could be added back in as part of the data table itself.



```
Feature: Formal Names for Wedding Labels

Scenario: Formal Variations
Given the first title as <title1>, with name of <fname1> <lname1>
And the second title as <title2>, with name of <fname2> <lname2>
And they are <mstatus>
When I ask for formal names
Then I get <formal names>
```

Figure 24: Example Parameterized Scenario

JBehave still used wiki markup and pipes to create data tables, just like the other frameworks. However, it had the benefit that the actual scenario was in a readable parameterized form. At least it was only the example data that had to be entered as in table format. The examples still had to be put into tables with pipes, but at least the examples are separated from the scenario itself. The flexibility still suffers though, due to the challenge of re-formatting pipes and whitespace whenever a piece of data needs to be changed.



```
Example: {title1=Ms., fname1=Karen, lname1=Morand, title2=Mr., fname2=Mike, lname2=Morand, suffix2
Ms. Mike Morand}

   Given the first title as Ms., with name of Karen Morand
   And the second title as Mr., with name of Mike Morand
   And they are married
   When I ask for formal names
   Then I get Mr. and Ms. Mike Morand

Example: {title1=Ms., fname1=Karen, lname1=Morand, title2=Doctor, fname2=Mike, lname2=Morand, suf
and Ms. Mike Morand}

   Given the first title as Ms., with name of Karen Morand
   And the second title as Doctor, with name of Mike Morand
   And they are married
   When I ask for formal names
   Then I get Doctor and Ms. Mike Morand    (FAILED)

 org.junit.ComparisonFailure: expected:<[Mr.] and Ms. Mike Morand> but was:<[Doctor] and Ms. Mike Morand>
        at org.junit.Assert.assertEquals(Assert.java:123)
        at org.junit.Assert.assertEquals(Assert.java:145)
        at com.labels.steps.LabelSteps.thenIGetFormalNames(LabelSteps.java:36)
        (reflection-invoke)
        at org.jbehave.core.steps.StepCreator$ParameterisedStep.perform(StepCreator.java:537)
        at org.jbehave.core.embedder.StoryRunner$FineSoFar.run(StoryRunner.java:477)
 ...
```

Figure 25: Example JBehave Report

JBehave reports, as noted above in the technical evaluation, needed styling in order to make them readable. However this was not an insurmountable hurdle, and once a style-sheet was found, they proved to be clean and fairly easy to understand. A reminder can be seen in Figure 25. The colored formatting was helpful, pending tests were marked differently than failing tests, and a failing test provided some information as to why it failed. The information (a stack trace) was probably more technical than it should be, but it was still nice to see what the problem was.

Based on the above discussion, this research assigned JBehave the following ratings:

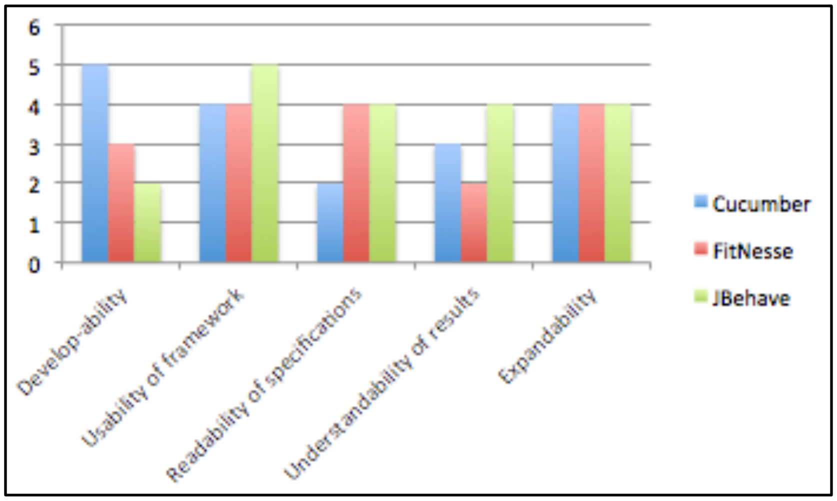
Table 3: JBehave Metrics

Criteria	JBehave
Develop-ability	2
Usability of framework	5
Readability of specifications	4
Understandability of results	4
Expandability	4

Comparisons

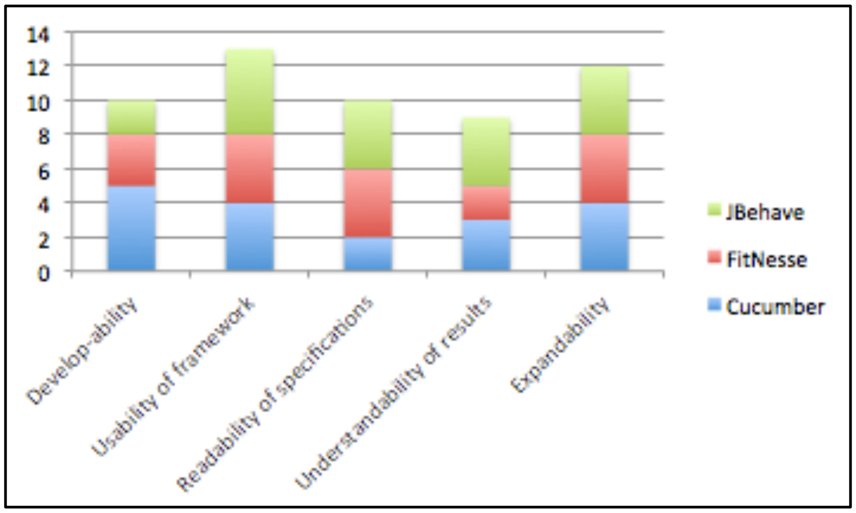
Plugging all of the ratings into a single table (Table 4), we see how each framework has been rated for each criterion. It is interesting to note that Develop-ability was widely distributed, while Expandability was identical for each framework. All frameworks had high Usability marks. Readability and Understandability were both fairly low and scattered.

