Example Driven Architecture

Moving beyond the fragile test problem once and for all

Gerard Meszaros

FeedXL Pty Ltd

pnsq2014@gerardm.com

Abstract

By using effective automated testing (A.K.A. checking) applications can be evolved more safely as developers add functionality. But the structures of most software systems make it hard to automate tests in a way that doesn't lead to fragile tests.

By preparing automated tests before building the functionality, we can force testability/checkability into the software architecture thereby making test automation both faster and less fragile. As a side benefit, it improves our software design by making it more modular (by making each part of the application testable in isolation).

Modular automated tests have worked well on projects where we have used this approach. It has led to less rework (because the automated tests describe what a successful outcome looks like) and less effort spent on automation. With legacy systems we have been able to drive the evolution of the architecture in a favorable direction by using automated tests for new (or even existing) functionality.

Example Driven Architecture is a people and process solution that can be applied to pretty well any system or application as long as we start early enough to influence the architecture.

Biography

Gerard Meszaros is an independent software development consultant and trainer with 30+ years experience in software and over a decade of experience in agile methods. He started doing eXtreme Programming in 2000 and quickly discovered that close attention to test craftsmanship was essential to keep the cost of change low. He described his key learnings in his book: "xUnit Test Patterns – Refactoring Test Code", which was published in May 2007 by Addison Wesley in the Martin Fowler Signature Series and it won a Jolt Productivity Award in the Best Technical Book category. Since then he has turned his attention to applying the same concepts to organizing the automated acceptance tests as executable examples. He is based near Calgary, Canada and has coached teams and taught courses as far afield as China, India and Europe. He is also the CTO of FeedXL.com which provides a web based diet optimization tool for horse nutrition.

Gerard has a BSC (Hons) in Computer Science from the University of Manitoba.

1 Introduction

This paper describes our experiences driving development using executable examples and how this helped drive testability into the architecture of our system. We start by describing our motivation and the problems we typically encountered. This is followed by a description of the process of specifying behavior by using executable examples.

1.1 A Note on Terminology Used

While the use of the term “automated testing” is widespread in the testing community, there are those in the community who insist that this is actually “automated checking” since testing requires the presence of a “sentient being”. [Johnson 2011] But the term “check automation” is not widely in use, therefore I will continue to use “test automation”; if you prefer to call them automated checks, feel free to do a mental global change from “test” to “check” as this does not change the key message of this paper.

To distinguish a new style of automated tests that avoids many of the pitfalls of traditional test automation, I introduce the terms “example” and “executable example”. In the agile community, there is much debate about whether we should call these self-verifying examples “acceptance tests”, “specifications”, “executable examples” or something else. For the purpose of this paper, I tried to standardize on calling them “examples” but I hope you’ll forgive me if the odd “test” or “check” or “spec” slips in. The reframing of these automatable tests as examples and how that change in terminology changes everything is the essence of this paper. So I’ll be calling them “examples” and referring to the automation as “automated checking of executable examples”.

2 Motivation

Effective automated testing of the behavior of a software intensive system can simplify evolution of the software by making it safer to embark upon changes. Automated testing of the behavior can quickly and cheaply detect any regression bugs while new functionality is being added or bugs are being fixed.

3 Problems With Traditional Approaches to Test Automation

Anyone who has tried to implement automated testing (or “checking”) of a software intensive system has probably encountered many issues, not the least of which is the “fragile test” problem. These issues result in tests that are expensive to build, slow to execute, tests that frequently break and that require frequent expensive fixing.

3.1 The Fragile Test Problem

Fragile tests occur because the structures of most software systems make it hard to automate tests in a way that doesn’t lead to fragile tests. The logic we want to verify is typically tightly coupled to the surrounding “plumbing” (database contents and access mechanism, run-time environment, interacting applications) and, because automated testing efforts typically start after the code has been built, it is very difficult to decouple the logic to facilitate automated testing.

Also, there are many dependencies that tests depend upon which may be outside the control of the automated tests. These include: dependencies on data in the database; dependencies in other systems; and dependencies on properties of the run-time container including permissions and the current system time and date. These are described in more detail in [Meszaros, Gerard. 2004].

3.2 Simply Automating Manual Tests Doesn’t Work

But it is not enough to just automate the tests or checks that human testers would execute manually. Manual testing is very labor intensive and some of the standard practices used to reduce execution cost do not work well for fully automated tests. For example, a manual tester may execute a series of tests where each test uses the leftover state from a previous test. If the previous test gives unexpected results, the tester can adjust their approach on the fly to repair the state of the system, execute a different follow-on test or abandon testing. An automated test cannot be expected to display this level of flexibility; to do so would make the tests/checks unreasonably complicated and even harder to maintain. Therefore, a different approach is required.

