valtech days

Patrick SMITH

Agitar Software

De Fragile à Agile

octobre 07

Résumé

"The past is a foreign country. They do things differently there."

LP Hartley

The famous opening sequence in LP Hartley's 1953 book "The Go-Between" provides a very optimistic opening sentiment. If only fragile development practices were a thing of the past.

While we may now practice agile methodologies, many of us still work on applications that were developed without the benefits those methodologies provide. These applications are complex to understand and lack adequate testing. Changing or expanding the scope of these applications is risky – things break when changes are made. Unfortunately, we generally cannot avoid making changes.

This presentation explores what makes software projects fragile, and how to begin transforming such fragile projects into agile ones capable of meeting today's rapidly changing business needs.

valtech days

Table des matières

1. AN AGILE VIEW OF TESTING 4
1.1 A trip down memory lane 4
1.2 Is it any different now? 4
1.3 Change is inevitable 5
1.4 Minimize the cost of change 5
2. THE MANY FACES OF UNIT TESTS 6
2.1 Why Test? 6
2.2 Nirvana 7
2.3 Reality 7
2.4 Introducing characterization tests 8
2.5 A closer look at characterization tests 8
3. AGILITY MADE PRACTICAL 9
3.1 Overcoming the testing paradox 9
3.2 No human judgment needed 9
3.3 Develop software with Confidence 9
4. CONCLUSION 10

Liste des Figures

Figure 1: Depiction of traditional development practices as inadequate accommodating change. 5

Figure 2: Contrasting cost of change by development practice. 6

Figure 3: Typical agile lifecycle. 7

Figure 4: Existing applications are typically not created using agile practices. 8

Figure 5: Web sites where Agitar's test generation technology can be evaluated. 10

1. An agile view of testing

1.1 A trip down memory lane

Like any maturing process, development methodologies evolved (and are still evolving) to meet the business needs of the time.

Before software became pervasive, development practices were quite ad-hoc. The focus was on delivering simple functionality that just worked. As software became more pervasive, it started to become more critical to the success of the business. Consequently, development practices started evolving into more formal processes that held the software accountable for delivering the expected services. The notion of specifications and test cases emerged as a way to ensure that key requirements were being met, borrowing in large part from hardware manufacturing practices.

With these more formal development practices, the software was trusted to deliver increasingly complex functionality. Special test harnesses were needed to keep up with the growing body of specifications. These custom test harnesses were expensive to create and maintain. Over time, technology progressed to a point where testing was being designed into the programming languages in the form of multiple entry points, such as main() methods, and special frameworks such as JUnit, were created, shared, and emerged as standards for automating test execution and reporting.

These frameworks made it possible to bring testing sooner and sooner into the development process; to the point that one can now begin testing before the code is even created. In Test-Driven Development, developers use test to drive the design, specifically the APIs and their use, of the application being developed.

1.2 Is it any different now?

"The current state of the software industry is as if the leading edge doctors had tested penicillin, found it to be effective, and integrated it into their practices, only to have 75% of doctors continue to use leeches and mustard poultices."

Steve McConnell

The picture Steve paints may seem disappointing. But consider that not too long ago, unit testing was an unknown practice. There are more conferences, more articles, more general awareness, more tools, and most importantly, more evidence that agile practices really do provide competitive advantages to businesses. I interpret Steve as describing a point in time, not a trend. The industry is starting to capitalize on the benefits of unit testing. To contrast this to a better known technology, IDEs provide value today that was not anticipated when they were first introduced. We will witness that same type of increase in use and value for unit testing. As the industry learns to rely on it more and more, it will discover more ways to draw value from their investments in unit testing.

But it is not just the state of testing that is evolving. More importantly, in my mind, the business of developing software has also changed drastically. Software is not hardware. One of the key benefits of software is that it can be changed easily. Businesses have learned that exploiting this trait is a way to gain a competitive edge. Organizations need to be able to change the applications they create if those applications are going to be long lived.

1.3 Change is inevitable

Originally, software requirements mimicked hardware practices very closely. When parts and equipment need to be manufactured, you really cannot afford to allow requirements to change frequently. Software used to be developed with the same mentality. Applications were expected to remain largely unmodified for their entire lifetime. Changes were really only considered when problems were detected. This philosophy meant that change was not anticipated. This in turn led to practices that made changing code very expensive. And that lead to development practices that sought to minimize the need for change.