Table 4: Comparing frameworks by criteria



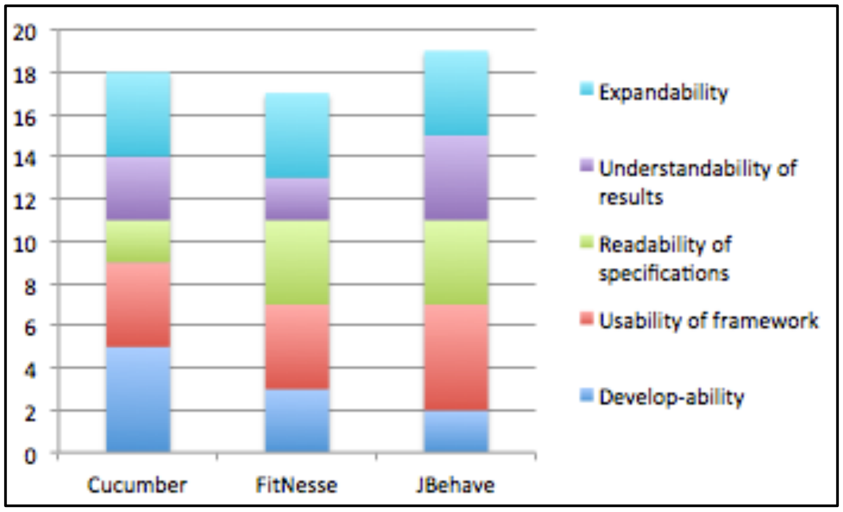
Another way to visualize these comparisons is by stacking the results from each criterion on top of one another. The chart in Table 5 shows what criteria were most successfully achieved. The top criterion is Usability, with Expandability shortly behind. The criterion which the frameworks achieved the least is Understandability.

Table 5: Comparing criteria against each other



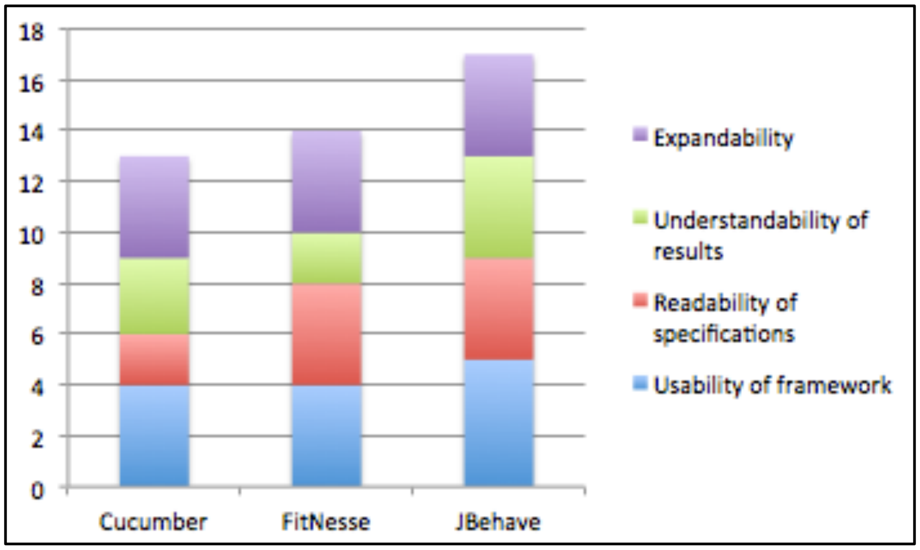
Yet a third way to visualize these comparisons is by stacking the results from each framework on top of one another. Table 6 shows each framework's total score. At first glance, they look fairly even, with JBehave slightly ahead.

Table 6: Comparing Frameworks



However Story Testing is supposed to be a tool primarily for the non-technical members of a team. Thus Table 7 removes the "develop-ability" factor. Here we see Cucumber fall significantly behind, while JBehave rises to the top.

Table 7: Comparing Frameworks from a User Perspective



Final Observations

One interesting thing to note is that Cucumber supports Parameterized Scenarios, just like JBehave (Lawrence, 2010). Looking at this last chart, we see that Cucumber's biggest downfall was Readability of Specifications. Parameterized Tables make tests much more readable, so had the Cucumber implementation for this research tried them as well, it probably would have been a much closer race between Cucumber and JBehave.

On the other hand, the charts as depicted above *actually* show the difference between three different approaches to writing Story Tests with tabular data:

1. Given/When/Then with tables (Cucumber)
2. Given/When/Then with Parameterized Tables (JBehave)
3. Decision Tables (FitNesse)

This is valuable information to have. Given that the two criteria that performed the worst were Readability of Specifications, and Understandability of Results, it will be important to identify what techniques are out there to make specifications and results more readable.

Summary

In the end, although the implementations were all different, the frameworks had more similarities than differences. The three frameworks all scored lower on understandability and readability of the specifications, due largely to difficulties with representing tabular data. Each implementation used a different approach, but no approach was both easily readable *and* easily maintainable. However, each of the frameworks achieved the primary goals of Story Testing described in Chapter Two. Each captured requirements in some form of natural language. Each could be run regularly. Although only one was easily run ad hoc, all could be run in an automated fashion.

Most importantly, each of the frameworks produced reports that could be used to describe the status of the project. With a clear indication of “passing” or “failing” for every project requirement, each framework, used properly, could be used to track and manage changes to project requirements.

Chapter 5 – Discussion

As noted in Chapter One, software projects often fail to achieve their goals within the targeted timeframe and budget. Some of this can be attributed to a disconnect between business requirements and the final product. Some of it can also be attributed to business requirements that change during software development, causing a cycle of rework, bugs, and missed requirements. However in order to stay competitive in today's dynamic market, businesses have to be able to evolve rapidly. If a company is making software to be used by Widget A, but suddenly massively popular Widget B comes on the market, the software had better support Widget B too, or its customers will go elsewhere.