3.3 Automating Tests Through the User Interface is Problematic

Another problem with automating manual tests is that they typically interact with the user interface of the system and therefore result in very long, detailed test scripts when automated. This makes the automated test scripts very difficult to understand because they are dominated by the details of the interactions with the user interface elements (buttons, links, fields, etc.) and very prone to widespread breakage as every test that refers to a changed element in the user interface will break when that part of the UI is changed.

3.4 The Root Cause is the Software Development System

The true root cause of this problem is the way we organize our software development organizations: the people who build the system live over there and the people who test it live over here. And there is very little true collaboration between the two groups (or with the specifiers) during the actual development process.

Use cases are too abstract because they enumerate a range of possibilities. (See [Cockburn 1995].) Even use case scenarios are abstract enough to be completely misinterpreted. On the other hand, most automated test cases are too detailed to be easily understood; their primary audience is the computer software that executes them.

Software is inherently difficult to test automatically. This is because each distinct community builds what they think is required (specs, tests, software) and there is little to no cross-checking between them as they work. So the three kinds of artifacts “drift” farther and farther apart. And the test automaters are stuck trying to hold everything together with the predictable result of fragile tests. Improving the technologies available to the test automaters may make them more efficient at bridging this gap but this is a band-aid solution that tries to cover up the underlying cause: the system we typically use to specify, build and verify software is dysfunctional.

4 Specifying Behavior Using Examples

Because the root cause is how we organize the system for building software systems, we need to change that organization to effect a meaningful change. We need to break down the (metaphorical or physical) walls between the specifiers, the builders and the testers, each with their own bosses and goals, to facilitate true collaboration. We need one team (definition: a group of people with a single common goal of specifying, building and testing the software intensive system). By preparing automated tests before building the functionality, we can force testability into the software architecture thereby making test automation both faster and less fragile. As a side benefit, automated tests improve our software design by making it more modular (by making each part of the application testable in isolation).

4.1 Single Cross-Functional Team

The specifiers need to specify the expected behavior of the system using potentially executable examples. They probably don’t have experience doing this, so they should collaborate with the testers in transforming vague statements of requirements into concrete examples that can be automated. But since most tests are too detailed to act as clear examples, the specifiers need to stay involved as the examples/tests are created and not just hand off the specs to the testers.

4.2 Using Examples to Specify Behavior

Concrete examples (that can later be automated as tests) need to be prepared before the corresponding production software is built. This has the following benefits:

• First, the developers have concrete examples of how the system should behave; they can use the real data in these examples to better understand the expected behavior.

• Second, developers are expected to build the necessary adapters to read the examples and use them to execute the relevant parts of the software and verify the correct results are obtained. This forces the developers to design the software to be testable. The examples define both the functional behavior and the testability of the software.

4.3 Managing Scope vs. Detail

Unfortunately, a naive or simplistic approach to automating the testing of a complex system results in very long, complex and detailed test scripts that are difficult to understand and maintain. Since the amount of information a person can hold in their head is widely recognized as 7+/-2 items [Miller, G. A. 1956], we need to strive to keep all examples short and to the point. This requires that we actively manage the level of detail in our examples with this limit in mind. Since we cannot describe all essential details in an example this short, we need to break our specification into different kinds of examples, some of which provide an overview of the functionality (the “big picture”) and others that provide details about various parts of the logic (the “zoomed in” view).

The exact nature of the system being specified will influence how many kinds and levels of examples we require. A typical business information system will have at least three such levels:

1) One or more specifications of the overall workflow
2) Specifications for each major transaction type (e.g. for each use case)
3) Specifications for each business rule or algorithm

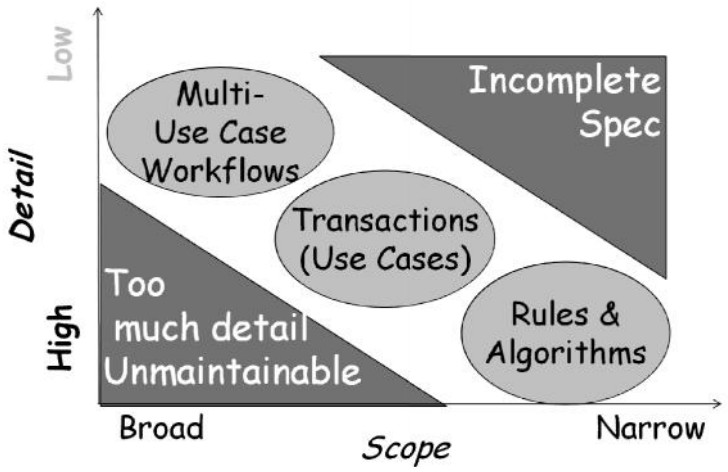

Figure 1 Scope vs. Detail of Tests

We often use a keyword or *action word* based approach [Buwalda et al, 2001, Zylberman et al, 2010] for the first two kinds of specifications but with different keywords for each level. Business rules and algorithms are frequently specified using tabular tests with each row describing an independent example.

The following sections expand and elaborate on this approach.

