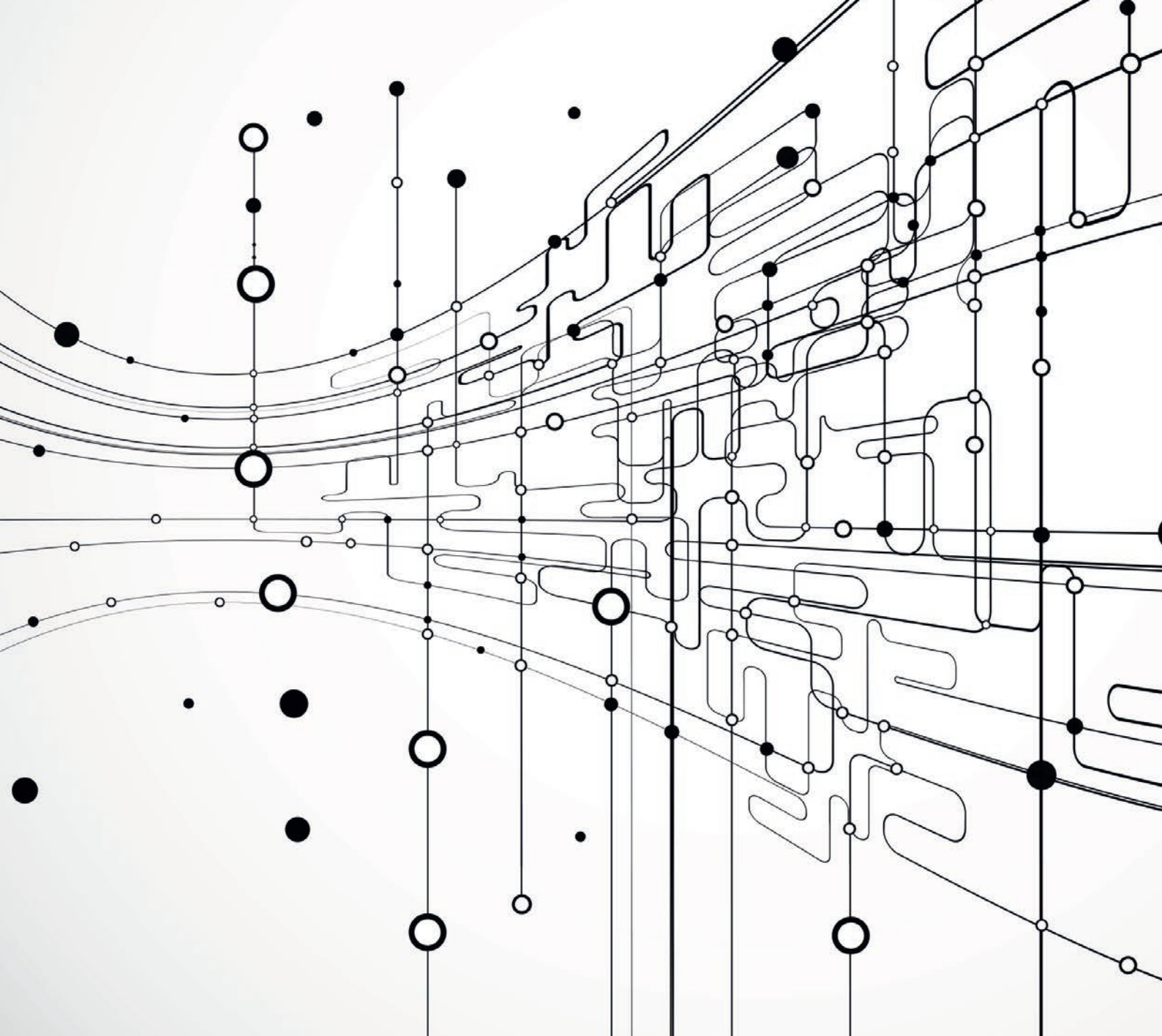


Agile Record

The Magazine for Agile Developers and Agile Testers



Refactoring

www.agilerecord.com

free digital version

made in Germany

ISSN 2191-1320

November 2013

issue 16

JUNE 16–18, 2014 IN BERLIN/POTSDAM

EUROPE'S CONFERENCE FOR MOBILE APPS



The conference for applying innovative knowledge to the mobile app life cycle process.



Management



Marketing



Development



Testing



Design



www.mobileappeurope.com



Dear Readers,

This is the last issue of Agile Record this year and we should also be thinking about the need to refactor the magazine and the magazine's website. We are more or less aiming for the same goals: restructuring the existing body, but keeping the external behavior! We want to improve readability, reduce complexity in the process of generating the issue, and keep or even improve the quality. Of course we also want to increase the readership.

We have seen that code smells and code refactoring is needed in many projects. There are two general categories of benefits that result from refactoring: maintainability and extensibility. There are some techniques that allow you to do more abstraction, techniques for breaking code apart into more logical pieces, and even techniques for improving the names and location of code. Please take a look at this issue in which the authors share their experience in this field with you.

By the time you have received this magazine, the **Agile Testing Days** (www.agiletestingdays.com) will have just taken place. This event is becoming an ever bigger success each year. Besides putting together this issue of Agile Record, the last few weeks have been very busy with preparations for the conference. We not only had amazing speakers and attendees, but the best program ever. Almost half of the conference is conceived as a practical experience. We have early morning lean coffee and the afternoons continue in a practical vein. We have open space, testing and coding dojos, workshops, games, etc. and the conference reflects the agile mindset! We hope to see you again at next year's conference.

We adopt the same concept for the **Agile Dev Practices**. We are just about to finish preparing the program, so please keep tuned and take a look at the program on www.agiledevpractices.com at the end of November. Expect to see many inspiring sessions with gurus from the field of agile development practices. We will rock the market once again. Save the dates and start to allocate the required budget!

I am very proud to inform you about our newest software-related event in the heart of Europe – **Mobile App Europe**. This conference applies innovative knowledge to mobile app management, marketing, design, development, and testing, and takes place on June 16–18, 2014 in Potsdam, Germany. If you are a passionate mobile app expert, don't miss this chance to become an active part of the conference and submit your speaker proposal by December 31, 2013 at www.mobileappeurope.com.

Last but not least, I wish you and your family a great end to the year and, if you celebrate Christmas, have a very Merry Christmas and a Happy New Year 2014.

All the best! Enjoy the read.

Cheers,
José Díaz



| | | | |
|---|----|---|----|
| Editorial..... | 1 | Automated Blackbox Testing of New Age Websites | 41 |
| Editorial Board..... | 3 | <i>by Nilay Coşkun</i> | |
| COLUMN: Refactoring and Technical Debt: It's Not a Choice, It's a Responsibility..... | 7 | Refactoring to Combinators..... | 43 |
| <i>by Robert Galen</i> | | <i>by Carlos Blé</i> | |
| Three Main Business Cases for Refactoring | 12 | Build Your Immune System and Maintain a Healthy Codebase | 46 |
| <i>by Larry Apke</i> | | <i>by Augusto Evangelisti</i> | |
| Big-Data, Cloud and Mobility Are Coming!..... | 14 | Reuse of Unit Test Artifacts – Allow Us to Dream | 49 |
| <i>by Alon Linetzki</i> | | <i>by Yaron Tsubery & Dani Almog</i> | |
| Distributed Agile – Pointless or Possible?..... | 16 | COLUMN: By the way | 53 |
| <i>by Julie Calbourn & Helmut Holst</i> | | <i>by Tanja Schmitz-Remberg & Werner Lieblang</i> | |
| Refactoring – to Sustain Application Development Success in Agile Environments | 21 | Three Tips for Test Refactoring..... | 54 |
| <i>by Narayana Maruvada</i> | | <i>by Gil Zilberfeld</i> | |
| How to Test Refactoring..... | 25 | Business First, Not Test First: How to Create Business Value from Acceptance Tests | 57 |
| <i>by Jeroen Mengerink</i> | | <i>by Dr. Chaehan So</i> | |
| Refactoring – “To Be or Not to Be” | 27 | Explorative C# Web Scripting Using <i>scriptcs</i> and <i>FluentAutomation</i> | 60 |
| <i>by Rashmi Wadhawan</i> | | <i>by Vagif Abilov</i> | |
| COLUMN: Refactoring or the Prevention of | 29 | Ensuring Sustainable Quality of the Product in an Agile Environment with Automated Test Generation | 65 |
| <i>by Daniël Wiersma</i> | | <i>by Anahit Asatryan</i> | |
| How Agile Methods Help Supermassive Games Deal with the Rapid Pace of Game Development | 31 | Refactoring Your Best Asset – Your People – Through Mentoring..... | 67 |
| <i>by Jonathan Amor</i> | | <i>by Peter Saddington</i> | |
| Stumbling Blocks | 33 | Writing Testable Use Cases Using Enterprise Architect.. | 70 |
| <i>by Gurpreet Singh</i> | | <i>by Sander Hoogendoorn</i> | |
| Risk Management in an Agile Way..... | 35 | Masthead..... | 76 |
| <i>by Edwin van Loon</i> | | Picture Credits..... | 76 |
| COLUMN: Agile Architecture Engineering: Dynamic Incre- mental Design Selection and Validation..... | 37 | Index of Advertisers | 76 |
| <i>by Tom Gilb & Kai Gilb</i> | | | |

Editorial Board

Plamen Balkanski



With over 14 years of experience in IT, Plamen Balkanski comes from software development background and has worked in agile, Scrum/XP and Kanban/Lean environments. He has been involved in co-organizing events for the Agile South Coast User Group and ScrumFest.org. He is passionate about helping teams, individuals and companies discover how to work better together and how to continue learning while delivering high quality solutions to business problems.

Matt Block



During his career, Matt Block has played the role of developer, development manager, and product manager. He has learned what development practices and processes are critical to the success of agile and how to drive that adoption process. He has learned just how critical and how difficult the Product Owner role is in enabling the business to realize the full potential agile can bring. Learn more at www.developmentblock.com.

Arjan Brands



Arjan Brands has acted as a test and quality management consultant since 1996. His focus is on process and strategy issues related to testing. As author of TPI Automotive (test process improvement) and test consultant for several branches and companies, Arjan has very broad experience in test management, test process improvement and other quality related themes. Arjan is currently working as a team lead "Agility and Quality" and as a managing consultant for Díaz & Hilterscheid Unternehmensberatung GmbH (Germany), focusing on agile transition and testing activities in agile projects. He is also one of D&H's trainer for different programs (CAT, ISTQB and IREB).

Andreas Ebbert-Karroum



Andreas Ebbert-Karroum leads the Agile Software Factory (ASF) at codecentric. For more than four years, he has been a certified ScrumMaster. Since then he has been able to contribute his competences in small and large (> 300 people), internal and external, or local and global projects as a developer, ScrumMaster or product owner. He is also a Professional Developer Scrum Trainer, conducting a hands-on training for Scrum team members, which codecentric has developed together with scrum.org and in which he shares with joy the knowledge gained in the ASF. More than 15 years ago, his interest in JavaSE/EE awakened and it has never vanished since then. His focus at codecentric is the continuous improvement of the development process in the Agile Software Factory, where technical, organizational and social possibilities are the challenging, external determining factors.