As long as business requirements are written in unstructured documents, and software is written in executable code, with nothing connecting the two, software projects will continue to struggle to meet the needs of the business in a timely fashion. Story Testing is an effort to connect the business requirements directly to the developed application with executable code. Executable code can be verified, proving that the software meets the requirements. This verification means that even when requirements change, the development team will have a very clear definition of when each requirement is complete. Thus the Story Tests aim to provide a guarantee that the completed application matches the business need.

The practice of Story Testing has the potential to bridge the gap between business requirements and completed software. Although there have been many efforts to do this via a variety of frameworks, the practice is not widely adopted. This research paper sought to evaluate the usability of Story Testing frameworks, with a goal of addressing some part of *why* the practice is not more widely adopted. It addressed this problem by implementing three current Story Testing frameworks to analyze their utility and effectiveness.

To compare the frameworks, this research began by selecting a small business application with a complicated set of business rules. It then wrote requirements for the same application in three frameworks: Cucumber, FitNesse, and JBehave. Along with the requirements, enough code was written in each framework to be able to run the framework's reports and compare the results in different states: not started, pending, failing, and succeeding.

The completed artifacts were compared using five criteria: usability of the framework, readability of the specifications, understandability of the results, develop-ability of the framework, and expandability of the specifications. Each implementation of the frameworks achieved the primary goals of Story Testing described in Chapter Two. Each framework captured requirements in some form of natural language. Each could be run regularly, in an automated fashion. Most importantly, each of the frameworks produced reports that could be used to describe the status of the project. With a clear indication of "passing" or "failing" for every project requirement, each framework, used properly, could track and manage changes to project requirements.

Each of the frameworks struggled with making specifications that were easy to both read and modify, to one degree or another. In particular, the challenge was setting up large amounts of input data. Putting inputs together field-by-field would be very wordy and hard to read, but wiki tables, while easier to read, are not easy to maintain.

Conclusions

The Design Science approach to this research constructed many artifacts, as seen in Chapter Three. The results from these artifacts proved difficult to quantify, but this is not surprising. The point of Design Science is not to quantify, but to explore solution alternatives. The exploration and understanding provided by the construction of three different frameworks is more valuable than the charts produced at the end of Chapter Four. Thus the conclusions drawn from this research will center around not just those charts, but on both the data and the lessons learned while building the artifacts.

Four primary conclusions are drawn from the experience of building the artifacts described in this project.

1. Story Testing can help prove a project's completeness

Every framework implemented was able to define the requirements more clearly and more rigorously than the original requirements pulled from the Martha Stewart Weddings website. Writing the story tests helped find and patch gaps in the original requirements, making them more complete. Each framework's reports clearly showed which requirements were passing, and which were not, thus showing how close the application was to fulfilling the requirements.

This allows us to conclude that a Story Testing framework can provide a set of executable specifications that will be able to provide a "definition of done" that free-form paragraphs will never offer. Furthermore, Story Test specifications allow changed requirements to be recorded and tracked. If a requirement changes in a Story Test framework, the framework reports will verify that the change has made it into the application itself. This helps business people, developers, and testers know if changed requirements have correctly made it into the code.

A tool that *programmatically verifies* that a changed requirement is handled successfully will be a big step toward reducing requirements duplication. Reducing requirements duplication will then reduce missed requirements.

2. Specifications are still too technical

On the downside, the Story Tests created showed that natural language specifications are not easy to read or maintain, especially when there are enough variables that they require tabular data entry. This is unfortunate, since any application that is complicated enough to find a Story Testing framework useful is going to have many moving variables. Each of the three of frameworks scored low on readability, due largely to the difficulty of using pipes to build tables.

The natural language approach that scored the highest, and therefore showed the most promise, was Given/When/Then with Parameterized Tables. This is worth exploring further, but even it presented some challenges. Although the primary goal of Story Testing is to present executable specifications in a format that can be easily read and written by a non-technical person, none of the tests created for this research really succeeded. This is likely to be the biggest hurdle to their adoption.

3. Implementation is not overly complex

One might assume that the installation, development, and maintenance of a new framework could be too large of a technical hurdle. However this research indicates that this may not be too much of an obstacle in adopting any of the frameworks. Although a detailed study on the setup and implementation of the frameworks was beyond the scope of this research paper, and certainly each required some amount of development effort to install, set up, and create fixtures, they all proved manageable in implementing the artifacts. Even JBehave, which was the most work technically, would be easy to work in once the initial setup was completed.

4. Story testing is worth it

The final conclusion drawn from this research sums up the first three: even though it might not be possible to completely replace written software requirements with executable ones, this research shows that it is still worth writing executable specifications. Story Tests, with their executable specifications, provide a safety net to capture bugs and help manage requirements. This is particularly true if requirements are likely to change during development, and it is also particularly true in applications with complex business rules that are unrelated to the user interface, like was the case in the sample application.

When a business requirement can be written as a Story Test, it can be verified directly against the application code, in an automated fashion. This verification means that even when requirements change, the development team will have a very clear definition of when and whether each requirement is complete. This is how software requirements as executable code provide a guarantee that the completed application matches the business need.

Summary of Contributions

While there are a lot of papers written on various flavors of Story Testing, there are no formal comparisons between frameworks. This research fills that gap. Implementing the same problem three different ways provided a formal comparison. It also shows the different questions one might ask in choosing a framework, and what kind of considerations one should take into account. It establishes how to compare frameworks. It shows how we can use different frameworks to prove a project's completeness.

Unfortunately this research also clearly demonstrated the challenges of implementing executable requirements in natural language. Complex input data is difficult to represent in a natural language. While a tabular approach seems to be the best option, and at first glance a wiki-style of markup seems like a logical approach, it is difficult to work with wiki-style tables in an ordinary text editor. The research demonstrated several options, but it also indicates that this is an area with significant room for improvement.

The artifacts also demonstrate the flow from “business-style” requirements to several kinds of “Story-Test-style” requirements. They provide practical information on how a development team new to Story Testing might implement it in a project. There is a lot of information available to developers on *why* to use Behavior Driven Development, or Acceptance Test Driven Development, but not as much about the *process* of how one would implement one of these Story Testing frameworks for a set of application requirements.