4.4 Workflow Specification

To provide an overview of how the system is used in the business, we need to provide some examples that illustrate the overall workflow of a business transaction as it flows from actor to actor. The actors may be humans (such as the originator of a request and each person who works on it) or machine (automated processing of workflow steps).

The workflow examples should have one step for each step of the workflow; that is, each actor involved in the workflow should have a single statement describing what they do. Any more than a single statement would result in workflow examples that exceed the 7+/-2 rule for any but the most trivial workflows.

To achieve the single statement per step goal, we need to abstract ruthlessly; we need to remove all non-essential details. To paraphrase Einstein, “If a detail is not essential to the understanding of the workflow, it is essential to exclude the detail.” Therefore, we define keywords for each step that take the fewest possible arguments.

When transitioning from manual test scripts or refactoring existing automated test scripts, we can remove the extraneous details by selecting multiple steps and replacing them with a summary of why they are done. Any data used in the script that is not material to differences between workflows can be “pushed down” into the underlying keyword definitions. By necessity, we must omit any references to user interface in these examples.

Even if an actor goes through multiple steps to execute a single transaction, we capture this as a single keyword with the end result of that series of interactions. For example, the user might use multiple pages to populate their shopping cart, enter their delivery and payment details and confirm the transaction. But for the purpose of defining the workflow illustrating how the purchase will be fulfilled, we would capture all this as a single keyword as in:

1. User makes a purchase and provides payment and delivery details
2. System checks payment info and reserves funds
3. System decrements stock on hand for selected items and produces stock list
4. Fulfillment staff assemble package based on stock list
5. Fulfillment staff mark order as complete
6. System transfers waybill to shipping company to prepare for pickup
7. System processes payment
8. System notifies user of order status

We don’t need to show any of these details in the examples unless what was purchased or how it was paid for affects the fulfillment process. For example, if some items are to be drop shipped from the manufacturer, we might say this as part of the relevant step keywords:

1. User makes a purchase of an item requiring drop shipping
2. System checks payment info and reserves funds
3. System sends drop shipping request to manufacturer
4. Etc.

We can hide the details of what makes an item “drop shippable” in the implementations of the keywords as that is not what we are specifying here.

4.5 Transaction (Use Case) Specification

When defining the expected behavior of a single transaction, we want to show the back and forth interactions between a single actor and the system in question. We want to focus on the essence of the conversation while avoiding the details of the mechanism. That is, we want to show the essential data being communicated but avoid the details of the user interface used to communicate it. We typically define one keyword for each action the user takes and another for each response the system provides.

For example, we might define the transaction behavior as:

1. User provides the name and details of the recipient
2. System constructs a preview of the item
3. User confirms the preview
4. System saves the item

Each of these list items is a single keyword. Note that in this example we do not provide any variable data but it is possible provide arguments to any of these keywords.

4.6 Business Rules Specification

The business rules and algorithms embedded in a software system tend to be complex and require extensive testing. The rules may be hard-coded or they may be defined through configuration (or “meta”) data. There tend to be a lot of cases to be tested based on any one set of rules. Therefore, business rules are best described through common dependency configuration followed by a set of individual examples that can be verified one-by-one. The most common way of doing this is via a data driven approach wherein the same test execution logic is applied to many rows of data, each of which represents an example to be tested.

For example, we could define how a person’s pay is calculated using the following Fit [FIT] table:

HourlyPayDue			
Total Hours	Hourly Rate	Overtime Hours()	Total Pay Before Deduction()
40	20	0	800
41	20	1	830

Table 1 Simple Column Fixture used for Business Rules Example

This table uses a FIT “ColumnFixture” [FIT] to enumerate a set of independent test scenarios or examples, one per row. This table contains two such scenarios: the first 40 hours are paid at the hourly rate and that any additional hours are considered “overtime hours” and are to be paid at time and a half. Note how we pick numbers at the boundary conditions to make it easy to see where the cutoff point is. The first row indicates the name of the “fixture” to be used to interpret this table: HourlyPayDue. This could be a Java class, a C# class or even a Ruby, Groovy or Python class, depending on which version of FIT we are using. FIT allows tables to be surrounded by arbitrary text that can be used to explain the logic or rules which the examples in the table illustrate.

5 Example Driven Architecture

After the specifiers and the testers (traditional role names) have prepared the examples, they are shown to the developers as part of the iteration planning process. Once the team has decided that a particular user story is to be built, the related examples are examined and the high level design of the software can commence.

5.1 Automating Execution of the Examples

The first thing to consider is how the examples can be used as automated tests. Ideally, the examples will not require any changes to do this beyond using a common vocabulary (itself good thing) and syntax; instead, we want to “connect” the examples to the appropriate part of the software being built. Since the example already exists, we know what it needs access to and this helps us define the high level design of the code to facilitate automated execution of the examples.