Pat Guariglia



Pat Guariglia is an agile coach for Elegant Agile, Inc. He is certified as a PMP with the Project Management Institute, a Certified Scrum Practitioner (CSP) and Certified Scrum Master (CSM) with the Scrum Alliance. Pat has been leading information technology projects and coaching agile teams for the last twelve years. Pat continues to champion agile practices and Scrum in the Upstate New York and New York City areas. He speaks at conferences, seminars and holds workshops across the Northeast and worldwide. Pat contributes articles to the ScrumAlliance.org web site, is the organizer for the Upstate New York Agile User Group, and participates in the organization and networking of regional northeast US agile groups.

Ciaran Kennedy



Ciaran Kennedy (MSc Technology Management, CSM, CSP) hails from Dublin and has a huge passion for Technology and all things agile. He founded Scrum Ireland (www.scrum.ie), a leading and free social network dedicated to agile IT professionals across the globe. With many leading companies and individual members including Mike Cohn and Jeff Sutherland, Scrum Ireland is growing fast in the community. During the day Ciaran runs his own Agile Company, Chillistore Technologies Ltd. (www.chillistore.ie), providing Agile Office Services to leading blue chip and high tech innovative start-ups. He also does some part time lecturing on Agile Project Management in his spare time and has been a guest speaker at PMI and Engineer Ireland events.

Roy Maines



Roy Maines graduated Cum Laude from Chapman University with a BS Degree in Computer Information Systems. With over 30 years' experience in the IT technology field, Mr. Maines has an extensive background in multiple disciplines in Technical Project Management. Mr. Maines is a certified Project Management Professional (PMP), Certified Scrum Master (CSM) and Certified Scrum Professional (CSP). Mr. Maines has broad experience supporting the full SDLC engineering efforts, Test & Evaluation, Formal Information Assurance Certification efforts. Mr. Maines has held various technical and Project Management positions with a number of fortune 500 companies including Microsoft Corp, Perot Systems, Wachovia Bank NA, and Mantech Systems Engineering Corp. Mr. Maines diverse experiences enable systematic analysis and troubleshooting of large scale technical issues and management of key business metrics.

Editorial Board

Maik Nogens



Maik Nogens works as a testing consultant with Díaz & Hilterscheid, where he focuses on the agile aspects of testing. As part of his passion to support the profession of testers and testing, he is very active in many peer and community setups. Maik is a co-founder of two international peer networks, GATE (German Agile Testing and Exploratory Workshop) and PotsLightning (the pre-Agile Testing Days event held in Potsdam). In his hometown Hamburg he moderates the STUGHH (Software Test Usergroup Hamburg) and also leads the special interest group "Agility" for the ASQF organization in Germany. He is a black belt in the Miagi-Do School of Software Testing, attended the BBST foundation course, is a practicing CAT (Certified Agile Tester) trainer, conducts Testing Dojos and runs workshops for Kanban, SCRUM and other agile areas.

Steve Rogalsky



Steve Rogalsky gets a thrill out of solving problems, working with teams, and helping people learn and improve. He applies this to software development using lean and agile techniques as a process hacker, coach, analyst, tester, developer, speaker, and agile advocate. He blogs at winnipegagilist.blogspot.com and works at protegra.com.

Zuzanna Sochova




Zuzanna Sochova is a leader of the Czech agile community Agile Association – AgilniAsociace.com, organizing conferences and events and sharing the agile experience all around. She works as a trainer, consultant and coach for software organizations, supports them in tailoring their agile adoption processes to company culture (agile-scrum.com). She regularly speaks at international agile conferences. She is Managing Director of LOTOFIDEA.

Declan Whelan




Declan Whelan is an agile coach at LeanIntuit where he helps organizations improve value delivery through agile and lean methods. He is a cofounder and CTO at Printchomp where they are disrupting the print industry by building a realtime marketplace for printing. Declan is also a board member at the Agile Alliance.

Build your agile team with the



iSQL Agile Teaming certifications.



Certified Agile Essentials

Introduction to Agile Methods

This 2-day course is an introduction for anyone who wants to learn the fundamentals of agile processes and take the first step into the world of agile. This course was created for developers, testers, analysts, project managers, engineers, IT directors and everyone else working in an agile environment.



2 days



English, German




1,198 €



incl. exam

Booking and more information:
training.diazhilterscheid.com

Build your agile team with the



iSQL Agile Teaming certifications.



Certified Agile Tester®

Professional Agile Testing | Transition to Agile

This pragmatic, soft skills-focused, industry-supported 4-day course (plus 1-day exam) was designed especially for advanced software testers, developers and everyone else working in testing-related projects. The fifth day consists of the practical assessment and the written exam.



5 days



English, German



2,500 €



incl. exam



Díaz Hilterscheid

Díaz & Hilterscheid GmbH / Kurfürstendamm 179 / 10707 Berlin / Germany

Tel: +49 30 747628-0 / Fax: +49 30 747628-99

www.diazhilterscheid.com training@diazhilterscheid.com

Refactoring and Technical Debt: It's Not a Choice, It's a Responsibility

by Robert Galen



I was coaching a rather large group of Scrum teams at an email marketing SaaS firm. The group was relatively mature and had been practicing Scrum for over 4 years. Over the years, though, the organization had embraced Agile principles and was well on its way to becoming a high-performance agile organization. Most of my efforts were towards “fine-tuning” from the perspective of an “external set of eyes”. It was a privilege working with this organization and its development teams.

But, as with anything in life, there were always challenges and room for improvement. I remember attending a noteworthy backlog maintenance meeting with one of the teams. This particular team was incredibly strong, so I was simply attending to check on how well they were grooming. To be honest, I was hoping to share some lessons from their approaches with some of the less experienced teams.

Jon was one of the “lead engineers” on the team. He had been a Scrum Master for a while, so his agile chops were mature and balanced. However, I was surprised when the following happened:

Max, the Product Owner introduced a User Story for the second time in maintenance. The team had already seen it once and had realized two things:

1. It was bigger than a Sprint's worth of work for the team (call it an Epic or non-executable Story), and
2. They needed more information about the legacy code-base surrounding implementing the story.

So they created a Research Spike that represented technical investigation into the story.

This session was the first time the team had got back together after their “learning” from the Spike. Jon had

taken the lead on the Spike, working with two other team members.

He went over the implications from a legacy code base perspective. Jon started the discussion. He and his small team recommended that they split the Epic into three sprint-digestible chunks for execution. Two of them had a dependency, so they needed to be worked in the same Sprint. The other needed to be worked in the subsequent Sprint in order to complete the original Epic.

Jon and his team had reviewed the legacy code base and said, in order to do the work properly; it would take a total of approximately 40 Story Points. However, he pointed out that this might be perceived as excessive and that approximately 25 of those points would be spent on refactoring the older code base. The specific breakdown was 18 points for the “new functionality” and 25 points to refactor related legacy code.

The Product Owner excitedly opted for the 18 points and deferring the refactoring bits. Jon and his small Spike team wholeheartedly agreed and the entire team went along for the ride. From a backlog perspective, the 18 points worth of stories became high priority and the refactoring work dropped to near the bottom of the list.

And the meeting ended with everyone being “happy” with the results.

I decided not to say anything, but I left the room absolutely deflated with this decision. It was opposed to everything we had been championing at an agile leadership level. Clearly put, we wanted the teams to be doing solid, high quality work that they could be proud of. In fact, all of our Definition-of-Done and Release Criteria surrounded those notions.

If the cost of this Epic was approximately 40 points to do it “right”, then that was the cost – period. Splitting into the parts you “agreed with” and the ones you “didn't agree with” were not really options. Sure, each team needed to make the case to the Product Owner surrounding the “why” behind it, but it was not a product-level decision; it was a team-based, quality-first

decision. De-coupling the two broke our quality rules and that decision would haunt us later as technical debt.

To close this story, I used my not-so-inconsequential influencing capabilities to change this outcome. We decided that this Epic was important enough to do properly and that the approximately 40-point cost was worth the customer value. In other words, we made a congruent and sound business decision without cutting corners. And the team fully appreciated this opportunity, without second-guessing and guilt, to deliver a fully complete feature that included the requisite refactoring to make it whole.

Now, I only hope they continue to handle “refactoring opportunities” the same way.

Refactoring Versus Technical Debt

Any discussion on refactoring has also to include the notion of technical debt. The two are inextricably linked in the agile space, meaning refactoring is a way of removing or reducing the presence of technical debt. However, not all technical debt is a direct refactoring candidate. What I mean by that is that refactoring typically refers to software or code, while technical debt can surface even in documentation or test cases. So it can be a bit broader if you want to consider it in that context.

Broad Versus Narrow Consideration

Typically any discussion on refactoring is embedded in “the code” – usually the application or component-level code in which you are delivering your product. Sometimes, although much more rarely, the discussion extends to supporting code such as automation, build infrastructure, and supporting scripts.

I would like to make the coupling even stronger between technical debt and refactoring. To me, you refactor away technical debt. You *identify the debt* and the *effort to remove it is refactoring*. Now code is a primary place for it, but I believe you can and should refactor “other things”, for example:

- The graphical **design** on the wall that no longer represents the design of your product;
- The **test case** (manual, automated, or even exploratory charter) that is no longer relevant given your products behavior;
- The **wireframe** that has iterated several times with the code and is now out of date;
- That wiki page that speaks to new team members on how to build the application or other team-based **documentation**;

Show your full potential!



Call for Articles

Become an author for the Agile Record magazine and share your knowledge and experience with other professionals from the field.

The next issue of Agile Record, on the topic of “Security Testing in an Agile Environment”, will be out in February 2013.

Get your articles in for review by December 15, 2013!

Find more information on our website at:

www.agilerecord.com

- The **test automation** that the team broke during the last Sprint and failed to fix;
- The **tooling** that everyone uses to measure application performance, but that needs an update;
- The team **measures** on throughput that have not been updated and no longer apply because the team moved from Scrum to Kanban;
- Or the current **process** a team is using for story estimation that is not serving them very well.

Clearly I lean towards a broad-brush view to refactoring responsibilities and connecting them to the various kinds of technical debt. From my perspective, I'd recommend that you deal with it as broadly as possible within your own contexts. But let's move beyond talking about refactoring and instead explore some strategies for dealing with it.

Strategies

I have used a set of strategies quite effectively to combat technical debt and inspire refactoring in several companies. There is no succinct "silver bullet". However, if you apply the following with persistence, you will be well on your way to delivering more sound and robust products.

Stop Digging the Hole Any Deeper

Almost a no-brainer initial strategy is to "stop making your debt worse"! This involves all new functionality. For every story that you develop, you want to ensure that you do not make your technical debt any worse. So, the initial story is very relevant here. You want to hold the line on new work and make sure you are "doing things right".

While I was coaching at iContact, a trigger word in our team discussions was "hack". Whenever a team member spoke about "hacking something together", we knew that it would be creating technical debt and need later refactoring. So we worked incredibly hard to avoid "hacks".

Fill in the Hole

Once you show the discipline to hold the line on new work, you can then go back and start refactoring legacy crud that has developed over time. This usually is a longer-term strategy in many organizations and requires great persistence. It is also a moving target to some degree, so patience is needed as well.