In a similar vein, this research also demonstrated the “how to” for each framework, and the differences and similarities between them.

Lessons Learned

One of the challenges in building the artifacts was the need to switch contexts. On beginning work on the first framework, it was all new, and therefore relatively straightforward. On beginning the second framework, there was a desire to switch regularly back to the first set of code, comparing and contrasting. This meant it was challenging to look at the second framework in a fresh way. By the time the third framework was started, the tests were definitely being written in relation to the first two frameworks, as opposed to just building it from scratch. This kept the later frameworks from being approached with the same “fresh and new” mindset that the first framework received.

If beginning this or a similar research project, it would be wise to delegate the installation and exploration of each framework to a different person, while the researcher directed and coordinated the efforts. That way there would be less influence between the implementations, and less confusion between the different sets of code. Another possibility would be to maintain just a single developer, but separate each implementation effort by several weeks of doing something else entirely, in order to approach each framework with a fresh perspective.

Another issue that presented challenges in two of the frameworks was CSS styling of the results pages. Although that was not directly related to the study, it was distracting when the results did not look good. This led to a lot of time looking for style-sheets, time that would have been better spent enhancing other parts of the implementation. The frameworks themselves should consider providing default style implementations that are clean and easy to find. Then a developer only needs to do styling if they want something different than the default.

Recommendations for Future Research

This research project consisted of constructive research aimed at learning more about three Story Testing frameworks. As such, the conclusions drawn were only the experiences and opinions of one researcher. A very interesting qualitative project would be to take the artifacts built here and gather other opinions of the successes and failures of each framework. This could be done via interviews of business people and developers who might use a Story Testing framework. It could also be done by surveying many individuals in a variety of roles in software development.

One of the biggest areas for further research is more in-depth investigation into ways to depict complex requirements in natural language. This research project indicates that none of the three more well-known Story Testing frameworks succeed in doing this. Is there a research project that could find a more useful way of presenting requirements in a non-technical language, which can still be executed by a computer? During this research, there was only one study found exploring the use of executable acceptance tests from a customer perspective (Melnik, Maurer, & Chiasson, 2006), and a good deal more research could be done in this vein. This remains a significant hurdle to the adoption of Story Testing. The approach to tabular data that showed the most promise was the JBehave example, which used Given/When/Then with Parameterized Data Tables. This merits more exploration.

Another area for further research is to create similar artifacts for other Story Testing frameworks. This research only covered three frameworks. As we saw in the Literature Review, these are not the only ones out there. It would be useful to do the same parallel comparison with other frameworks, to see if they provide any additional benefits or ideas. While doing this, it would be wise to take note of the Lessons Learned provided above. Rather than having a single individual implement all of the frameworks, have each one implemented by one or two different individuals, but all implementing the same specifications. This will keep each framework's results from influencing any other framework's results.

Another effort that would enhance the current body of research would be to take this same comparison approach and actually *implement* the same application using each framework. This project only consisted of comparing the requirements. The actual implementations were stubbed out. It would be interesting to see each project evolve within each framework.

Another suggestion is: investigate ways to make the accessing and writing/changing of story tests more accessible to a non-technical user. Source control systems are not by their nature very friendly.

Recommendation for Improvement

This could be considered either a recommendation for future research, or it could be a recommendation for improvement aimed directly at a group developing a Story Test framework. Upon considering the above conclusions and recommendations for future research, two things stand out as "low-hanging fruit" that could easily make the use of Story Testing more appealing to a non-technical person. The first item is the difficulty of working with wiki-style table markup in a standard text editor. The second item is the difficulty working with traditional source control tools.

An application with a friendly User Interface could be developed, to work as a layer on top of an existing framework (Cucumber, for example). This UI could present a simple view into the application's tests under source control. When the user selects one, she could edit it in a simple WYSIWYG wiki-style editor that could edit and render wiki tables gracefully. On making changes and saving them, they could be checked directly into the project's source control, so the changes remained a living part of the project. That check-in should trigger an immediate build and run all existing Story Tests, so that the user can then verify the success of the newly entered requirements.

Summary

The original problem statement of this thesis began, "Story Testing has the potential to bridge the gap between business requirements and completed software. There have been many efforts to do this via a variety of frameworks, but none have been widely adopted." This problem statement can be broken down into two key phrases:

1. Story Testing could bridge the gap between requirements and software
2. Despite many efforts to do this, none have been widely adopted

This research paper confirmed the first statement with its first conclusion: *Story Testing can help prove a project's completeness*. The gap between requirements and software represents all the unknown differences between requirements as written, and the final software product as developed. Story Testing, with its ability to run executable requirements, allows us to track changing requirements, identify where they are not met, and implement them.

This research identified and cast doubt on one possible explanation for the second statement with its third conclusion: *Implementation is not overly complex.* One could easily hypothesize that teams are not adopting Story Tests because of implementation challenges. This research showed each of them to be fairly simple to implement.

Finally, this research did discover a strong possibility that could explain the second statement, with its second conclusion: *Specifications are still too technical*. We learned how to write executable requirements in three different frameworks. However we also learned that the writing of these executable requirements still needs some improvement, before they can be easily written and read by non-technical business people. We learned that software requirements as executable code are not likely to be widely adopted until that hurdle is overcome. Finally, we identified a recommendation for improvement of Story Testing frameworks, one that might increase their attractiveness to non-technical business representatives.

References

A Bit Better Corporation. (Producer). (n.d.). Screen beans clip art. [Web Graphic]. Retrieved from http://office.microsoft.com/en-us/images/results.aspx?qu=screen beans

Ambler, S. (2003). Examining the agile cost of change curve [Web log message]. Retrieved from http://www.agilemodeling.com/essays/costOfChange.htm

Ambler, S. (n.d.). Introduction to user stories. Retrieved from http://www.agilemodeling.com/artifacts/userStory.htm

Arlow, J, & Neustadt, I. (2005). Uml 2 and the unified process: practical object-oriented analysis and design. Addison-Wesley Professional.