There are many open source and even some commercial tools available to facilitate connecting the examples to the system. (See [AAFTT 2009] and the section “Tools for Automating Execution of Examples” later in this paper.) The good ones allow us to define multiple levels of keywords to allow us to hide unnecessary details “in plain sight” within the keyword definitions. This keeps the examples short enough while still allowing the curious to see what data is actually being used without having to resort to reading code.

5.2 Automating Workflow Specifications

When automating the workflow specification, we define the tests in terms of keywords that describe what the actor is trying to achieve.

5.2.1 Defining Workflow Keywords

We push the non-essential (but technically necessary) details down into the keyword definitions. If several keywords definitions need to access the data, then we use “variables” to hold the data so we don’t have to repeat them.

Even if an actor goes through multiple steps to execute a single transaction, we capture this as a single keyword with the end result of that series of interactions. For example, the user might use multiple pages to populate their shopping cart, enter their delivery and payment details, and confirm the transaction. But for the purpose of defining the workflow illustrating how the purchase will be fulfilled, we would capture all this as a single keyword as in “user makes a purchase and provides payment and delivery details”. We don’t need to show any of these details in the examples unless what was purchased or how it was paid for affects the fulfillment process.

5.2.2 Implementing Workflow Keywords

The keywords are implemented by making calls into the software system passing the necessary data. This data may have come from within the example (as arguments of a keyword) or from within the keyword definition (if it is necessary but not essential to what we are describing in the specification). Ideally, we do not need to interact with the user interface when interacting with the system; instead we make calls to the same API that the user interface interacts with. If the user interface includes business logic, we may want to interact with the UI in some cases. Some of the “functional test” frameworks (e.g. Robot Framework) provide the means to use different keyword definitions in different test runs; we can use this capability to run the same examples either through the UI or bypassing it simply by providing two sets of keyword definitions.

5.3 Automating Transaction Specifications

The examples for each transaction tell us what information the user will exchange with the system. Therefore we must define a programming interface that corresponds to the requests the user makes and how the system responds. The good news is that the user interface often requires the same set of requests and types of data to function. If this is the case, we can define a common API used by the UI and the transaction specification keywords. The main job of the keyword implementations is to map the format provided by the automation framework to the specifics of the API. For our sample transaction example, these might look like this:

5.3.1 Keyword: user provides the name and details of the recipient

This keyword takes the name and account information provided as arguments, or generates appropriate values that are known to be acceptable. This avoids having to visit a number of examples should the rules for what is acceptable change in the future. It then submits the request to the system under test and stashes the returned result for use by the next keyword.

5.3.2 Keyword: system constructs a preview of the item

The keyword takes the stashed result from the first keyword and validates the contents based on what was submitted, whether provided as arguments by the example author or generated by the keyword. If the stashed result doesn’t match what the keyword expects, it marks this step as failed.

5.3.3 Keyword: user confirms the preview

This keyword calls the API to confirm the information provided by the system. If the system is “stateless”, this will involve sending back the information provided by the system in the previous request as part of the confirmation API.

5.3.4 Keyword: system saves the item

This keyword needs to verify that the system has indeed saved the item. This may be done by interacting directly with the system’s database, or preferably, by interacting with the API used to query the existing items in the system. The latter is preferable since it is less tightly coupled to the implementation in the system. If this results in very slow tests, it is acceptable to interact directly with the database inside a keyword implementation. This is more acceptable than it would be within the example steps since the keyword implementation is a single place that can easily be evolved as the database structure evolves.

5.4 Automating Business Rule Specifications

For business rule specifications, we typically have one row for each scenario we want to illustrate. We need to be able to invoke the software component that implements the business rule, passing it all the data in the row. The component’s API should return the result so that the test execution framework can compare it with the expected result provided in each row.

One testability consideration is where the data on which the example depends comes from. Most often this will be sitting in a database. The example needs to provide the depended-on data as part of its preconditions (prerequisites), so that the example is not dependent on pre-existing data which when changed would result in failing tests. It is less than optimal to have to load the prerequisite data into a database each time the tests are run. A better solution is to architect the component such that it is passed either the prerequisite data or a “data provider” when it is constructed. When used from a test, the data can be passed directly to the component as a “test stub” [Meszaros 2007] containing all the data. This avoids any database access and makes it possible to execute hundreds of examples per second rather than many seconds per example. This is a good example of how “Example Driven Architecture” improves the testability of the design.

6 Tools for Automating Execution of Examples

There are many tools available for automating the execution of the examples.

6.1 Criteria for Tools Selection

The main criteria for selection of a tool for expressing and automating examples are two-fold:

- The ability to express the requirements clearly in the language of the domain experts.
- The ability to connect the examples to the underlying implementation logic as easily as possible.

Most commercial functional test automation tools were built without these as specific goals. Any attempt to use such tools to implement Example Driven Development is doomed to failure.

6.2 Sample tools

The tools listed here are merely examples of tools that have been used successfully with this process. As tools can and do change regularly, the goal of providing these tools as examples is to help the reader understand how to evaluate tools (“teach a man to fish”) rather than to provide a list of tools to be used (“give a man a fish”). The Functional Test Tools project of the Agile Alliance maintains a spreadsheet of testing related tools intended for use on agile projects [AAFTT 2009].