I like to engage the team in identifying refactoring targets. Avoid the "we have to fix everything" syndrome and ask the team for the highest priority refactoring targets by way of value – for example, removing impediments to the development team's efficiency or capacity.

Broadly Attack Refactoring

Balance is a key in refactoring. Attack technical debt in all its forms and do not necessarily focus on one component or type of debt. You want to look at your entire codebase, tool-base, script-base, documentation-base, etc. in your retrospectives

and select high-impact, high-return refactoring opportunities. Then apply a bit of relentlessness in pursuing and improving those areas.

Make the Business Case

Even though I talk about refactoring being an organizational and team responsibility, it does not get supported by magic. Teams need to identify (document) their refactoring work on their Product Backlogs. The business case for each improvement needs to be explored or explained, particularly if you are going to get your Product Owner to support you.

So yes, it is a responsibility. But you need to put the rationale and the ROI in clear business terms. Then "connect the dots" back to the ROI after you have refactored the code, perhaps discussing or showing improved implementation speed in a Sprint Review.

Talk About the COST

Remember that refactoring often has a cost in time-to-market. Bugs take longer to fix or cluster in ways that influence customer confidence and usability. Maintainability is a strong factor in being truly nimble and creative. At iContact we often selected and justified our refactoring targets by how they would support our future product roadmap and support faster implementation times.

Then, when we had completed the refactoring, we would look back on those improvement estimates and speak to the reality of the improvements – connecting the dots, if you will.

Don't Attack Too Much at Once

One of the hardest things to do in many organizations, those with debt-rich legacy systems, is to prioritize the technical debt. There is so much and it is causing so much harm, that the inclination is to try and fix it all at once. But nothing could be more detrimental from a business perspective. As you would handle anything on your backlog, prioritize it and systematically attack it.

Invest in Test Automation

I have often heard the notion that a value proposition of building solid test automation is that it provides a "safety net" so that the team can courageously refactor. The point is that if there is little to no test automation, teams are reluctant-to-fearful to refactor because of side effects and how hard it is to detect (test for) them. I have found this to be incredibly true.

So a part of any refactoring effort should be focused on building test automation coverage. The two efforts strategically go hand-in-hand.

Find Release Tempo Opportunities

Most agile teams today have multiple tempos: sprint tempo, release or release train tempo, and calendar or external release tempo. You want to think about your various tempos and perhaps find opportunities within them for a focus on technical debt and refactoring. For example:

Many SaaS product companies have downtime periods in the calendar year when they do not necessarily want to release new code to their clients. At iContact, our time was over the Christmas holidays. From November to December each year we needed to keep releases to a minimum while our customers focused on holiday revenue. Given that, we would plan “Refactoring Sprints & Releases” over that period. Sometimes we focused on product code, for example broad-brush defect repairs and component or feature refactoring. Another season we worked on our continuous deployment capabilities – focusing on scripting, tools, and deployment automation.

It was a great way for us to make investments and not disrupt our Product Roadmap plans.

Make it an Ongoing Investment

And the final strategic point is making it clear to everyone that technical debt and refactoring are an ongoing challenge and investment. They will not “go away”. Even if you are implementing a greenfield project, you will be surprised how quickly you gain refactoring debt. It is driven by the nature of software – the customer needs change, technologies evolve and change, and teams change. In other words, change happens, so simply factor it into your strategic plans, roadmaps, and ongoing product backlogs.

Wrapping Up

The sub-title for this article was: it’s not a choice, it’s a responsibility. I hope the introductory story helped to crystalize that point. But I would like to emphasize it even more now.

Stakeholders will rarely tell you where and when to refactor. In fact, they typically hate the notion of refactoring, infrastructural investment, ongoing maintenance, etc. Instead they usually push their teams towards more and more new features. This pressure is organizational and will seep into the product organization, each Product Owner, and their teams. However, just because we are under pressure, it does not mean we need to abdicate our responsibilities and blindly succumb to it.

Rather, we need to activate our craftsmanship, our professionalism, our responsibility for doing good work and our courage to deliver that work. In other words, delivering software that will stand the test of time, that will exceed our customers’ expectations, and that we can be proud of. All of that might sound trite or too simplistic, but it is a core part of the principles behind the Agile methods.

And, beyond simply words, each agile team and organization needs to make its technical debt (risks) and its refactoring efforts (investments) transparent. They need to become part of the everyday discussion that teams, managers, and senior leaders have as they transform their organizations towards agile execution. Striking a transparent balance should be the goal. And I strongly suspect that everyone’s “common sense and gut feelings” will let him or her know when they have achieved it.

As always, thanks for listening,
Bob.

References

- [1] Technical debt definition – http://en.wikipedia.org/wiki/Technical_debt
- [2] *Managing Software Debt* by Chris Sterling is a wonderful book dedicated to all aspects of technical software debt.
- [3] Here’s a link to an article/whitepaper I wrote on Technical Test Debt – a variant of technical debt that focuses on the testing and automation aspects – <http://www.rgalen.com/presentation-download/articles-general-guidance/Managing%20Technical%20Test%20Debt.pdf>
- [4] A recent perspective by Henrik Kniberg – <http://blog.crisp.se/2013/07/12/henrikkniberg/the-solution-to-technical-debt>
- [5] A fairly solid overview of technical debt with some solid references – <http://queue.acm.org/detail.cfm?id=2168798>
- [6] Israel Gat of the Cutter Consortium has published several papers with his views on measuring and the ROI of Technical Debt. Searching for his works would be a good investment. ■

> about the author

Bob Galen



Bob Galen is President & Certified Scrum Coach (CSC) at RGCG, LLC a technical consultancy focused towards increasing agility and pragmatism within software projects and teams. He has over 30 years of experience as a software developer, tester, project manager and leader. Bob regularly consults, writes, and is a popular speaker on a wide variety of software topics. He is also the author of the books: “Agile Reflections” and “Scrum Product Ownership”.

He can be reached at: bob@rgalen.com

Twitter: [@bobgalen](https://twitter.com/bobgalen)

SHARE YOUR MOBILE EXPERIENCE

Submit your proposal now and become a speaker at **Mobile App Europe** – the conference for applying innovative knowledge to the mobile app life cycle process.



Management



Marketing



Development



Testing



Design



**JUNE 16–18, 2014 IN
BERLIN/POTSDAM**

www.mobileappeurope.com



Three Main Business Cases for Refactoring

by Larry Apke

Chances are if you are reading this, you agree that refactoring is a good thing. Odds are also good that you are working for a company that does not see things the same way. When it comes to the tactics of refactoring, I believe that you will often find a great deal of agreement – identify the most complex code that is enhanced often, write appropriate automated test coverage, employ an appropriate refactoring strategy, and so on. And, while you should be commended for your commitment, code quality, and knowledge of how to improve, you will get nowhere without having the support and funding necessary to achieve your vision. This article will serve to explain why it's crucial as a business to refactor existing code and why it makes sense to take care of it now rather than later.

This is not news, but business folks are funny about money. They usually will not allow you to spend money unless you can show them a definitive return on investment. This is the first hurdle that makes refactoring a tough sell. While business easily understands chasing the next feature or BSO (Bright Shiny Object), they are less apt to understand shoring up fragile code. I can imagine the conversation to be something like this:

Code Quality Enthusiast (CQE): “Please give me a chunk of your precious budget so that I can improve our underlying code.”

Business: “OK. So what are you going to give me as far as functionality?”

CQE: “There will not be any new functionality. I will merely be making the existing code better.”

Business (from a distance, chasing the next BSO): “No, thank you. I only have so much money and I have too many new features to build.”

Sounds familiar?

Three Business Reasons for Refactoring

There are three main business reasons for refactoring existing code. There are any number of books, blogs, and papers that outline these, but I recently stumbled across a new PhD thesis by Dan Sturtevant at MIT, entitled “Technical Debt in Large Systems: Understanding the cost of software complexity”. According to the thesis, he conducted the study within a successful software firm, Iron Bridge Software, and measured the architectural complexity of eight versions of their product. He also measured the defect density, developer productivity and staff turnover rate along with the eight versions. I feel that his findings more than adequately demonstrates these key things:

- **Quality** — complex code has been shown to contain more defects and the chances of defects are greater when enhancing complex code.
- **Productivity** — developer productivity decreases with code complexity.
- **Employee Morale** — when developers are forced to work with complex code, they tend to find less job satisfaction and tend to leave companies in greater numbers

Quality

It is intuitive that the number of defects would be greater, the more complex the code. Also intuitive is the fact that when we have enhancements that force us to work with existing code, we are more liable to inject defects. The question is not really whether this is true, but to what extent does complexity contribute to defects.

Sturtevant put some numbers to the obvious by using actual code from a “successful” software company (which I expect would be biased towards quality code, as most companies are not into airing dirty laundry), and he found that there is a 310 % increase in defect density as code moves from low to high architectural complexity (as measured by dependent classes). Also, there is a 260 % increase as code moves from a low to high McCabe Cyclomatic Complexity (a measure of the number of linearly independent paths – can also be thought of as branches – through a program’s source code).

When the two measures of complexity are combined, as they usually are for bad code, the effect is an 830 % increase in defect density as you move from the simple to the complex. These are sobering numbers. When it comes to defects, you can “pay me now” or pay dearly later when you ship defective code or spend a great deal of time fixing all the defects you created by not writing clean code in the first place.

Productivity

It is also intuitive that working with complex code will be harder than working with non-complex code. Has anyone ever been on an excruciatingly long email chain just trying to figure out where a problem is in some code, or maybe that “all hands on deck” phone call that has dozens of people trying to assess a defect all day long? I remember one particular issue that took four different email chains, countless conference calls, and one face-to-face meeting to simply diagnose the cause of a defect. On one of these never ending email threads I counted 15 people, represented by 12 different managers who had sent 22 emails over a seven day period – and this was one of four

such threads! (In addition to proving lowered productivity due to complexity, it also proves lowered productivity due to siloing.)

I am sure that anyone in the software business has similar harrowing tales, but these are anecdotal. What hard numbers can we put to the productivity that is lost through code complexity? Again, Sturtevant provides us with an answer. There is a about a 50 % decrease in productivity as you move from the simple to the complex. In the end, I receive half as much productivity for my money with complex code bases. As we move to more agile processes, this explains some of the “failures” that (mostly) large companies are experiencing. It is very difficult to increase time-to-market when the complex code base bogs our developers down.

Employee Morale

We all know it is not fun to have to work with bad code, but how does this affect things like employee morale? In my opinion this is the most sobering statistic of all. Sturtevant found that when you move from a simple to complex code base, the number of voluntary and involuntary employment terminations increases ten times!

Sturtevant’s numbers do not explain why turnover happens with bad code, they only show a correlation. My interpretation is this: good software development companies know what it takes to write good code. Therefore, since they know more about the code itself, they also know about the people who do the coding and they treat them accordingly. The places that have a lot of complex code obviously either know very little about software development, or lack the power to adhere to good software development principles. Either way, they are not able to treat their developers as they should be treated and their developers vote with their feet by walking out the door.