Avram, A. (2008, August 13). Presentation: Martin Fowler and Dan North Talk Over the Yawning Crevasse of Doom [Web log message]. Retrieved from http://www.infoq.com/news/2008/08/Fowler-North-Crevasse-of-Doom

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A. et al. (2001) Manifesto for Agile Software Development. Retrieved on the 10th of March 2011 from http://agilemanifesto.org/

Beck, K. (2003). Test-driven development. Boston: Pearson Education.

Behat – BDD for PHP. (n.d.). Retrieved from http://behat.org/

Behat – Writing Features. (n.d.). Retrieved from http://docs.behat.org/guides/1.gherkin.html

Brookes, F. (1987). No Silver Bullet: Essence and Accidents of Software Engineering. Retrieved from http://www.virtualschool.edu/mon/SoftwareEngineering/BrooksNoSilverBullet.html.

Cucumber. (n.d.). Retrieved from http://cukes.info/

Cucumber Continuous Integration. (n.d.). Retrieved from https://github.com/cucumber/cucumber/wiki/Continuous-Integration

Cockburn, A. (1997). Structuring use cases with goals. Journal of Object-Oriented Programming, doi: HaT.Technical Report.1995.01

Cunningham, W. (2002, August). Acceptance testing as document authoring and annotation. Workshop: Agile Acceptance Testing Xp/agile universe, Chicago, IL. Retrieved from http://c2.com/doc/xpu02/workshop.html

Cunningham, W. (2005, Mar 1). Introduction to Fit. Framework for Integrated Test, Retrieved from http://fit.c2.com/wiki.cgi?IntroductionToFit

Cunningham, W. (2007, Oct 12). Old welcome visitors. Framework for Integrated Test, Retrieved from http://fit.c2.com/

Davis, W., & Yen, D. C. (1999). The information system consultant\'s handbook: Systems analysis and design. Oxford: CRC Press LLC.

De Wille, E. & Vede, D. (2008) Software Process Models. Retrieved on the 1st of May 2011 from http://www.the-software-experts.de/e_dta-sw-process.htm

Dinwiddie, G. (2010). A "Lingua Franca" to Ensure You Get the Right System [Web log message]. Retrieved from http://blog.gdinwiddie.com/wp-content/uploads/2010/02/LinguaFranca-Feb2010.pdf

EasyB makes it easy, man. (n.d.). Retrieved from http://www.easyb.org/

Evans, E. (2004). Domain-driven design, tackling complexity in the heart of software. Addison-Wesley Professional.

Eveleens, J. L. & Verhoef, C. (2010) "The Rise and Fall of the Chaos Report Figures," IEEE Software, pp. 30-36, January/February, 2010

Farley, D. (2007, August 8). Introduction to user stories. Retrieved from http://behaviour-driven.org/DomainDrivenDesign

Fitnesse history. (n.d.). Retrieved from http://fitnesse.org/FitNesseHistory

Fitnesse Decision Table. (n.d.) Retrieved from http://fitnesse.org/FitNesse.UserGuide.SliM.DecisionTable

Fowler, M. (2008, May 15). Domain Specific Language [Web log message]. Retrieved from http://martinfowler.com/bliki/DomainSpecificLanguage.html

Fowler, M. (2008, December 15). Business Readable DSL [Web log message]. Retrieved from http://martinfowler.com/bliki/BusinessReadableDSL.html

Gherkin. (n.d.). Retrieved from https://github.com/cucumber/cucumber/wiki/Gherkin

Glass, R. (1998). In the Beginning: Recollections of Software Pioneers. Los Alamitos: IEEE Computer Society.

Google trends. (n.d.). Retrieved from http://www.google.com/trends/

Hellesøy, A. (2011, June 20). Cucumber 1.0.0 [Web log message]. Retrieved from http://aslakhellesoy.com/post/6734058541/cucumber-one-oh

Hendrickson, E. (2008). Driving development with tests: ATDD and TDD. Proceedings of the STANZ 2008, http://testobsessed.com/wp-content/uploads/2011/04/atddexample.pdf

Hevner, A., March, S., Park, J., & Ram, S. (2004). Design science in information systems research. MIS Quarterly, 28(1), 75-105.

Holmstrom, J., Ketokivi, M., & Hameri, A. (2009). Bridging practice and theory: A design science approach. Decision Sciences, 40(1), 65-87.

Hubbard, B. (n.d.). Lean Learning: What is JAD RAD? Retrieved from http://bobsleanlearning.wordpress.com/reference/what-is-rad-jad/

Janicki, R. & Wassyng, A. (2003). On tabular expressions. In Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research (CASCON '03), Darlene A. Stewart (Ed.). IBM Press 92-106.

Jaspers, T. (2011, March 25). Automated acceptance testing using JBehave. [Web log message]. Retrieved from http://blog.codecentric.de/en/2011/03/automated-acceptance-testing-using-jbehave/

Jbehave. (n.d.). Retrieved from http://jbehave.org/

Jennerich, B. (1990). Joint Application Design: Business requirements analysis for successful re-engineering . Retrieved from http://www.bee.net/bluebird/jaddoc.htm

Khalili, M. (2011, December 25). Introduction to bddfy. Retrieved from http://www.mehdi-khalili.com/bddify-in-action/introduction

Laukkanen, P. (2006). Data-Driven and Keyword-Driven Test Automation Frameworks (Master's Thesis). Retrieved from http://eliga.fi/Thesis-Pekka-Laukkanen.pdf

Lawrence, R. (2010, January 4). [Web log message]. Retrieved from http://www.richardlawrence.info/2010/01/04/how-to-remove-duplication-in-cucumber-tests-using-scenario-outlines/

Lew, A. & Tamanaha, D. (1976). Decision table programming and reliability. In Proceedings of the 2nd international conference on Software engineering (ICSE '76). IEEE Computer Society Press, Los Alamitos, CA, USA, 345-349.

Martin, R. (n.d.). One minute description. Retrieved from http://fitnesse.org/FitNesse.UserGuide.OneMinuteDescription

Michalak, M., Laukkanen, P. (2008) "Robot Framework for Test Automation". Testing Experience, December 2008.