6.2.1 FIT/Fitnesse

FIT (the Framework for Integrated Test) [Mugridge R, 2005] uses tables to represent examples. Fitnesse is a wiki based version of FIT. Each table names a “fixture” class that contains the code used to interpret the table. Implementations of FIT are available for many popular languages. Most include fixture styles that support all three styles of examples described here. The ColumnFixture is the easiest way to implement a business rules test. ActionFixtures and FlowFixtures can be used for keyword driven workflow tests in FIT but they are a bit more complex to create than Robot Framework keywords.

6.2.2 Robot Framework

Robot Framework [Robot Framework] (A.K.A. RF or just “Robot”) can be used to express examples in a keyword driven style. RF Tests can be parameterized with a list of examples to use the data driven style although this is not quite as clean as Fit’s ColumnFixture. Keyword definitions can be stored in libraries, which can be selected at runtime. Keywords may be defined in terms of other keywords or in the underlying programming language. RF is implemented in Python, but can also run on Jython or IronPython allowing keywords to be implemented in JVM languages or .net languages. RF includes a large collection of connectors enabling RF examples to interact with the system under test directly via API calls, via the web browser or many other ways.

6.2.3 Cucumber

Cucumber [Cucumber] is a Behavior Driven Development tool that implements the Given-When-Then style of examples. The specification language is called “Gherkin” and is also implemented by a number of other tools.

7 Sample Specifications

To help illustrate the difference between a traditional, naive approach to test automation and the use of well designed examples at multiple levels, we have a fully worked set of examples to describe how a bank might allow its customers to configure thresholds for when they want to be notified of transactions on their bank and credit card accounts.

7.1 Overview of Functionality

The bank wants its customers to be able to configure thresholds for when they want to be notified of transactions on their bank and credit card accounts.

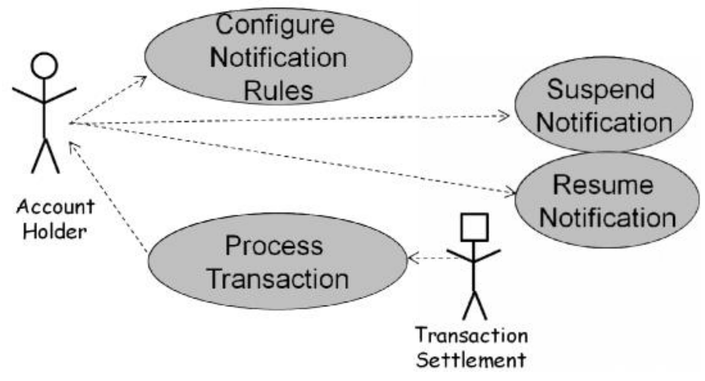

Figure 2 Use Cases of Sample Application

The account holder can initiate one of three different use case transactions. *Configure Notification Rules*, *Suspend Notification* and *Resume Notification*. The Transaction Settlement system initiates the fourth use case, *Process Transaction*, which results in a notification being delivered to the Account Holder.

The business rules of interest are configured in the *Configure Notification Rules* use case and exercised in the *Process Transaction* use case. Therefore minimal workflow we could use to verify the proper implementation of the rules consists of having an Account Holder *Configure Notification Rules* and then having the Transaction Settlement System *Process Transactions*. A more complex workflow would also include the *Suspend Notification* and *Resume Notification* use cases.

7.2 A Traditional Automated Test

The following is an example for the scenario “Configure and Test a Simple Notification on a Single Account” written in Fitnesse using a traditional test automation style.