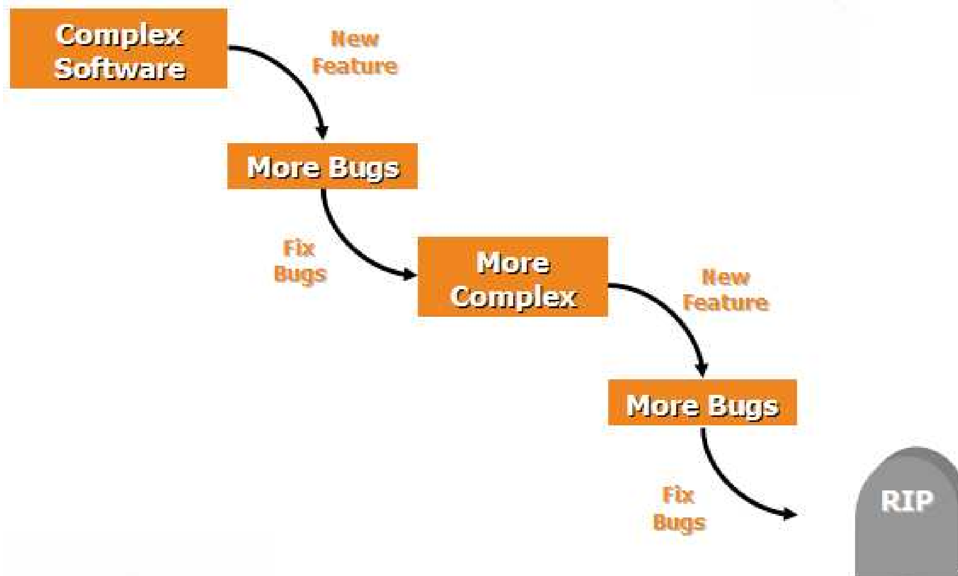

Figure 1: Depiction of traditional development practices as inadequate accommodating change.

As businesses seek to gain competitive edge through software, these development practices started to become inadequate. Change was a constant and business needed a way to embrace and accommodate change. Of course, this gave rise to agile practices that embody the notion that requirements will always and continuously change.

1.4 Minimize the cost of change

Agile methodologies aim to minimize the cost of making changes. They accomplish this through development practices that in part make developers responsible for accommodating change, rather than penalizing them for performing changes. Developers are encouraged to find ways to improve the code so that it can facilitate change rather than spending time on analyses and design activities intended to make change unnecessary. The result is a process rich with feedback that can be used throughout the development process to assess the need for change and gracefully accommodate its introduction.

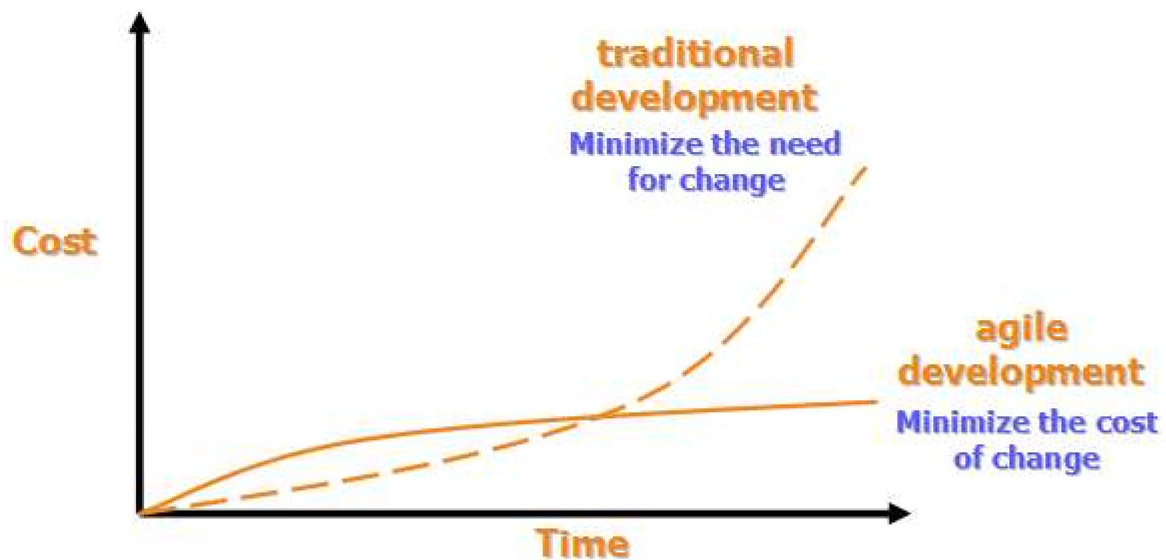

Figure 2: Contrasting cost of change by development practice.