Melnik, G., Maurer, F., & Chiasson, M. (2006). Executable acceptance tests for communicating business requirements: customer perspective. In Agile Conference, 2006 (pp. 12-46).

North, D. (2006, March). Introducing BDD. Better Software, Retrieved from http://dannorth.net/introducing-bdd/

North, D. (2007, June 17). Introducing RBehave [Web log message]. Retrieved from http://dannorth.net/2007/06/17/introducing-rbehave/

Palermo, J. (2006, May). Guidelines for Test-Driven Development. MSDN Library. Available at: http://msdn.microsoft.com/en-us/library/aa730844%28v=vs.80%29.aspx

Peterson, D. (n.d.). Concordion faq. Retrieved from http://www.concordion.org/Questions.html

Randell, B. (1968). The 1968/69 NATO Software Engineering Reports. Retrieved on the 1st of May 2011 from http://homepages.cs.ncl.ac.uk/brian.randell/NATO/NATOReports/index.html

Robot framework. (n.d.). Retrieved from http://code.google.com/p/robotframework/

Royce, W. (1970, August). Managing the development of large software systems. Proceedings IEEE WESCON. Retrieved from http://leadinganswers.typepad.com/leading_answers/files/original_waterfall_paper_winston_royce.pdf

Rspec. (n.d.). Retrieved from http://rspec.info

Specflow - pragmatic BDD for .NET. (n.d.). Retrieved from http://specflow.org/home.aspx

Standish Group (1995). The CHAOS Report (1994). Report of the Standish Group. Available at: http://www.standishgroup.com/sample_research/chaos_1994_1.php

Stewart, M. (n.d.). Addressing and mailing your wedding invitations. Wedding Etiquette Adviser, Retrieved from http://www.marthastewartweddings.com/228650/addressing-and-mailing-your-wedding-invitations/@center/272440/wedding-etiquette-adviser

StoryQ. (n.d.). Retrieved from http://storyq.codeplex.com/

Weisfeld, M. (2004). Object-Oriented Thought Process, The (3rd Edition). Indianapolis, Ind: Sams Pub.

Williams, W. (2011, Spring). Givwenzen– behavior driven development for fitnesse. Retrieved from http://www.methodsandtools.com/tools/tools.php?givwenzen

Appendix A (Cucumber)

weddingLabel.feature

```
Feature: Formal Names for Wedding Labels

  Scenario: Married couple with the same last name - formal
      Given a row like:
      | title1 | fname1 | lname1 | title | fname2 | lname2 | married? |
      | Ms.    | Karen  | Morand | Mr.   |   Mike | Morand | yes      |
      When I ask for formal names
      Then I get "Mr. and Ms. Mike Morand"

      Given a row like:
      | title1 | fname1 | lname1 | title | fname2 | lname2 | married? |
      | Ms.    | Karen  | Morand | Mr.   |   Mike | Morand | yes      |
      When I ask for informal names
      Then I get "Mr. and Ms. Mike and Karen Morand"

  Scenario: Married couple with the same last name where man is a doctor
      Given a row like:
      | title1 | fname1 | lname1 | title | fname2 | lname2 | married? |
      | Ms.    | Karen  | Morand | Doctor|   Mike | Morand | yes      |
      When I ask for formal names
      Then I get "Doctor and Ms. Mike Morand"

      Given a row like:
      | title1 | fname1 | lname1 | title | fname2 | lname2 | married? |
      | Ms.    | Karen  | Morand | Doctor|   Mike | Morand | yes      |
      When I ask for informal names
      Then I get "Doctor and Ms. Mike and Karen Morand"

  Scenario: Married couple with the same last name where woman is a doctor
      Given a row like:
      | title1 | fname1 | lname1 | title | fname2 | lname2 | married? |
      | Doctor | Sally  | Carter | Mr    |   John | Carter | yes      |
      When I ask for formal names
      Then I get "Doctor Sally Carter and Mr. John Carter"

      Given a row like:
      | title1 | fname1 | lname1 | title | fname2 | lname2 | married? |
      | Doctor | Sally  | Carter | Mr    |   John | Carter | yes      |
      When I ask for informal names
      Then I get "Doctor Sally Carter and Mr. John Carter"

  Scenario: Married couple with the same last name where both are doctors
      Given a row like:
      | title1 | fname1 | lname1 | title | fname2 | lname2 | married? |
      | Doctor | Sally  | Carter | Doctor|   John | Carter | yes      |
      When I ask for formal names
      Then I get "The Doctors Carter"

      Given a row like:
      | title1 | fname1 | lname1 | title | fname2 | lname2 | married? |
      | Doctor | Sally  | Carter | Doctor|   John | Carter | yes      |
      When I ask for informal names
      Then I get "The Doctors Carter"

  Scenario: Married couple with the same last name and different professional titles
      Given a row like:
      | title1       | fname1  | lname1 | title      | fname2   | lname2 | suffix2
| married? |
      | The Honorable | Pamela  | Patel  | Lieutenant | Jonathan | Patel  | U.S. Navy
| yes      |
      When I ask for formal names
      Then I get "The Honorable Pamela Patel and Lieutenant Jonathan Patel, U.S.
Navy"

      Given a row like:
      | title1        | fname1  | lname1 | title      | fname2   | lname2 | suffix2
| married? |
      | The Honorable | Pamela  | Patel  | Lieutenant | Jonathan | Patel  | U.S. Navy
| yes      |
      When I ask for informal names
      Then I get "The Honorable Pamela Patel and Lieutenant Jonathan Patel, U.S.
Navy"

  Scenario: Married couple with different last names
      Given a row like:
      | title1 | fname1 | lname1      | title | fname2 | lname2 | married? |
      | Ms.    | Karen  | Wasielewski | Mr.   |   Mike | Morand | yes      |
      When I ask for formal names
      Then I get "Ms. Karen Wasielewski and Mr. Mike Morand"

      Given a row like:
      | title1 | fname1 | lname1      | title | fname2 | lname2 | married? |
      | Ms.    | Karen  | Wasielewski | Mr.   |   Mike | Morand | yes      |
      When I ask for informal names
      Then I get "Ms. Karen Wasielewski and Mr. Mike Morand"

  Scenario: Unmarried couple with different last names
      Given a row like:
      | title1 | fname1 | lname1      | title | fname2 | lname2 | married? |
      | Ms.    | Karen  | Wasielewski | Mr.   |   Mike | Morand | no       |
      When I ask for formal names
      Then I get "Mr. Mike Morand \nMs. Karen Wasielewski"

      Given a row like:
      | title1 | fname1 | lname1      | title | fname2 | lname2 | married? |
      | Ms.    | Karen  | Wasielewski | Mr.   |   Mike | Morand | no       |
      When I ask for informal names
      Then I get "Mr. Mike Morand Ms. Karen Wasielewski"
```