Because it is written in Fitnesse, it already abstracts away from the user interface, focusing on the user’s actions and the system’s responses. In Fitnesse, vertical bars (|) are used to delimit table cells and each group of lines separated by a blank line represents one table. Note how this example has imposed the testability requirement of being able to advance the system time, something that would not typically be provided but which is essential to effective testing of such systems.

```
| Customer     | bobma     | logs in     |

| System lists all available accounts for the authorized customer    |
|      account            |     type              | notifications            |
| 10035692877        | chequing            | disabled                    |
| 10035692890        | savings              | disabled                    |
| 20010928892        | credit line          | disabled                    |

| Customer sets notification threshold for | all | transactions from | all |locations to |
     $10,000.00    | on account    | 10035692877 | via | email | to |!-bobma@live.com-!  |

| ensure    | No system messages      |

| ensure    | System log contains | "Customer bobma set notification threshold for all
     transactions from all locations to $10,000 on account 10035692877" |

| System lists all available accounts for the authorized customer    |
|    account         |      type         | notifications                     |
| 10035692877   | chequing       | enabled                            |
| 10035692890   | savings         | disabled                           |
| 20010928892   | credit line     | disabled                           |

| Notification settings for account                  | 10035692877                             |
| transaction type | location where initiated  | threshold amount| via    |     address            |
|         all            |                all                | $10,000.00        | email | !-bobma@live.com-!|

| Time now is | 9:30AM, 03/18/2008     |

| Bank processes  | debit       | to| 10035692877    | in the amount of| $15,000.00   |
| Bank processes  | debit       | to| 10035692877    | in the amount of|   $9,000.00   |
| Bank processes  | debit       | to| 10035692877    | in the amount of| $11,000.00   |
| Bank processes  | debit       | to| 20010928892    | in the amount of| $12,000.00   |
| Bank processes  | credit      | to| 10035692877    | in the amount of| $13,000.00   |
| Bank processes  | credit      | to| 10035692877    | in the amount of|   $9,999.99   |
| Bank processes  | charge     | to| 10035692877    | in the amount of|   $9,999.99   |
| Bank processes  | charge     | to| 10035692877    | in the amount of| $11,000.00   |

| New notifications sent to customer                | bobma                                          |
| type   |        account     |   timestamp         | amount        | via     |      address           |
| debit   | 10035692877| 9:30AM, 03/18/2012| $15,000.00   | email | !-bobma@live.com-!|
| debit   | 10035692877| 9:30AM, 03/18/2012| $11,000.00   | email | !-bobma@live.com-!|
| credit  | 10035692877| 9:30AM, 03/18/2012| $13,000.00   | email | !-bobma@live.com-!|
| charge| 10035692877| 9:30AM, 03/18/2012| $11,000.00   | email | !-bobma@live.com-!|
```

Table 2 Overly Detailed Workflow Test for Threshold Based Transaction Notification

This test uses a style of Fit fixture called an Action Fixture. Blank lines separate tables with different functions. Some tables consist of a table type line (e.g. New notifications sent to customer | bobma) followed by a row of column headings followed by a series of table rows. Other tables consist of a single line starting with a verb plus some arguments. Other test automation frameworks may use a different syntax but the intent is the same: to describe the behavior expected in a particular scenario using human readable text.

The problem with this test is that it contains too much detail for the scope (workflow) thereby making it too long and hard to understand. It takes conscious effort to match the notifications to the transactions so that we understand which ones should be notified. And the excess detail is not helping us understand the overall workflow.

Because this is the simplest workflow in the system, we need to find ways to simplify the example so that the more complex workflows won't be even more complex and verbose.

7.3 Changing Scope or Detail

The workflow test contains lots of useful information but too much for the workflow example. Much of it will be useful for the transaction example focused on configuration. But for the workflow example, we want to reduce the level of detail. And for testing the business rules, we need a more compact and easily traceable format of example.

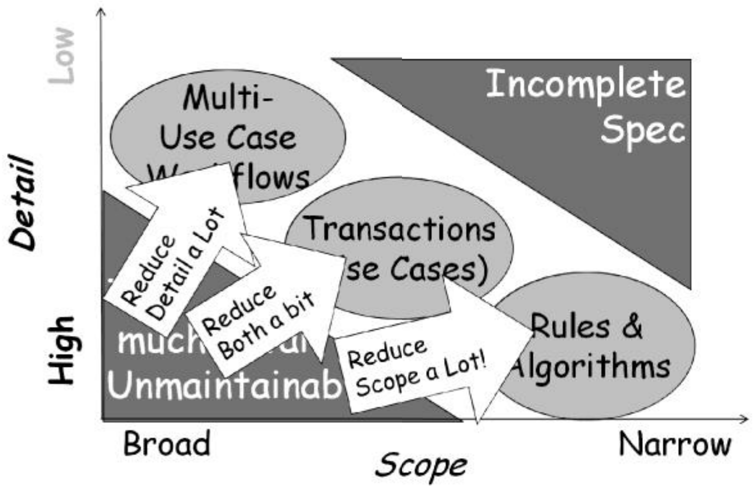

Figure 3 Reducing Detail, Scope or Both

Given that we have too much detail, we are down in the bottom left corner of the diagram and we either need to move up (less detail in a workflow example), to the right (reduce the scope to just the rules component) or both (focused on a single scenario of a single use case).

7.4 Well Factored Workflow Specification

The overall workflow needs to focus on what each actor does and what the expected outputs of the system are.

```
| at some time     |
| customer sets notification threshold to | $10,000 | for all transaction on bobs-checking |

| at a   | later   | time   |
| Bank processes  | debit  | to| bobs-checking   | in the amount of| $10,000  |
| Bank processes  | debit  | to| bobs-checking   | in the amount of|   $9,999  |
| Bank processes  | credit| to| bobs-checking   | in the amount of| $10,000  |

| new notifications sent to customer      | bobma          |
| type        | account               | timestamp    | amount   |
| debit       | bobs-checking   | later             | $10,000  |
| credit      | bobs-checking   | later             | $10,000  |
```

Table 3 Minimal Workflow Specification Example for Threshold Based Transaction Notification

This example contains much less information but it still conveys the overall workflow. Some irrelevant details have been omitted entirely while others have been replaced by meaningful placeholders like “bobs-checking”. We could include more sample transactions but we can do a much better job of illustrating the notification decision logic in the examples for the notification business rule.