Conclusion

In the end, trying to get management to understand the value of refactoring and producing good code is a tough sell. I hope that these numbers can help make a good business case for aggressively refactoring complex code that is actively being changed, especially since no matter what you do you will pay the price. The question is – would you rather pay less today or more tomorrow? ■

> about the author

Larry Apke



Larry Apke has had over six years' experience as an Agile and Scrum Expert and as a highly accomplished visionary executive with a history of technology leadership and innovation. He is a results-oriented, decisive leader with proven success in devising and implementing solutions that

deliver solid ROI for organizations in various industries, including software, healthcare, and aviation. He has also had success with building client relationships and establishing effective long-term IT and operations strategies. He has worked with dozens of teams and hundreds of team members helping transition them from a phased-gate development approach to a more agile methodology. During this time he has held many different titles but all of his experience has been with Agile.

He is the President of Dr. Stork Software and has previously worked with Harris Computer Systems as the Director of Software Development, the Managing Partner for vicCio Group, the Software Engineering Manager for the Apollo Group, the Scrum Master for American Traffic Solutions, the Scrum Master & Project Manager for Early Warning Services and as an Agile Expert for Neudesic.

Twitter: [@Agile_Doctor](#)

Big-Data, Cloud and Mobility Are Coming!

by Alon Linetzki

If we look at the changes that technology brings us today, and at what organizations have to deal with in their development teams, we witness big-data, cloud, and mobility – which are the main factors for the complexity in many systems.

Those technologies, in combination with the DevOps approach, which is catching up and evolving to include production in the scope of the basic development life cycle process, introduce many challenges to the testing world.

This short article presents some of the trends and complexities coming our way that demand attention and preparation in software testing, and so will be relevant and add value.

What are we facing? In which areas should we prepare?

Testers are no longer faced only with technical challenges, but have to answer and bring solutions to extremely high complex algorithms, data which is scattered throughout the world, legal issues of data with regard to the privacy act (especially when using cloud technology and distributed databases), and the fact that our phones (especially Android versions) are facing high challenges with regard to security and privacy.

In light of the above challenges, I believe testing has to change. We will have to get closer to requirements in order for us to add more value to the business, thereby introducing greater and more robust test design, integrated with risk based testing approaches, to leverage production. We will have to introduce new tools to test things in the cloud, combining big data elements and legal elements. Elasticity, performance, and many other things (e.g. security) have to be addressed in the testing life cycle, in methods, and in processes and tools, including the automation approach, and automation tools and platforms.

We will have to use more and more modeling techniques to be able to introduce robust model-based testing into our automation, and to be able to do it fast and efficiently, as Agile approaches are becoming more and more in demand and common, and are starting to be implemented in corporates, in addition to the technologies mentioned.

A pattern we are likely to see in multiple products is that a big data cluster will be connected to the cloud, and end users will retrieve data from it onto their laptops, tablets, and mobile devices in a secure way. This will give us good performance and reliable information. The information will arrive on time (high speed) and will be location-based in many cases, or at least relevant to our personal profile (as individuals).

The use of big data is already introducing complex algorithms and new search engines to DBs (like graphical engines/DBs), which present a challenge to test (e.g. how can we verify that the data we got, which came from many DBs scattered in the cloud, are the right ones for the question/query we have requested). Securing this data over the air for mobile devices and tablets will demand new approaches, tools, and mechanisms, as well as higher speed from cellular operators, to be able to open up the market to such application vendors.

In order to prepare for such challenges, we should be innovating new testing platforms (for automation), new tools (for cloud and big data investigation), and new processes and methods for developing test design using risk based testing as an integrated part of our life cycle.

As testers, we accumulate a lot of data on testing, on defects, on impacts, and many other important and critical items of information. I believe we should start looking at BI (business intelligence) tools for testing the data, so we will be able to analyze it faster and with higher accuracy. That way, it should be more immediately available to us and will enable us to investigate trends over projects, over time, and across organizations. BI tools are implemented today in many other fields and are used in many cases for the same scenarios I have just mentioned. Ways of integrating BI tools to help testers and test managers might be found very useful in years to come, and enable us to cope not only with the complexity of technology on the development side, but also to handle the enormous amount of test data which is now being collected, and will be coming in even greater amounts, when trying to solve the testing challenges introduced in this article.

The cloud and other technologies also bring huge amounts of data that we cannot (or will be not be able to) simulate in our labs. If you combine that with the DevOps approach which is

starting to catch up in the world, the result is that testing in production is a trend which will be forced on us in order to cope with the business demands. It will introduce tools and mechanisms that have to be enhanced and developed, together with code developed inside the production code to cope with identifying those data elements as test elements in production. We already have such patterns and mechanisms in production for the credit card companies, as they have developed such mechanisms for vendors who want to integrate their systems into the credit card system, meaning things have to be tested in real life scenarios. Today you can introduce a test-like credit card into the credit card company's production system and do all activities on that card in order to test it.

Summary

Complex technologies and new development approaches that are being enhanced and introduced to the market today will greatly impact the testing methods, tools, processes, platforms, and skills required.

We should pay careful attention to those trends and be proactive in our training, and in innovating new tools, platforms, and processes to support them. We should discuss this worldwide in forums and discussion groups, involving the development engineers to give us feedback, and being closer to the business side so we can learn what our customers' requirements are and

what solutions they need. This will enable us to develop and enhance those areas and be ready on time to give an answer to those challenges from the testing and quality side. ■

> about the author

Alon Linetzki



Alon Linetzki, founder and managing director of Best-Testing, has been a coach and consultant in development, testing and quality assurance for 28 years.

Alon has been involved in supporting organizations to enhance the test engineer's professional

and personal skills, training test managers in optimization and test process improvement and optimizing their test operations, increasing ROI.

Alon's main domains of expertise are in Agile Testing and Transition to Agile, Exploratory Testing, Test Process Improvement and Optimization, Risk-Based Testing and Test Automation.

He is a part of the ISTQB® authors and review team for the ISTQB® Agile Tester Add-on certification, co-founded ISTQB® in Israel, leads the ISTQB® Partner Program and has established the SIGIST Israel.



The Magazine for
Professional Testers

**te testing
experience**
www.testingexperience.com

Subscribe to
the free online
edition!

Distributed Agile Pointless or Possible?

by Julie Calboutin & Helmut Holst

1. Management Summary

Most of the significant challenges facing IT project management span the well-known triad of time, cost and quality. Growing complexity and increasingly integrated solutions in turn exacerbate these challenges. In recent years, Agile approaches have often been employed to meet these challenges. The core tenets of Agile, however, include collocation and face-to-face collaboration – so how can the seemingly contradictory models of Agile and offshore be combined to deliver the best of both worlds?

Agile methodologies and variations on it have worked their way into projects around the world and, from those projects, the statistics have started surfacing. According to Scott Ambler's Agile Adoption Strategies Survey 2011, collocated Agile projects are as successful (34%) as near-collocated ones (34.5%), which in turn are only very slightly more successful than those involving far-collocated (including globally distributed) (32%) (Ambler and Gorans, November 2011). However, it is also becoming clear that an Agile approach does have advantages over traditional software development approaches. In fact, statistics show that the percentage of failures is decreasing: 55% of projects recorded successes in 2010 while 63% recorded successes in 2011. (Ambler S. W., 2011 IT Project Success Rates Survey Results, 2011)

Companies manage to work around the limitations of geographical space and time and, although they recognise that face-to-face collocation and collaboration is still the number-one choice and most effective method of working, the overall benefits of reduced costs and the 'follow-the-sun' working hours approach associated with distributed Agile methodology seem to be paying off.

SQS has been involved in a large number of Agile projects, many of which have been successful in combining offshoring with Agile practices. This white paper will present a number of the lessons learned from these engagements and share some of the practices which have led to successful offshoring within an Agile model.

2. Distributed Agile

2.1 Common Challenges

The most common challenges distributed teams (Agile or traditional) generally face are the following:

- Time zones and working hours
- Cultural and language differences

- Availability and access to tools
- File sharing
- Team dynamics
- Telephone dynamics

All these challenges are important to establishing the single element that is an essential ingredient for an efficient and successful project team: trust. Therefore, almost all of these challenges can be mitigated either partially or completely by mastering the greatest challenge of all: communication. As already discussed, co-located teams have the highest rate of success, but why?

- They communicate face to face
Pros:

- ☐ Highest collaboration level
- ☐ Richest communication level
- ☐ No loss of non-verbal communication
- ☐ Promotes self-organisation of the team
- ☐ Permanent participation of the entire team

Cons:

- ☐ Requires a collocated team
- They obtain instant feedback from team members
- They benefit from communication fidelity — the degree of accuracy between the meaning intended and the meaning interpreted (Petersen, 2007):
 - ☐ 55% of the meaning is conveyed by physical body language,
 - ☐ 38% is conveyed by culture-specific voice tonality, and
 - ☐ Only 7% of the meaning is conveyed by words.

Much of the focus around communication and working side by side is about building trust.

2.2 Practical Guide to Succeeding in Distributed Agile Teams

2.2.1 Communication solutions

This section covers the basic suggestions of how to overcome the communication challenges faced by a distributed Agile team. Although these are rather general suggestions, it should

be noted that implementing the appropriate communication model is absolutely critical to the success of a distributed Agile approach, given the nature of the iterative and continuous feedback approach that is the core of the methodology.

- Establish a synchronisation and communication plan:
 - Define how the client, local team, and distributed teams will communicate and maintain synchronisation
 - Define daily and distributed stand-ups, retrospectives and sprint review time

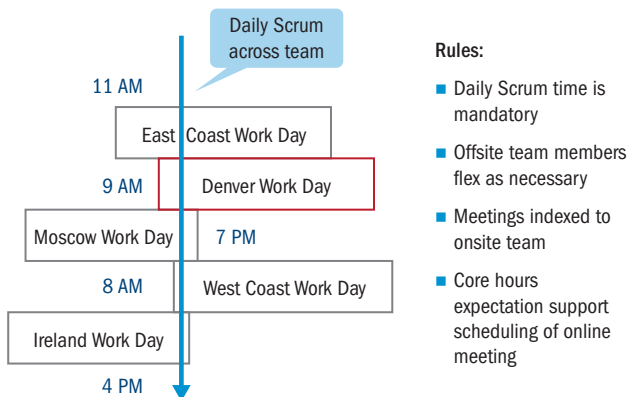


Figure 1. (Source: <http://scalingsoftwareagilityblog.com/wp-content/uploads/2007/12/daily-scrum.gif>)

- Use interactive communication software with a voice layer to assist in keeping all parties across distributed teams engaged, as well as being able to ask and answer questions easily. Also, talking is more efficient than typing – you can use hands-free headphones and web cameras to facilitate voice communication.
- Use interactive communication software with screen sharing capabilities for up-skilling and troubleshooting so a visual relationship can be established which helps to improve trust.
- Establish a central repository for project information which is permanently available to all team members and remains current, particularly for distributed teams that do not have a large time zone overlap. Ensure some sort of versioning is in place for project documentation in shared locations:
 - Shared drive.
 - Document management tools.
- Establish centralised wikis and discussion forums (knowledge base):
 - They allow dispersed team members to post questions and receive answers quickly from team experts

te testing
experience

Become an author for Testing Experience!