WeddingLabelStepDefs.java

```
package cucumber.examples.java.thesis;

import junit.framework.Assert;
import cucumber.annotation.en.Given;
import cucumber.annotation.en.Then;
import cucumber.annotation.en.When;
import cucumber.runtime.PendingException;
import cucumber.table.DataTable;

public class WeddingLabelStepdefs {

	@Given("^a row like:$")
	public void a_row_like(DataTable arg1) throws Throwable {
		// Express the Regexp above with the code you wish you had
		// For automatic conversion, change DataTable to List<YourType>
//		throw new PendingException();
	}

	@When("^I ask for formal names$")
	public void I_ask_for_formal_names() throws Throwable {
		// Express the Regexp above with the code you wish you had
//		throw new PendingException();
	}

	@Then("^I get \"([^\"]*)\"$")
	public void I_get(String arg1) throws Throwable {
		Assert.assertEquals(arg1, "Mr. and Ms. Mike Morand");
	}

	@When("^I ask for informal names$")
	public void I_ask_for_informal_names() throws Throwable {
		// Express the Regexp above with the code you wish you had
//		throw new PendingException();
	}
}
```

RunCukesTest.java

```
package cucumber.examples.java.thesis;

import cucumber.junit.Cucumber;
import org.junit.runner.RunWith;

@RunWith(Cucumber.class)
@Cucumber.Options(format = {"pretty", "html:target/cucumber-html-report", "json-
pretty:target/cucumber-report.json"})
public class RunCukesTest {
}
```

Appendix B (FitNesse)

FitNesseRoot/WeddingLabels/Content.txt

```
!path /cucumber/fitnesse/labels/bin

Scenario: Married couple with the same last name
|fixtures.LabelFixture
|
|title1|fname1|lname1|title2|fname2|lname2|married|formal?               |informal?
|
|Ms.   |Karen |Morand|Mr.   |Mike  |Morand|yes    |Mr. and Ms. Mike Morand|Mr. and Ms.
Mike and Karen Morand|

Scenario: Married couple with the same last name where man is a doctor
|fixtures.LabelFixture
|
|title1|fname1|lname1|title2|fname2|lname2|married|formal?
|informal?                        |
|Ms.   |Karen |Morand|Doctor|Mike  |Morand|yes    |Doctor and Ms. Mike Morand|Doctor
and Ms. Mike and Karen Morand|


Scenario: Married couple with the same last name where woman is a doctor
|fixtures.LabelFixture
|
|title1|fname1|lname1|title2|fname2|lname2|married|formal?
|informal?                           |
|Doctor|Sally |Carter|Mr.   |John  |Carter|yes    |Doctor Sally Carter and Mr. John
Carter|Doctor Sally Carter and Mr. John Carter|


Scenario: Married couple with the same last name where both are doctors
|fixtures.LabelFixture
|
|title1|fname1|lname1|title2|fname2|lname2|married|formal?          |informal?
|
|Doctor|Sally |Carter|Doctor|John  |Carter|yes    |The Doctors Carter|The Doctors
Carter|

Scenario: Married couple with the same last name and different professional titles
|fixtures.LabelFixture
|
|title1       |fname1|lname1|title2    |fname2|lname2|suffix2  |married|formal?
|informal?                                         |
|The Honorable|Pamela|Patel |Lieutenant|James |Patel |U.S. Navy|yes    |The Honorable
Pamela Patel and Lieutenant Jonathan Patel, U.S. Navy|The Honorable Pamela Patel and
Lieutenant Jonathan Patel, U.S. Navy|

Scenario: Married couple with different last names
|fixtures.LabelFixture
|
|title1|fname1|lname1     |title2|fname2|lname2|married|formal?
|informal?                           |
|Ms.   |Karen |Wasielewski|Mr.   |Mike  |Morand|yes    |Ms. Karen Wasielewski and Mr.
Mike Morand|Ms. Karen Wasielewski and Mr. Mike Morand|


Scenario: Unmarried couple with different last names
|fixtures.LabelFixture
|
|title1|fname1|lname1     |title2|fname2|lname2|married|formal?
|informal?                              |
|Ms.   |Karen |Wasielewski|Mr.   |Mike  |Morand|no     |Mr. Mike Morand \nMs. Karen
Wasielewski|Mr. Mike Morand \nMs. Karen Wasielewski|
```

LabelFixture.java

```java
package fixtures;

import fit.ColumnFixture;

public class LabelFixture extends ColumnFixture{

       private String title1;
       private String title2;
       private String fname2;
       private String fname1;
       private String lname2;
       private String lname1;
       private boolean married;
       private String suffix2;

       public void setTitle1(String t) {
              title1 = t;
       }
       public void setTitle2(String t) {
              title2 = t;
       }
       public void setFname1(String t) {
              fname1 = t;
       }
       public void setFname2(String t) {
              fname2 = t;
       }
       public void setLname1(String t) {
              lname1 = t;
       }
       public void setLname2(String t) {
              lname2 = t;
       }
       public void setSuffix2(String t) {
              suffix2 = t;
       }
       public void setMarried(String b) {
              married = (b.equalsIgnoreCase("yes"));
       }

       public String formal() {
              return "Mr. and Ms. Mike Morand";
       }
       public String informal() {
              return "Mr. and Ms. Mike and Karen Morand";
       }
}
```