At the heart of this feedback are unit tests. Changes are expensive because things can break when changes are made. In fact the more complex the application is, the more likely changes are to introduce other defects. And the harder it is to detect when things have actually broken. Agile practices minimize the cost of change by addressing the factors that make it expensive:

€ If adding features causes regressions, test everything that can possibly break. In order words, make changes, but detect what you break because things will break.

€ If fixing bugs increase complexity, refactor relentlessly. Simplify everything touched, all the time. Embrace the notion that change is inevitable, and make changing code beneficial rather than harmful.

Code that is less complex is less likely to have defects. Defects in code that is well tested are more likely to get detected. Agile practices provide a disciplined process where time is spent embracing and accommodating change rather than preventing it. Where the ability to accommodate change provides a competitive advantage, businesses that follow these practices stand to benefit tremendously.

2. The many faces of unit tests

2.1 Why Test?

The traditional answer to this question was to find bugs and to prove that requirements were met. The agile answer is much more sophisticated. Tests still specify functionality and demonstrate bugs. But they can also detect regressions, document behavior, drive the design, and even characterize the existing application.

They have evolved into different testing disciplines that even require different tooling. For example, XP describes unit tests as technology (or API) facing tests in the language of the developer. In contrast, XP describes acceptance testing as business (or user) facing tests in the language of the end user. The issue is not which form of testing is better or more effective. These testing disciplines guard against different points of failure. Neither is elective. Both must be performed, along with other forms of testing for performance, security, etc. All of these things can independently break. They must all be tested.

2.2 Nirvana

In a pure, green field, agile project, tests evolve with the code. They exist for every layer of the application to test all of the unique things that can break. Just as the application builds on previous iterations, so do the tests. Adding features or fixing bugs is facilitated by tests that can alert you when you break something unintended, which will inevitably occur.

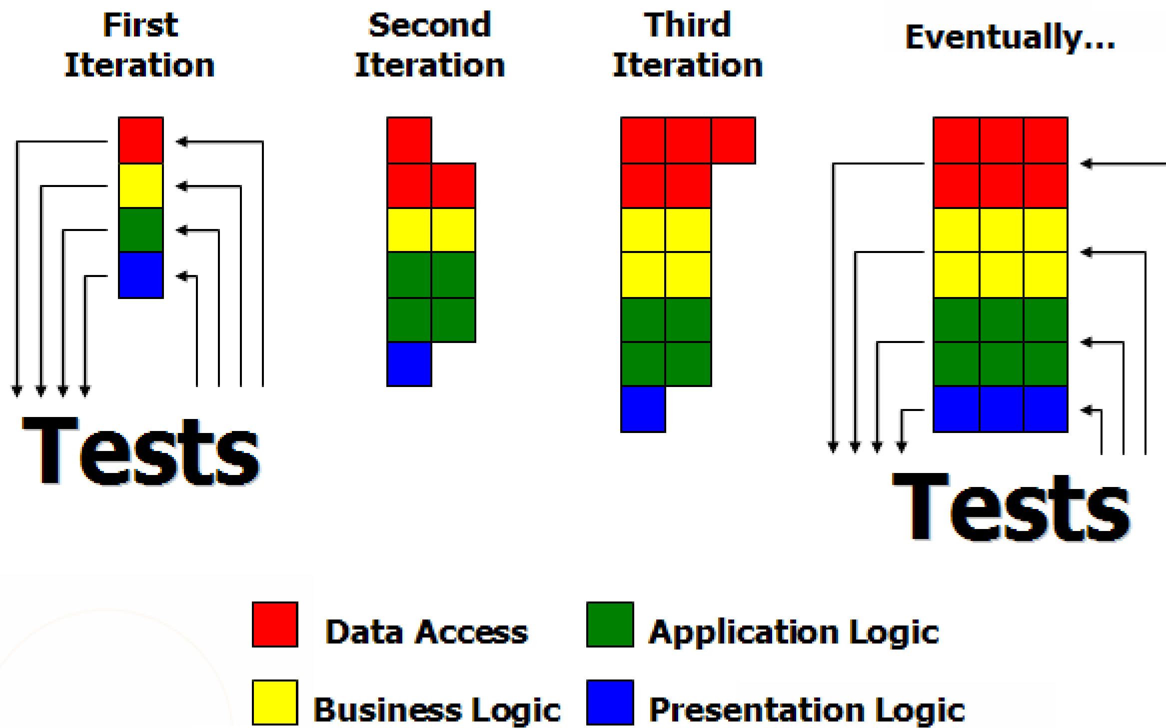

Figure 3: Typical agile lifecycle.