No. 25 “Crowd Testing”

Publication: March 2014

Deadline for article submissions: January 15, 2014

Submit your article for our next issue and share your experiences and knowledge with your peers.

More information at:

write.testingexperience.com



anywhere in the world; posts should be a searchable information source.

- Use a teleconference facility. If you are not using web cameras, try to introduce yourself each time before you start speaking until everyone recognizes each other's voices:

- It is ideal for distributed teams with overlapping hours.
- It provides a backup for colocated teams.
- It allows team members to interact directly.
- It allows permanent participation of the entire team.
- It enables blockers to be discussed and removed them immediately.

- Use a videoconference facility:

- It potentially enriches the communication experience.
- It allows team members to interact directly.
- It helps turn names into people.

- Use enterprise tools (Quality Center, Communicator, TeamForge, POD):

- Breeding synergy, transparency, productivity, and trust increases efficiencies across projects and organisations.
- When a tester updates an artefact, that update triggers a monitoring event which sends an email to everyone monitoring that artefact.

- Account for language differences:

- Keep sentences simple and concise and use common words – do not get creative by using the most obscure words in the dictionary. Develop a common low-level vocabulary where you understand one another and build from there.
Remember: in US English, someone who is 'blue' is sad – in German, 'blue' is 'blau', and someone who is 'blau' is drunk.

- Give everyone a chance to speak:

- The language barrier can make it more difficult for non-native speakers to step into the conversation and supplement other team members' ideas – Jean-Louis Marechaux shares his technique for engaging everyone on the team:
"I usually facilitate the sharing of ideas by calling on each person to give them a chance to speak and to make sure each person's contribution is captured. This is even more valuable when some team members speak a first language other than the one used in the meeting. The pause gives them time to translate their thoughts into words and to contribute to the conversation."

- This also ensures that anyone using teleconference facilities does not get left out.

- Confirm what team members understand:

- Ask leading questions or have members summarise in their own words to confirm their understanding is correct; typically, while one person summarises, others can quickly determine if their own understanding was correct or ask additional questions to clarify.

- Use a solid, proven distributed development environment

- In order to enable the team to focus on the communication challenges discussed above, it is critical that shared, distributed access to code, environments, data, and tools is established and working well. This means that from both a latency and access perspective the environments are fully available and proven.

- Initially execute three to four sprints with the entire team at the local site:

- It is advisable to have the offshore team travel to the onshore base site for a period of time and work together with the onshore team in order to prove the operating model before taking it offshore.
- It will at least take three to four sprints of two weeks each to build relations.
- Use the time to define norms together, as well as setting up frameworks, initial architecture, and design.
- This enables the distributed team to build a relationship with the client, and business processes and requirements are explained.

- Meet face to face to build trust:

- Budget in recurring face-to-face meetings between the client, local team, and distributed team.
- Shorter than the initial visit but should be more or less regular: e.g. a one-week trip every six months.
- Plan for potential visits of key people when a significant change is planned, like a new system design or a major refactoring.

- Establish a shared project vision:

- Participation in this activity by the whole team emphasises ownership of the project results.

- Establish a rigorous norming and chartering plan to achieve high quality:

- Determine a set of activities the team will perform to ensure and maintain high-quality software.
- Define a consensus-based design, coding standards, code reviews, Scrum-of-Scrums, pair programming, a source control philosophy, a defect tracking mechanism, and define 'ready' and 'done'.

- Use short sprints:
 - Short iterations ensure visibility of the distributed team's activities, and feedback can be given as quickly as possible.
- Employ a Scrum Master at all locations:
 - Most impediments will need to be addressed within the context and environment of each sub-team.
 - It is critical that the SM acts as an active coach for the entire team to embed the practices needed to support distributed Agile.
- Involve the entire team in the release planning, iteration planning, review, and retrospectives.
- Use multiple clocks showing different time zones on the wall.
- Know about local holidays for all of the distributed team members.
- Work with an offshore provider with a proven retention track record. The Agile delivery will be fundamentally undermined if the offshore personnel are regularly changing.
- Stick photos of the offshore team on the wall of the onshore office (and vice versa).
- Share social stories/updates from events.

2.2.2 Onsite coordinator

One of the methods to improve quality of communications with the offshore team is to have a dedicated person to coordinate and oversee its activities from onsite. This role is there to ensure the communication flow, act as liaison between the teams, and often interpret information from local to offshore languages. Even if both sides speak English fluently (e.g. outsourcing to India) there are lots of subtle differences in business lingo that need translation. Add to it logistical challenges – this person typically ends up working long, odd hours – and you realize that it is not an easy task to find some who can do it (Krym). The onsite coordinator must be briefed by and work closely with the product owner and will find the following characteristics and skills very useful:

- Open-mindedness to absorb domain and business knowledge quickly.
- Excellent communication skills.
- Accessibility to the distributed team to discuss business processes.
- Ability to assemble a training and knowledge transfer manual for possible distributed on-boarding.

2.2.3 Practical tools advice

Distributed project planning tools are well developed to support asynchronous Agile operation.

However, the following restrictions apply:

1. Multi-cultural aspects and languages are generally not supported; English is the dominant language.
2. Data exchange between Agile planning tools and MS project is problematic.
3. Existing tools hardly maintain synchronous Agile planning meetings.

According to Xin Wang, an analysis of existing tools shows that nearly all of them focus on and support asynchronous features such as progress tracking and card management. The next step will need to support synchronous project planning meetings, setting up ubiquitous project planning environments, and enabling data exchange between different Agile tools and/or with non-Agile planning tools.

2.3 Case Study and Personal Experience

One offshore Agile project which has now been running for three years enjoys fairly mature processes. Key to the success of this project – as has already been indicated above – are communication and tools.

The following factors have helped the project to succeed:

1. Daily stand-ups both in the UK and US, followed by a smaller stand-up with a smaller group of UK and US project leads.
2. An online Agile project management tool (Rally) to manage and track tasks and defect management.
3. Extensive use of Skype features like Group Chat, which keeps everyone in the loop.
4. Stories, requirements/acceptance criteria defined and agreed in advance of planning.
5. Clear and unambiguous acceptance criteria such as Given, When, Then.
6. Consistent sprint schedules (2 weeks) proved most productive.
7. Three-way (BA, Development, Tester) discussions and meetings on stories.
8. A daily defect management meeting to discuss and triage defects.
9. In some cases, flying developers/testers over to be in one place for release planning/pre-releases proved effective.

Due to experience with complex functionality where rework was required, our challenge is to get the functionality right the first time round. In order to address this issue, we are increasing end user involvement in sessions with business analysts.

3. Conclusion and Outlook

“Action is the foundational key to all success.”
– Pablo Picasso

SQS experience of offshore Agile delivery has taught us the following:

- The reality is that working with distributed teams can be a nightmare if badly structured. The challenges increase as the type of distribution spreads from collocated, to distributed with overlapping working hours, to distributed with no overlap in working hours.
- One important key to success as a distributed team is to ensure a high commitment level from all team members, and the best way to achieve this is to give them ownership over how they will work.
- Retrospectives help teams evaluate whether communication is working for them, as well as being responsive to their stakeholders' needs.
- Teams should feel continually free to adjust any of the approaches and solutions to better suit their needs.
- Tight collaboration and coordination is imperative.
- Tools are critical, but they are not the only answer – having good processes in place is indispensable.
- Technology will help overcome most obstacles, therefore code review, wikis, discussion forums, bug tracking, requirements tracking, and continuous integration are essential.
- An integrated platform to support synergies, transparency, productivity, and trust increases efficiencies across projects.

Bibliographical References

- [1] Ambler, S. (n.d.). *Dr. Dobb's*. Retrieved 2012, from www.ddj.com/architect/200001986
- [2] Ambler, S. W. (n.d.). Retrieved from www.ambysoft.com/scottAmbler.html
- [3] Ambler, S. W. (2011, 10). *2011 IT Project Success Rates Survey Results*. Retrieved 01 04, 2013, from www.ambysoft.com/surveys/success2011.html
- [4] Beck, K. (n.d.). *Agile Manifesto*. Retrieved from agilemanifesto.org/history.html
- [5] Highsmith, J. (n.d.). *Agile Manifesto*. Retrieved from agilemanifesto.org/history.html
- [6] Klocwork. (n.d.). Retrieved from www.klocwork.com/blog/2010/02/agile-adoption-an-update/
- [7] Krym, N. (n.d.). *The Myth of the Onsite Coordinator*. Retrieved from Pragmatic Outsourcing: pragmaticoutsourcing.com/2008/11/20/the-myth-of-the-onsite-coordinator/
- [8] Mthembu, N. (2012, 10 03). *My experience working in a distributed Agile project*. South Africa.
- [9] Petersen, G. (2007, 10 18). *Mythos: 93% der Kommunikation ist nonverbal – My Skills*. Retrieved 01 08, 2013, from blog.my-skills.com/2007/10/18/mythos-93-der-kommunikation-ist-nonverbal.html
- [10] Scott W. Ambler + Associates. (2008, 06). *Agile Adoption Rate Survey Results: February 2008*. Retrieved 01 08, 2013, from www.ambysoft.com/surveys/agileFebruary2008.html
- [11] VersionOne. (n.d.). Retrieved from www.versionone.com
- [12] Wikipedia. (n.d.). Retrieved from www.google.co.za/url?url=http://en.wikipedia.org/wiki/Self-organised&rct=j&sa=X&ei=13liUI_pHpSLhQel-toGwCQ&ved=OCDEQngkAA&q=self+organised&usg=AFQjCNGA5TlzKfNjglPjhelgclKklEv16Q
- [13] Wikipedia. (n.d.). Retrieved from [en.wikipedia.org/wiki/Scrum_\(development\)](http://en.wikipedia.org/wiki/Scrum_(development))
- [14] Xin Wang, F. M. (n.d.). Retrieved from ebe.cpsc.ucalgary.ca/uploads/Publications/Wangetal2010.pdf

> about the authors

Julie Calboutin



Julie Calboutin has a degree in computer science and physics and has been a consultant at SQS South Africa since 2006. She has 17 years of experience in software development and testing. Julie has successfully delivered complex systems across multiple industries in various countries across Europe as well as in South Africa. Julie is a Certified Agile Tester and South Africa's second Certified Agile Tester Trainer. Julie's primary responsibilities include test management, consultant coaching, and delivery assurance. This is backed up with the ISTQB Advanced Test Manager certification.

Helmut Holst



Helmut Holst has a degree in German language, history and political science from the University of Lüneburg and has been a senior consultant for SQS for 6 years. During his time in the research and development group he has been instrumental in developing the learning zone modules for methodology PractiQ. His achievement has been the development of a malaria data base for Africa under medical council. He is fully bilingual and travels professionally between Europe and Africa. His practical experience includes banking, shipping, real estate, and telecommunication. Helmut is a contributing member of the Test Management, Managed Services, Offshore, Global Delivery Model, Automation, Agile, and the SAP innovation groups.