Appendix C (JBehave)

wedding_label.story

```
Feature: Formal Names for Wedding Labels

Scenario: Formal Variations
Given the first title as <title1>, with name of <fname1> <lname1>
And the second title as <title2>, with name of <fname2> <lname2>
And they are <mstatus>
When I ask for formal names
Then I get <formal names>

Examples:
|title1       |fname1|lname1 |title2    |fname2|lname2 |suffix2  |mstatus|formal names
|
|Ms.             |Karen |Morand |Mr.       |Mike  |Morand |         |married|Mr. and
Ms. Mike Morand                                            |
|Ms.         |Karen |Morand |Doctor    |Mike  |Morand |         |married|Doctor and
Ms. Mike Morand                                       |
|Doctor       |Sally |Carter |Mr.       |John  |Carter |         |married|Doctor Sally
Carter and Mr. John Carter                           |
|Doctor       |Sally |Carter |Doctor    |John  |Carter |         |married|The Doctors
Carter                                               |
|The Honorable|Pamela|Patel  |Lieutenant|James |Patel  |U.S. Navy|married|The
Honorable Pamela Patel and Lieutenant Jonathan Patel, U.S. Navy|
|Ms.          |Karen |Walsh  |Mr.       |Mike  |Morand |         |married|Ms. Karen
Walsh and Mr. Mike Morand                             |
|Ms.          |Karen |Walsh  |Mr.       |Mike  |Morand |         |single |Mr. Mike
Morand \nMs. Karen Walsh                                |

Scenario: Informal Variations
Given the first title as <title1>, with name of <fname1> <lname1>
And the second title as <title2>, with name of <fname2> <lname2>
And they are <mstatus>
When I ask for informal names
Then I get <informal names>

Examples:
|title1       |fname1|lname1 |title2    |fname2|lname2 |suffix2  |mstatus|informal
names                                                   |
|Ms.             |Karen |Morand |Mr.       |Mike  |Morand |         |married|Mr. and
Ms. Mike and Karen Morand                                  |
|Ms.          |Karen |Morand |Doctor    |Mike  |Morand |         |married|Doctor and
Ms. Mike and Karen Morand                             |
|Doctor       |Sally |Carter |Mr.       |John  |Carter |         |married|Doctor Sally
Carter and Mr. John Carter                           |
|Doctor       |Sally |Carter |Doctor    |John  |Carter |         |married|The Doctors
Carter                                               |
|The Honorable|Pamela|Patel  |Lieutenant|James |Patel  |U.S. Navy|married|The
Honorable Pamela Patel and Lieutenant Jonathan Patel, U.S. Navy|
|Ms.          |Karen |Walsh  |Mr.       |Mike  |Morand |         |married|Ms. Karen
Walsh and Mr. Mike Morand                             |
|Ms.          |Karen |Walsh  |Mr.       |Mike  |Morand |         |single |Mr. Mike
Morand \nMs. Karen Walsh                                |
```

WeddingLabel.java

```
package com.labels.stories;

import com.labels.WeddingLabelStory;

public class WeddingLabel extends WeddingLabelStory {
}
```

WeddingLabelStory.java

```
package com.labels;

import java.util.List;

import org.jbehave.core.configuration.Configuration;
import org.jbehave.core.configuration.MostUsefulConfiguration;
import org.jbehave.core.io.LoadFromClasspath;
import org.jbehave.core.junit.JUnitStory;
import org.jbehave.core.reporters.Format;
import org.jbehave.core.reporters.StoryReporterBuilder;
import org.jbehave.core.steps.CandidateSteps;
import org.jbehave.core.steps.InstanceStepsFactory;

import com.labels.steps.LabelSteps;

public class WeddingLabelStory extends JUnitStory {

    // Here we specify the configuration, starting from default
MostUsefulConfiguration, and changing only what is needed
    @Override
    public Configuration configuration() {
        return new MostUsefulConfiguration()
            // where to find the stories
            .useStoryLoader(new LoadFromClasspath(this.getClass()))
            // CONSOLE and TXT reporting
            .useStoryReporterBuilder(new
StoryReporterBuilder().withDefaultFormats().withFormats(Format.CONSOLE, Format.TXT,
Format.HTML_TEMPLATE));
    }

    // Here we specify the steps classes
    @Override
    public List<CandidateSteps> candidateSteps() {
        // varargs, can have more that one steps classes
        return new InstanceStepsFactory(configuration(), new
LabelSteps()).createCandidateSteps();
    }

}
```

LabelSteps.java

```
package com.labels.steps;

import static org.junit.Assert.assertEquals;

import org.jbehave.core.annotations.Given;
import org.jbehave.core.annotations.Named;
import org.jbehave.core.annotations.Then;
import org.jbehave.core.annotations.When;

public class LabelSteps {

	@Given("the first title as <title1>, with name of <fname1> <lname1>")
	public void firstName(@Named("title1") String title, @Named("fname1") String
fname, @Named("lname1") String lname) {
	}

	@Given("the second title as <title2>, with name of <fname2> <lname2>")
	public void secondName(@Named("title2") String title, @Named("fname2") String
fname, @Named("lname2") String lname) {
	}

	@Given("they are <mstatus>")
	public void maritalStatus(@Named("mstatus") String married) {
	}

    @When("I ask for formal names")
    public void whenIAskForFormalNames() {
    }

    @Then("I get <formal names>")
    public void thenIGetFormalNames(@Named("formal names") String names) {
    	assertEquals(names, "Mr. and Ms. Mike Morand");
    }

    @When("I ask for informal names")
    public void whenIAskForInformalNames() {
    }

    @Then("I get <informal names>")
    //@Pending
    public void thenIGetinformalNames(@Named("informal names") String names) {
    	assertEquals(names, "Mr. and Ms. Mike and Karen Morand");
    }}
```