This is how Agile methodologies make your application more robust. Extensive testing requires you to more aggressively manage your complexity, which makes it easier to test.

2.3 Reality

Unfortunately, most of us work on projects that were not developed using agile methodologies. When we make changes to these applications, we really do run the risk of breaking something that we won't be able to detect. It is not so much the breaking of something unintended that is the problem, because that is inevitable. The real issue is not finding out. The real challenge we face is how do we take an existing fragile project, and turn it into a business asset that can accommodate change easily? In other words how do we make our existing projects more robust?

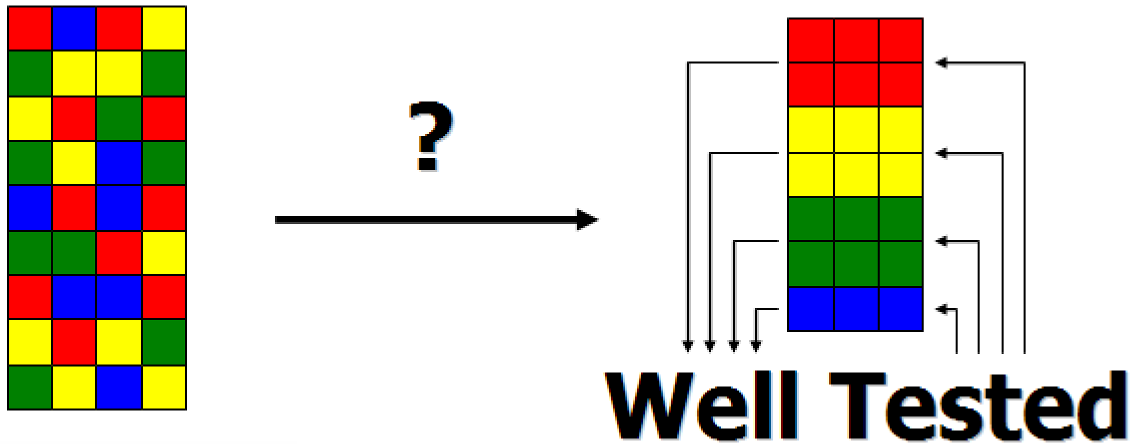

Figure 4: Existing applications are typically not created using agile practices.

As Michael Feathers suggests in his book "Working Effectively with Legacy Code," this is not an easy task to accomplish. Testing complex code is extremely difficult. Quite often, the design simply does not support testing. It may be exceedingly expensive to drive the application to certain desired conditions, and possibly impossible to observe the quantities to verify. Michael describes this as the testing paradox. Code is too complex to test, and too risky to simplify. So where do you begin?

Code is too complex to test, and too risky to simplify.

2.4 Introducing characterization tests

Michael suggests that unit testing can overcome this problem. Actually not just any form of unit testing, but rather specifically a subset of unit tests that he calls characterization tests. Unlike traditional unit tests, these tests do not verify functionality, so they cannot find bugs. Instead, characterization tests describe the behavior (good or bad) of an existing application, so they can inform you when behavior changes.

At a high level, the strategy that Michael lays out is this. Use characterization tests to record all observable behaviors of an existing application. Simplifying the application at every opportunity you get. If you break something, these tests will tell you. Then test parts of the application for correctness as they become simple enough to test.

2.5 A closer look at characterization tests

Characterization tests are unit tests. This means they are technology (or API) facing, and they are written in the language of the developer.

And, as previously mentioned, characterization tests describe the behavior of an existing application. They do not verify functionality. They don't drive design. They don't document APIs or usage well, partly because they don't recognize or respect scenarios. Because characterization tests aren't required to verify functionality or adhere to scenarios, they are actually significantly easier to create. They simply record what an application does, good or bad. Whether the application does what it should is the concern an entirely different unit testing domain. When these tests fail, they indicate behaviors that have changed instead of the presence of a bug.