7.5 Well Factored Use Case Specification

We can take the naïve workflow test and use the first half of it to describe how the user interacts with the configuration interface to set up the notifications.

```
| Customer     | bobma     | logs in     |

| System lists all available accounts for the authorized customer    |
|      account        |      type               | notifications                           |
| 10035692877   | chequing           | disabled                                  |
| 10035692890   | savings              | disabled                                  |
| 20010928892   | credit line          | disabled                                  |

| Customer sets notification threshold for | all | transactions from | all      | locations to   |
    $10,000.00 | on account | 10035692877    | via | email | to | !-bobma@live.com-! |

| ensure     | No system messages      |

| ensure    | System log contains | "Customer bobma set notification threshold for all
    transactions from all locations to $10,000 on account 10035692877" |

| System lists all available accounts for the authorized customer    |
|    account          |      type           | notifications                                   |
| 10035692877   | chequing        | enabled                                            |
| 10035692890   | savings           | disabled                                           |
| 20010928892   | credit line      | disabled                                           |
```

Table 4 Minimal Use Case Specification Example for Configure Notification Threshold

Here we have included more detail as we are watching for the change in state on the chequing account and we want to make sure all the information that would be shown to the user on the screen is correct.

7.6 Well Factored Business Rules Specification

Now that we have show how the notification rules are configured, we can illustrate how the notification decision is made. First we need to describe the rules via the CustomerAccounts and CustomerThresholds FIT tables. Then we ask the question “Is Notification Required?” for each of a representative set of example transactions carefully chosen to illustrate how the notification logic should work.

CustomerAccounts			
Customer	Account	Label	Added()
bobma	100373	Checking	ok

CustomerThresholds				
Customer	Account	Charge Type	Threshold	Added()
bobma	100372	All	$10,000	OK
bobma	100372	Travel	$1,000	OK
bobma	100372	Restaurant	$100	OK
bobma	100372	Groceries	$264.23	OK

NotificationRequired			
Account	Charge Type	Amount	Notify?
bobma	Travel	999.99	No
bobma	Travel	1,000.00	Yes
bobma	Restaurant	99.99	No
bobma	Restaurant	100.00	Yes
bobma	Groceries	264.22	No
bobma	Restaurant	264.23	Yes

Table 5 Minimal Business Rules Specification Example for Notification Threshold by Charge Type

Each of the rows in the final table describes one transaction to be processed. The first 3 columns represent the key data in the transaction. The final column expresses whether this should result in a notification. When executed, the final column's contents will be compared with what the notification component returns and the cells will be color coded green (for pass) or red (for fail). When the row fails, both the expected and the actual results are shown to help us understand what went wrong.

7.7 Example Driven Architecture for Sample *Examples*

In this section we'll describe how the examples in the previous section are automated and how their presence influences the architecture of our system.

7.7.1 Workflow Steps API

The workflow example requires easy access to an API that corresponds to each step in the example. Providing such an API (as opposed to just the user interface) makes automating the example checking nearly trivial. Since automation of the examples is the developer's job, they are incentivized to provide the API, since the total work to implement the API plus the automation will be less than the work to implement the automation without the API. There should be a strong synergy between automation of the examples and development of the user interface as the API provided for the examples should be a subset of that needed by the UI.

7.7.2 Transaction Steps API

The keyword definitions for the transaction examples should be able to use the same APIs as the user interface. But if this is not the case, the presence of the examples and the need to automate their checking should motivate the developer to refactor the APIs to meet the needs of both the keywords and the UI.

7.7.3 Notification Rules Component

The rules example will motivate the development team to define a NotificationRequired component with a simple API to ask the question "is notification required for a user transaction given the user's notification rules?" In code this would translate to:

```
rulesComponent.isNotificationRequiredFor(aUserTransaction,theUsersNotificationRules);
```

By having the need to automate the example checking, we must make the logic accessible. By passing in the rules with each request, the component becomes stateless making it much easier to test. It is the presence of the examples that drives this componentization of the architecture and the use of “data injection” into the component instead of accessing the rules directly from a database.

8 Results

This testing process outlined in this paper has worked well on new development projects where we have used this approach. It has led to less rework (because the automated tests describe what a successful outcome looks like) and less effort spent on automation. In *Specification by Example*, [Adzic] presents a number of case studies where this process has resulted in good business results.

It is essential to implement all aspects of the automated testing approach, as “cherry picking” of practices can result in less than optimal results. For example, in one case, a business team prepared extensive examples for use by an outsourced development team. But because the outsourcer had their own development process, they chose not to automate the examples and therefore much of the benefit (automated testing, improved testability) was lost.