Refactoring – to Sustain Application Development Success in Agile Environments

by Narayana Maruvada

The term “refactoring” was originally coined by Martin Fowler and Kent Beck which refers to “a change made to the internal structure of software to make it easier to understand and cheaper to modify without altering its actual observable behavior i.e. it is a disciplined way to clean up code that minimizes the chances of introducing bugs and also enables the code to be evolved slowly over time and facilitates taking an iterative and incremental approach to programming and/or design”. Importantly, the underlying objective behind refactoring is to give thoughtful consideration and improve some of the essential *non-functional attributes* of the software. So, to achieve this, the technique has been broadly classified into following major categories:

| | Technique | Description |
|---|--|---|
| 1 | Code Refactoring (clean-up) | It is intended to remove the unused code, methods, variables etc. which are misleading. |
| 2 | Code Standard Refactoring | It is done to achieve quality code. |
| 3 | Database Refactoring (clean-up) | Just like code refactoring, it is intended to clean or remove the unnecessary and redundant data without changing the architecture. |
| 4 | Database schema and design Refactoring | This includes enhancing the database schema by leaving the actual fields required by the application. |
| 5 | User-Interface Refactoring | It is intended to change the UI without affecting the underlying functionality. |
| 6 | Architecture Refactoring | It is done to achieve modularization at the application level. |

Refactoring is actually a simple technique where you make *structural* changes to the code in small, independent and safe steps, and test the code after each of these steps just to ensure that you have not changed the behavior – i.e. the code still works the same, but just looks different. Nevertheless, refactoring is intended to fill in some short-cuts, eliminate duplication and dead code, and help ensure the design and logic have been made very clear. Further, it is equally important to understand that, although refactoring is driven by certain good characteristics and shares some common attributes with debugging and/or optimization, etc., it is actually different because

- Refactoring is not all about fixing any bugs.
- Again, *optimization* is not refactoring at all.
- Likewise, revisiting and/or tightening up error handling code is not refactoring.
- Adding any defensive code is also not considered to be refactoring.
- Importantly, tweaking the code to make it more testable is also not refactoring.

Refactoring Activities – Conceptualized

The refactoring process generally consists of a number of distinct activities which are dealt with in chronological order:

- Firstly, identify where the software should be refactored, i.e. figure out the *code smell* areas in the software which might increase the risk of failures or bugs.
- Next, determine what refactoring should be applied to the identified places based on the list identified.
- Guarantee that the applied refactoring preserves the behavior of the software. This is the crucial step in which, based on the type of software such as *real-time*, *embedded* and *safety-critical*, measures have to be taken to preserve their behavior prior to subjecting them to refactoring.
- Apply the appropriate refactoring technique.
- Assess the effect of the refactoring on the quality characteristics of the software, e.g. *complexity*, *understandability* and *maintainability*, and of the process, e.g. productivity, cost and effort.
- Ensure the requisite consistency is maintained between the refactored program code and other software artifacts.

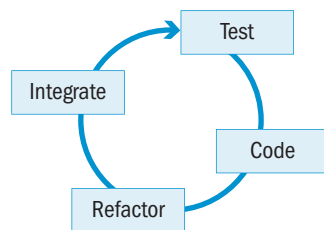
Refactoring Steps – Application/System Perspective

The points below clearly summarize the important steps to be adhered to when refactoring an application:

1. **Firstly, formulate the unit test cases for the application/system** – the unit test cases should be developed in such a way that they test the application behavior and ensure that this behavior remains intact even after every cycle of refactoring.
2. **Identify the approach to the task for refactoring** – this includes two essential steps:
 - Finding the problem – this is about identifying whether there is any code smell situation with the current piece of code and, if yes, then identifying what the problem is all about.
 - Assess/Decompose the problem – after identifying the potential problem assess it against the risks involved.
3. **Design a suitable solution** – work out what the resultant state will be after subjecting the code to refactoring. Accordingly, formulate a solution that will be helpful in

transitioning the code from the current state to the resultant state.

4. **Alter the code** – now proceed with refactoring the code without changing the external behavior of the code.
5. **Test the refactored code** – to ensure that the results and/or behavior are consistent. If the test fails, then rollback the changes made and repeat the refactoring in different way.
6. **Continue the cycle** with the aforementioned steps (1) to (5) until the problematic/current code moves to the resultant state.



So, having said about refactoring and its underlying intent, it can be taken up as a practice and can be implemented safely with ease because the majority of today's modern IDEs (integrated development environments) are inbuilt and equipped with various refactoring tools and patterns which can be used readily to refactor any application/business-logic/middle-tier code seamlessly. However, the situation may not be the same when it comes to refactoring a database, because database refactoring is conceptually more difficult when compared to code refactoring since with code refactoring you only need to maintain the *behavioral semantics*, whereas with database refactoring you must also maintain *information semantics*.

Refactoring a Database – a Major and Typical Variant of Refactoring

"A database refactoring is a process or act of making simple changes to your database schema that improves its design while retaining both its behavioral and informational semantics. It includes refactoring either *structural* aspects of the database such as table and view definitions or *functional* aspects such as stored procedures and triggers etc. Hence, it can be often thought of as the way to normalize your database schema."

For a better understanding and appreciation of the concept, let us consider the example of a typical database refactoring technique named *Split Column*, in which you replace a single table column with two or more other columns. For example, you are working on the PERSON table in your database and figure out that the DATE column is being used for two distinct purposes. a) to store the birth date when the person is a customer and b) to store the hire date when the person is an employee. Now, there is a problem if we have a requirement with the application to retrieve a person who is both customer and employee. So, before we proceed to implement and/or simulate such new requirement, we need to fix the database schema by replacing the DATE column with equivalent *BirthDate* and *HireDate* columns. Importantly, to maintain the behavioral

semantics of the database schema we need to update all the supporting source code that accessed the DATE column earlier to now work with the newly introduced two columns. Likewise, to maintain the informational semantics we need to write a typical migration script that loops through the table, determines the appropriate type, and then copies the existing date data into the appropriate column.

Classification of Database Refactoring

The database refactoring process is classified into following major categories:

1. **Data quality** – the database refactoring process which largely focuses on improving the quality of the data and information that resides within the database. Examples include introducing column constraints and replacing the type code with some boolean values, etc.
2. **Structural** – as the name implies this database refactoring process is intended to change the database schema. Examples include renaming a column or splitting a column etc.
3. **Referential Integrity** – this is a kind of structural refactoring which is intended to refactor the database to ensure referential integrity. Examples include introducing cascading delete.
4. **Architectural** – this is a kind of structural refactoring which is intended to refactor one type of database item to another type.
5. **Performance** – this is a kind of structural refactoring which is aimed at improving the performance of the database. Examples include introducing *alternate index* to fasten the search during data selection.
6. **Method** – a refactoring technique which is intended to change a method (typically a stored procedure, stored function or trigger, etc.) to improve its quality. Examples include renaming a stored procedure to make it easier to refer and understand.
7. **Non-Refactoring Transformations** – this type of refactoring technique is intended to change the database schema that, in turn, changes its semantics. Examples include adding new column to an existing table.

Why isn't Database Refactoring Easy?

Generally, database refactoring is presumed to be a difficult and/or complicated task when compared to code refactoring. not just because there is the need to give thoughtful consideration to the behavioral and information semantics, but due to a distinct attribute referred to as *coupling*. The term coupling is understood to be *the measure of the degree of the dependencies between two entities/items*. So, the more coupling there is between entities/items, the greater the likelihood that a change in one will require a change in another. Hence, it is understood that coupling is the root cause of all the issues when it comes to database refactoring, i.e. the more things that your database is coupled to, the harder it is to refactor.

Unfortunately, the majority of relational databases are coupled to a wide variety of things as mentioned below:

- Application source code
- Source code that facilitates *data loading*
- Code that facilitates *data extraction*
- Underlying Persistent layers/frameworks that govern the overall application process flow
- The respective database schema
- Data migration scripts, etc.

Refactoring Steps – Database Perspective

Generally, the need to refactor the database schema will be identified by an application developer who is actually trying to implement a new requirement or fix a defect. Then the application developer describes the required change to the concerned DBA of the project and then refactoring begins. Now, as part of this exercise, the DBA will typically work through all or a few of the following steps in chronological order:

1. **Most importantly, verify whether database refactoring is required or not** – this is the first thing that the DBA does, and it is where they will determine whether database refactoring is needed and/or if it is the right one to perform. Now the next important thing is to assess the overall impact of the refactoring.
2. **If it is inevitable, choose the most appropriate database refactoring** – this important step is about having several choices for implementing new logic and structures within a database and choosing the right one.
3. **Deprecate the original schema** – this is not a straightforward step, because you cannot simply make a change to the database schema instantly. Instead, adopt an approach that will work with both the old and the new schema in parallel for a while to provide the required time for the other team to both refactor and redeploy their systems.
4. **Modify the schema** – this step is intended to make the requisite changes to the schema and ensure that the necessary logs are also updated accordingly, e.g. *database change log* which is typically the source code for implementing all database schema changes and *update log* which contains the source code for future changes to the database schema.
5. **Migrate the data** – this is the crucial step which involves migrating and/or copying the data from old versions of the schema to the new.
6. **Modify all related external programs** – this step is intended to ensure that all the programs which access the portion of database schema which is for the subject of refactoring must be updated to work with the new version of the database schema.
7. **Conduct regression test** – once the changes to the application code and database schema have been put in

place, then it is good to run the regression test suite just to ensure that everything is right and working correctly.

8. **Keep the team informed about the changes made and version control the work** – this is an important step because the database is a shared resource and it is minimally shared by the application development team. So, it is the prime responsibility of the DBA to keep the team informed about the changes made to the database. Nevertheless, since database refactoring definitely includes some DDLs, change scripts, data migration scripts, data models related scripts, test data and its generation code, etc., all these scripts have to be put under configuration management by checking them into a version control system for better versioning, control, and consistency.

Once the database schema has been refactored successfully in the application development sandbox (a technical environment where your software, including both your application code and database schema, are developed and unit tested), the team can go ahead with refactoring the requisite Integration, Test/QA, and Production sandboxes as well, to ensure that the changes introduced are available and uniform across all environments.

Key Benefits of Refactoring

From a system/application standpoint, listed below are summaries of the key benefits that can be achieved seamlessly when implementing the refactoring process in a disciplined fashion:

- Firstly, it improves the overall software extendability.
- Reduces and optimizes the code maintenance cost.
- Facilitates highly standardized and organized code.
- Ensures that the system architecture is improved by retaining the behavior.
- Guarantees three essential attributes: readability, understandability, and modularity of the code.
- Ensures constant improvement in the overall quality of the system. ■

> about the author

Narayana Maruvada



Narayana Maruvada is a computer science and engineering graduate, currently working as System Analyst – QA with ValueLabs. He has more than 7 years of experience working on both developing and testing web-based applications. A major area of work and expertise lies in testing the applications and products that are built on an open-source technology stack for different domains. He is keenly interested in assessing the system's functional and non-functional attributes; investigating and implementing new testing tools, techniques and methodologies; and contributing to augmenting best practices and quality standards for test process improvement.



The Conference for Agile Developers



"Agile Effectiveness: Craftsmen, Hackers, DevOps and Common Sense"

Stay tuned - program coming soon!