3. Agility made practical

3.1 Overcoming the testing paradox

The ability to detect changes has an immensely practical benefit to organizations that directly or indirectly depend on software for revenue. Consider that if a developer's job is to create software that delivers services the business can use to compete successfully, then changes to the software's behavior could indicate a loss of a service that could jeopardize the revenue stream associated with that service. Characterization tests help businesses protect themselves against these kinds of regressions. Once a build is known (or assumed) to properly and correctly deliver some services, it can be characterized by unit tests that are run against later code changes to make sure none of these services are broken.

This is a very empowering trait. Remember that our projects are typically too complex to test, and too risky to simplify. The part that makes them too risky to simplify is that any attempt to simplify is likely to introduce additional defects. Characterization tests enable developers to make the changes they need to make to simplify their applications. They will still break functionality and introduce regressions, but characterization will detect those regressions so that the developers can fix them. The net result is code that is more robust, easier to test, and tests that are more powerful.

3.2 No human judgment needed

But there is a problem. In order for characterization tests to significantly reduce the risk associated with simplifying an application, the application must be extensively tested. The value of characterization tests is proportional to the extent of testing. Parts that are not exercised by tests cannot be protected from regressions. This has been the key obstacle to the widespread adoption of more traditional unit testing and should therefore not be underestimated.

However, it helps to realize that characterization tests are not required to adhere to particular scenarios, nor are they required to verify functionality. They simply need to record the existing behavior of an application. This realization is quite liberating. The implication is that no human judgment really is needed to create characterization tests. This, combined with the extent needed to significantly reduce risk makes it a perfect candidate for automated generation.

Similarly, familiarity with the domain and the context really don't provide any benefit when creating characterization tests. The implication of this trait is quite profound. Unlike other forms of testing, the creation of characterization is better done by specialists in characterization tests than by engineers with a strong grasp of the domain and contextual factors. Small teams of specialists can support larger teams of developers. In fact, this is a function that could safely be outsourced to contractors, partners, or vendors with little loss of knowledge continuity. This is a function that is well suited for delegation. Organizations can chose what works best for them and change over time.

3.3 Develop software with Confidence

Agitar's mission is to help enterprises "develop software with confidence." Agitar solutions include products and services designed to help organizations simplify their existing applications and create better tests. The family of products includes:

- **AgitarOne Test Generation**, a completely automated JUnit test generator designed to create highly effective characterization tests. This product is optimized to detect regressions.

- **AgitarOne Agitator**, a Java test automation plugin designed for exploratory testing. This product is optimized to detect unexpected behaviors, and can also be used to verify expected behavior.
- **AgitarOne Management Dashboard**, a reporting portal designed to bring transparency into the development process. This product is designed to provide reports of code rule violations and test results, and help teams assess risk.
- **AgitarOne Delegator Services**, a wide range of services to help teams simplify their code and create better tests.

Agitar's JUnit generation technology can be evaluated through a freely available service called JUnit factory at http://www.junitfactory.com. The entire solution can also be evaluated by requesting a free 10 day evaluation from the company web site at http://www.agitar.com/products/on_site_trial.html. Both of these evaluations are hosted environments, which mean that the code being tested gets uploaded to Agitar servers. The difference is that JUnit Factory is freely available and makes no commitment with respect to privacy or security. The hosted evaluation cluster on the other hand requires SSL, so code is encrypted, and Agitar does make commitments with respect to privacy and security.

Test generation Only

http://www.junitfactory.com

With Agitator and Dashboard

http://www.agitar.com/products/on_site_trial.html

Figure 5: Web sites where Agitar's test generation technology can be evaluated.

Of course, Agitar also offer Proof of Concept evaluations for organizations that are unable to take advantage of JUnit Factory or the Hosted Evaluation service.

4. Conclusion

Most of us work for companies whose success depends on the ability to change existing applications quickly and correctly. Agile methodologies, specifically characterization tests, provide a practical approach to convert existing applications into robust projects able to accommodate

change. Characterization tests have the unique trait of requiring human judgment in order to create. This means that the generation of characterization tests can, and should be, automated. It also means that organizations can delegate the generation of characterization tests should they need to.

Agitar Software is the recognized industry leader in Java unit test automation with best of breed products for JUnit test generation, exploratory testing, and test reporting. Agitar Software professional software organization is the most experienced in the industry at helping organizations create characterization tests to help simplify existing application and make them more robust.