8.1 Applicability to Legacy Systems

When dealing with legacy systems, we can slowly improve the testability of the architecture by first using keywords that interact with the system through the UI. This makes it safer to refactor parts of the system to make it more testable. Once that is achieved, the relevant keywords can be reimplemented via API calls without having to change the examples themselves because the difference is hidden in the keyword definitions.

9 Conclusions

The use of concrete, automatable examples to drive development is a people and process solution that can be applied to pretty well any system or application as long as we start early enough to influence the architecture. The examples not only help the developers understand what needs to be built, they also help the specifiers clarify their own thinking about what they want built. Furthermore, having the automatable examples on hand while they are building the system encourages the developers to make the architecture amenable to automated execution of the examples as this reduces the effort and uncertainty involved in developer testing of the software.

It is essential to make both process and organizational changes to be successful in this approach. It is the shared responsibility that motivates the various parties to play their part in making test automation simple to construct and maintain. If specifiers and testers are not required to provide the examples before development starts, there is no incentive for developers to define a test friendly architecture. If the developers are not responsible for implementing automated checking of examples, they are unlikely to provide the necessary APIs to make the automation easy to implement.

10 Key Terminology

Use Case: An abstract description of the various ways a user may achieve a particular goal while interacting with a particular system or application. By definition the goal must be achievable in a single sitting. A use case may be part of a larger workflow or business process.

Use Case Scenario: An abstract description of one way a user may achieve a particular goal.

Work Flow: A series of interactions between users and systems as part of a business process. Each user would be executing one use case scenario, likely from different use cases.

Feature: A set of functionality that will deliver value to some stakeholder. Often built incrementally as a series of user stories. May involve one or more use cases. May add scenarios to previously defined use cases.

User Story: The smallest unit of verifiable functionality; used as the increment of software delivery planning on agile projects. Typically involves implementing one scenario each of several use cases.

Test Scenario: An executable sequence of steps to verify that a system implements one or more use case scenarios.

Example: A concrete example that illustrates the behavior the system is expected to exhibit. It may illustrate the behavior describe in one or more use case scenarios. An example can be used as a test scenario, but not all test scenarios make good examples. Unlike a use case scenario, it is concrete, complete with the necessary data. Unlike a test scenario, its primary focus is communication of the desired behavior rather than verification of it. As a result, examples tend to be much simpler with most non-essential details encapsulated behind the fixture.

Executable Example: An example that can be checked automatically by running it via an example execution tool.

Fixture: The code and infrastructure required to enable an example (or test) to be executed. Not really part of the example, much like the store shelving fixture supports the merchandise but is not for sale.

11 References

AAFTT 2009, *Agile Automated Testing Tools Spreadsheet* https://docs.google.com/spreadsheet/ccc?key=0Apag_J97l3CTdHR0WS1sZGFVaFA0dEpXYWRqLXBxV3c&usp=drive_web#gid=1

Adzic, Gojko, *Specification by Example: How Successful Teams Deliver the Right Software.* Manning Publications

Buwalda, Hans. Undated. *Key Success Factors for Keyword Driven Testing* http://www.logigear.com/resources/articles-presentations-templates/389--key-success-factors-for-keyword-driven-testing.html

Buwalda, Hans, Dennis Janssen, Iris Pinkster. 2001. *Integrated Test Design and Automation: Using the Test Frame Method,* Addison-Wesley Professional

Cockburn, Alistair. 2001. Writing Effective Use Cases. Addison-Wesley Pearson Education

Cucumber, *Behavior driven development with elegance and joy.* http://cukes.info/

FIT, *The Framework for Integrated Tests*, http://fit.c2.com/ and http://en.wikipedia.org/wiki/Framework_for_integrated_test

Fitnesse, *The fully integrated standalone wiki and acceptance testing framework*, http://www.fitnesse.org

Johnson, Dion, *Checking Vs. Testing Is Hot! Cyber-Dueling Over 'Check' Vs. 'Test' and Other Semantics,* Automated Software Testing Magazine. June 2011

Keyword-driven testing, http://en.wikipedia.org/wiki/Keyword-driven_testing

Zylberman, Ayal and Aviram Shotte, *Test Language -Introduction to Keyword Driven Testing*, Summer 2010 issue of Methods & Tools http://www.methodsandtools.com/archive/archive.php?id=108

Meszaros, Gerard. 2004, *The Fragile Test Problem*. Proceedings of Agile United 2004 conference.

Meszaros, Gerard. 2007, *xUnit Test Patterns – Refactoring Test Code*, Addison Wesley Professional.

Miller, G. A. 1956. *The magical number seven, plus or minus two: Some limits on our capacity for processing information*. Psychological Review 63 (2): 81–97. Also: http://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two

Mugridge, Rick & Ward Cunningham 2005 *Fit for Developing Software - Framework for integrated Tests*, Pearson Education.

Robot Framework. *Generic test automation framework for acceptance testing and ATDD,* http://robotframework.org/