Join our keynote and tutorial speakers for Agile Dev Practices 2014:



*Marc-André
Cournoyer*



*Steve
Freeman*



*Bob
Galen*



*Sandro
Mancuso*



*Andy
Palmer*



*Nat
Pryce*



*Dr. Axel
Rauschmayer*



*Vaughn
Vernon*

May 26-28, 2014 in Potsdam/Berlin, Germany

www.agiledevpractices.com

Sponsors & Exhibitors 2014



How to Test Refactoring

by Jeroen Mengerink

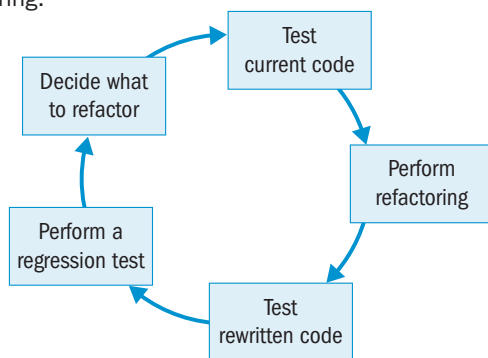
A fundamental part of the Agile methodology is refactoring: rewriting small sections of code to be functionally equivalent but of better quality. Don't forget to test the refactoring! What do you test? The answer is simple: you test whether the code really is functionally equivalent.

To test the rewritten code, you use the unit tests that accompanied the original code. But does unit testing alone prove that you really have functionally equivalent code? No! While refactoring, developers often change more than just the complexity and quality of the code. A tester's nightmare ... It appears to be a small change, but the code is quite likely used in several parts of the solution. So you must perform a regression test after testing the changed code itself.

First I will describe how to test the current and rewritten code with unit test. I have identified three scenarios that occur in practice. The code that needs refactoring has:

- no unit tests;
- bad unit tests;
- good unit tests.

After these scenarios, I will go into the regression test and explain the importance of proper regression testing while refactoring.



Unit test the current and rewritten code

Unit tests are tests to test small sections of the code. Ideally each test is independent, and stubs and drivers are used to get control over the environment. Since refactoring deals with small sections of code, unit tests provide the correct scope.

Refactor code that has no existing unit tests

When you work with very old code, in general you do not have unit tests. So can you just start refactoring? No, first add unit tests to the existing code. After refactoring, these unit tests should still hold. In this way you improve the maintainability of the code as well as the quality of the code. This is a complex task. First you need to find out what the functionality of the

code is. Then you need to think of test cases that properly cover the functionality. To discover the functionality, you provide several inputs to the code and observe the outputs. Functional equivalence is proven when the code is input/output conformant to the original code.

Refactor to increase the quality of the existing unit tests

You also see code which contains badly designed unit tests. For example, the unit test verifies multiple scenarios at once. Usually this is caused by not properly decoupling the code from its dependencies (Code sample 1). This is undesirable behaviour because the test must not depend on the state of the environment. A solution is to refactor the code to support substitutable dependencies. This allows the test to use a test stub or mock object. As shown in Code sample 2, the unit test is split into three unit tests which test the three scenarios separately. The rewritten code has a configurable time provider. The test now uses its own time provider and has complete control over the environment.

Treat unit tests as code

The last situation deals with a piece of code which has good unit tests. Just refactor and then you are done, right? Wrong! When you refactor this code, the test will pass if you refactor correctly. But do not forget to check the validity of the tests. You might think the tests are good, but the unit tests are code too. Every refactor action incorporates a check, and possibly a refactor, of the unit tests.

Perform a regression test

After unit testing the code, you need to verify if the code works in the solution's context. Remember: In Agile you must provide business value. To show the value, you need to perform a test that relates to the business. A regression test is designed to test the important flows through the solution. And these flows embody the business value. Do you run a complete regression test after each time you refactor? This depends on the risks and on the scalability of the regression test.

Create a scalable regression test

The use case is a common way to describe small parts of functionality. This is a great way to partition your regression test. Create a small set of regression test cases to cover a use case. When you use proper version management for the code, it is easy to see which part of the code belongs to which use case. Whenever a section of code is changed, you can see to which use case it belongs and then execute the regression tests for that use case.

However, when code is reused (another good practice), you target a group of use cases. I generally use mindmaps for track-

ing dependencies within my projects. The mindmaps provide insight in which code is used by which use cases. This requires a disciplined development team. When you reuse existing code, you need to update the mindmap!

Expand the scope of the regression test

Do you test enough when you scale the regression test to the scope determined in the mindmap? No, the regression test serves a larger goal. You check if the (in theory) unaffected areas of the solution are really unaffected. So you test the part that is affected by the refactoring and you test the main flows through the solution. The flows that provide value to the customer are the most important.

Refactoring requires testing

Every change in the code needs to be tested. Therefore testing is required when refactoring. You test the changes at different levels. Since a small section of code is changed, unit testing seems the most fitting level. But do not forget the business value! Regression testing is of vital importance for the business.

- Refactoring requires testing.
- Testing refactoring requires a good understanding of the code.
- A good understanding of the code requires a disciplined development team.
- A disciplined development team refactors.

Code sample 1: Unit test depending on the environment

From <http://xunitpatterns.com>.

```
1 public void testDisplayCurrentTime_whenever() {
2     // fixture setup
3     TimeDisplay sut = new TimeDisplay();
4     // Exercise sut
5     String result = sut.getCurrentTimeAsHtmlFragment();
6     // Verify outcome
7     Calendar time = new DefaultTimeProvider().getTime();
8     StringBuffer expectedTime = new StringBuffer();
9     expectedTime.append("<span class=\"tinyBoldText\">");
10    if ((time.get(Calendar.HOUR_OF_DAY) == 0) &&
11        (time.get(Calendar.MINUTE) <= 1)) {
12        expectedTime.append("Midnight");
13    } else if ((time.get(Calendar.HOUR_OF_DAY) == 12) &&
14        (time.get(Calendar.MINUTE) == 0)) { // noon
15        expectedTime.append("Noon");
16    } else {
17        SimpleDateFormat fr = new SimpleDateFormat("h:mm a");
18        expectedTime.append(fr.format(time.getTime()));
19    }
20    expectedTime.append("</span>");
21    assertEquals( expectedTime, result);
22 }
```

Code sample 2: Independent unit tests

From <http://xunitpatterns.com>.

```
1 public void testDisplayCurrentTime_AtMidnight()
2     throws Exception {
3     // Fixture setup:
4     TimeProviderTestStub tpStub = new
5         TimeProviderTestStub();
```

```
4     tpStub.setHours(0);
5     tpStub.setMinutes(0);
6     // Instantiate SUT:
7     TimeDisplay sut = new TimeDisplay();
8     sut.setTimeProvider(tpStub);
9     // Exercise sut
10    String result = sut.getCurrentTimeAsHtmlFragment();
11    // Verify outcome
12    String expectedTimeString = "<span class=\"tinyBoldText\">Midnight</span>";
13    assertEquals("Midnight", expectedTimeString, result);
14 }
15 public void testDisplayCurrentTime_AtNoon()
16     throws Exception {
17    // Fixture setup:
18    TimeProviderTestStub tpStub = new
19        TimeProviderTestStub();
20    tpStub.setHours(12);
21    tpStub.setMinutes(0);
22    // Instantiate SUT:
23    TimeDisplay sut = new TimeDisplay();
24    sut.setTimeProvider(tpStub);
25    // Exercise sut
26    String result = sut.getCurrentTimeAsHtmlFragment();
27    // Verify outcome
28    String expectedTimeString = "<span
29        class=\"tinyBoldText\">Noon</span>";
30    assertEquals("Noon", expectedTimeString, result);
31 }
32 public void testDisplayCurrentTime_AtNonSpecialTime()
33     throws Exception {
34    // Fixture setup:
35    TimeProviderTestStub tpStub = new
36        TimeProviderTestStub();
37    tpStub.setHours(7);
38    tpStub.setMinutes(25);
39    // Instantiate SUT:
40    TimeDisplay sut = new TimeDisplay();
41    sut.setTimeProvider(tpStub);
42    // Exercise sut
43    String result = sut.getCurrentTimeAsHtmlFragment();
44    // Verify outcome
45    String expectedTimeString = "<span
46        class=\"tinyBoldText\">7:25 AM</span>";
47    assertEquals("Non special time", expectedTimeString,
48        result);
49 }
```

> about the author

Jeroen Mengerink



Jeroen works as a test consultant for Polteq. In addition to his work for clients, he is involved in various test innovations. His main area of expertise is Agile, for which he is the person to talk to within Polteq. Jeroen teaches several test courses, e.g. about Agile, SOA and Cloud, and is a teacher of the Certified Agile Tester course (CAT). He is co-author of the book and approach Cloutest® on how to test when cloud computing is involved. He has contributed as a speaker to a number of internal and external events for Polteq and its clients. In several international assignments he has presented the results of TPI assessments to a variety of senior management. He has presented several times at events like ChinaTest, Eurostar and TestNet on a large variety of subjects.

Twitter: @angusvb

Refactoring – “To Be or Not to Be”

by *Rashmi Wadhawan*

Refactoring is about changing the internal structure of an application without changing its interaction with the outside world. Technical teams struggle to justify the need for refactoring in their existing application.

From the business stand point, refactoring is an excuse to spend more time on existing applications without adding any business value. “Do it right the first time” is the mantra given by the business whenever we approach them for technical debt handling or refactoring sprints. Return on investment is of utmost importance to the business. Refactoring does not show any apparent monetary gain as it does not include any business feature addition in the existing codebase.

The following are the top two questions asked by management when you submit your proposal for refactoring the design of existing code:

“Will there be measurable performance gains?”

“Will this application handle more load than it is currently handling?”

Justifying the refactoring task might be very difficult, but not impossible. Here are the tips for justifying the need for refactoring.

1. Future business changes will require less time. Refactoring will not give an immediate return but, in the long run, adding features will be less expensive as the code will become easier to maintain. Before refactoring, the code is fit for machine consumption but after refactoring it is fit for human as well as machine consumption.
2. Bugs will be fixed during refactoring. Hidden bugs or logics embedded in complicated unnecessary loops will be exposed, which might result in fixing some longstanding non-reproducible issues.
3. The current application will have a longer life. Prevention is better than cure. Refactoring can be considered to be a prevention exercise which will help to optimize the structure of the application for future enhancements.
4. There might be performance gains. You cannot promise any apparent or measurable performance gain. But if you are planning to do refactoring to achieve some performance gain, then you should have measurable counters showing the performance of the current app before you start refactoring. And after each change the performance counters should be recalculated to check the optimization.

5. Refactoring may result in a reduction in the lines of code, making it less expensive to maintain in the long run. During refactoring of your algorithm, you should follow the DRY (Don't Repeat Yourself) principle. Any application that has survived for 6 months to 1 year will have ample places to remove duplication of code.

While refactoring looks quite fancy and is required for most software applications to work efficiently in the long run, it is not recommended for applications without proper safety nets, as changing the design and code after the application has gone live requires us to have a solid foundation of good software development practices. The following are a few prerequisites that should be built in to your process if you would like to do refactoring.

1. **Unit test/behavior-driven development suite.** Opening the engine of a running car to fix something when you do not know what it is requires a lot of courage. After fixing and oiling some units in the car, we still want the car to work in the same way it did when it was not performance-tuned. The expectation that whatever was working before refactoring is still working will be what the business people expect and, to ensure that, we will need some solid evidence to make them believe that the application still works in the same way but the underlying engine/framework has been optimized. This suite will act as a safety net after applying major changes to the application. But this courage to apply refactoring can become very expensive if the safety net of an automated test suite is not in place.
2. **Configuration management tool.** This will help you to avoid OOPS while doing refactoring. You might tag/label your build before going ahead with a change that requires changes in multiple files. If, after checking in, your regression test suites fail or the testing team reports a critical business case failure, the configuration management tool will come to your rescue by giving you the power to reverse the refactoring changes.
3. **Pair Programming.** Two pair of eyes and double brain power will help you make fewer mistakes. Pair programming is an extremely powerful tool in coding, but in refactoring it becomes invaluable as it helps you to think of many scenarios that you could miss when working alone.
4. **Refactoring Tool Set.** Developers do not use the full potential of the refactoring tools available on the market. This might be due to a lack of knowledge or pressure of

timelines. During refactoring, these tools are extremely helpful and valuable as they reduce the chances of introducing an error when making big changes.

| Tool | Technology |
|-------------------------------|-------------|
| ReSharper Addon Visual Studio | .Net |
| XCode | Objective C |
| IntelliJ IDEA | Java |

5. Keep refreshing your refactoring principles – go through the bible of refactoring i.e. Martin Fowler's book on refactoring. The rules mentioned by Martin Fowler in 1999 are still applicable and fresh irrespective of the language or development environment. Keep reading and reapplying the principles in the technology you are working on.

Refactoring using the right tools and good software development practices will be a boon for any application's long life and sustenance. Refactoring is an opportunity to solidify the foundation of an existing application that might have become weaker after adding a lot of changes and enhancements. If you are making changes to the same piece of code for the third time, it means there is some technical debt that you have created and there is a need to refactor this code. ■

> about the author

Rashmi Wadhawan



Rashmi Wadhawan is Head of Quality Assurance and Centre of Excellence at GrapeCity India and has 12 years of experience in the software industry. She is a certified Scrum Master, practicing the agile Scrum process in her projects. Prior to GrapeCity, she worked with Globallogic. She is an active member of the process improvement group at GrapeCity and is proud to be associated with one award winning product. She has played all kinds of roles in the software Industry including business analyst, technical lead, and test architect.



Díaz Hilterscheid

Testing for Developers

Whilst training for testers has made great progress in recent years – alone in Germany there are more than 10,000 certified testers – the role of the developer in software testing is mostly underestimated; they are often the driving force in the area of component testing. For these reasons it is important that also developers receive basic knowledge in the central themes of software testing.

As a result Díaz & Hilterscheid has created the two-day course "Testing for Developers" on the basis of the internationally recognized ISTQB® Certified Tester training. The first day covers the fundamentals of software testing, including the terminology used,

the test process and its integration into the software development process, and the various test levels and testing types. The second day the techniques of static testing are covered and specification-based test design techniques are demonstrated, with exercises for deeper understanding. Finally, the principles of risk-based testing are covered and the principal aspects of defect management taught.

After completion of the course, developers are able to construct systematic test cases by themselves and can execute developer tests to achieve the test completion criteria. In addition, they can

use the necessary terminology in order to confer with system and acceptance testers. In this way an optimization of the entire test process is possible.

For current training dates, please visit our website or contact us:

Díaz & Hilterscheid Unternehmensberatung GmbH
Kurfürstendamm 179
10707 Berlin
Germany

Phone: +49 (0)30 74 76 28-0
Fax: +49 (0)30 74 76 28-99

E-mail: training@diazhilterscheid.com
Website: training.diazhilterscheid.com

Refactoring or the Prevention of ...

by *Daniël Wiersma*

As long as we can remember, humans have been using refactoring to make things better. The first wheel was made of a round stone that later became wood, then steel, and now carbon fiber. The same is applicable to the computer. Once it could fill a living room, now it is in your pocket. The software we write undergoes the same evolution. Let me explain this. In 1842 some people started working on the Analytical Engine to calculate Bernoulli numbers [1]. They wrote down a set of notes that specified in complete detail a calculation method. These notes evolved to punch cards and, in 1890, they agreed on a standard for using this system. So you could say it was the first standard program language.



Let's take a leap in time to the 50's. In 1950 we started out with three widespread modern programming languages whose descendants are still in use today. With the rise of the internet in the 90's, new languages developed such as Visual Basic, Java and PHP. Over time we have acknowledged Open Source as a developmental philosophy for languages, including the GNU compiler collection that brought us universal languages such as Python, Ruby, and Squeak. This made it possible for great minds to continue the evolution of software languages. With all these steps over time, we also became dependent on these software languages to let our devices work.

The great minds that worked on the evolution of the language from punch card to new offspring or simpler notation have made

great steps. Of course, nowadays our need for functionality is far greater than back in the days of the punch card. With all the new languages, the one thing that needed to remain unchanged was the functionality. So, for testers, this evolution means several things. We had to use Rapid Software Testing to get up to speed with the developers and learn about the new possibilities for testers to work with these new languages and, of course, the other side effect was the large amount of work for testers.

Refactoring

Now to this term "refactoring" – what does it mean? If you look it up in Wikipedia you will see that they have added the word "code" and, since I am constantly talking about software, it is a logical add-on. So you will find the following definition: *"Code refactoring is a 'disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.'"* When I read this it sounds a bit like "remixing a song", so I also looked up the definition of "remix": *"A remix is a song that has been edited to sound different from the original version. For example, the pitch of the singer's voice or the tempo might be changed, it might be made shorter or longer, or it might have the voice duplicated to create a duet."*

While in software development we do not want the functionality to change, in music you do not want the essence of the music to change when you do a remix. To my standards, a good remixed song is one where you still feel its emotional tension but everything else can be completely different. So, for example, today's artists sing songs from back in the day. Or make a completely new song like Moby did with the song *Natural Blues* [2]. A few years ago, Moby used elements from various old songs in newly constructed songs and created a completely new album. Just like in software development, the elements of this song are old (1937) but made relevant again on the album.

Definition of done

Let's get back to the software and refactoring. Since we all work with an agile mindset, we all know about the definition of done (DOD) and which items should be in there. But refactor-

ing is not always addressed there, although I think it should be. Most of the time we start a project that is the next step of development on from an existing baseline. So, therefore, a lot of code has already been written, some going back to the 70's. Now we are going to work on that code in an agile way and we speed up the writing of code, since we have to be finished in a sprint of, say, 2 weeks. Most of the time the DOD does not contain the element of hygiene on the code baseline or any project-related hygiene. So you create a technical debt by not including hygiene in your DOD.

My experience is that over time, during for example the second sprint, while adding new functionality to this 70's code, you will notice a change. Most of the time it is the user expert or tester who will notice a behavior change in the functionality. The most obvious one is the response time of the functionality, since we are adding code and functionality to the 70's version of our application. So, instead of working in the sprint on new functionality, we are going to take time to refactor the code to get the response time back to normal.

Testing

For testers, this means we have to test this functionality over and over again. Luckily for us, developers are human like us and they make mistakes like we do. So, often you will see that, after you have had the refactored code delivered, you need test more than once. Earlier we said that the functionality should not change, so we should use test automation to determine whether the software is still working. So when the team starts refactoring the code, the tester should use the current version of the software to make test cases.

Since developers are not the only ones causing a refactoring sprint, we should also take a look at the testers. They often use test cases or test scenarios to write down how they are going to determine whether the software is still working as it should. Often I see a lot of test cases in test databases that are not relevant any more. Either the functionality is not there any more or the process has changed and this test case cannot be used. So this means that the test cases could use some refactoring. If you are out of luck as a team, the tester just uses these test cases to do testing and then says: "Well, I think it's still not fixed, because my test doesn't work". But when you look in detail at the scenarios used, you will see that the test cases have not been updated, but the functionality has changed.

For test cases the same rules apply – don't use old stuff, always review, and make it better or create new. Of course, the core of testing never changes. But, as a tester, you should also evolve, use, and learn new ways of testing like Rapid Software Testing or Exploratory Testing. When describing test cases and scenarios, make sure that you don't write too much because when it comes to reviewing before reusing, it will take you hours to change your approach.

Preventing refactoring?

Can we prevent refactoring? I think there is not a single answer. If you start with a blank canvas, there is no legacy code, so it is easy to write beautiful clean code. If you are fully aware of the responsibly you have as developer and as a tester to keep it clean and simple, you are on the right path. My advice would be take code and test case reviewing very seriously and to make it part of the preparation of your sprint. If you first take a good look at what you have, you can take action to prevent a refactoring sprint from happening.

Remixing songs

I would like to take you back again to the remixing of songs. I want to share some more examples of refactored (remixed) songs because I am a real music geek. Let's start with *Be Thankful for What You Got* – William DeVaughn (1972) [2]. This song is one of my favorite songs, but I know a more recent version since I wasn't yet born in 1972. It was remixed by Massive Attack in 1991. Massive Attack liked the track so much that they stayed very close to the original. They just added a beat and hired another singer. Another song that has had a little more work is a song by Phoenix – *1901* (2009) [2]. I think most people know the version by Birdy, who had a great hit with her version of this song. So sometimes you make a small change that has a major effect, and sometimes you just need to start over again and reuse only a small part.

Keep in mind that not everything is crappy, it just needs love and attention. But beware of the time you spend loving and caring, you still need to deliver a great product. Keep learning, asking, coding, testing, and above all have fun in creating the best product.

Resources

- [1] http://en.wikipedia.org/wiki/History_of_programming_languages
- [2] the music in this article is available on: testowanie.nl/remix

> about the author

Daniël Wiersma



Daniël Wiersma is an Agile test consultant at codecentric, where he shares his passion for testing and Agile through his work and training on a variety of subjects. With 7 years of experience in IT and software testing, Daniel is experienced in many different testing roles. As a padawan, Daniel is learning about Rapid Software Testing and exploratory testing. Daniel maintains a blog at wiersma.net and testowanie.nl.

Twitter: [@dwiersma](https://twitter.com/dwiersma)

How Agile Methods Help Supermassive Games Deal with the Rapid Pace of Game Development

by Jonathan Amor

The world of the computer games industry is one of high pressure and fast pace. Game development studios are typically working to tight deadlines in what is an extremely competitive and ever-evolving marketplace. Modern games involve vast amounts of artwork, audio, and animation in addition to source code, so a wide variety of contributors are involved. Given aggressive release schedules, it is no surprise that the Agile methodology is a good fit for the games industry and, indeed, is becoming increasingly prevalent among game development teams.

One such example is Supermassive Games, based in Guildford in the UK. Since the company launched in 2008, it has established a strong track record, mainly developing games for Sony. It is currently working on two major PlayStation projects: “Until Dawn” and “Wonderbook: Walking with Dinosaurs”, which are both high-quality, character-based games with rich audio and animation.

Jonathan Amor is Supermassive’s Director of Technology and has been with the company since its early days, having joined from world-renowned games firm Electronic Arts. Jonathan describes how Agile has become an integral part of Supermassive’s business.

We have stuck to the core values of Agile beneath everything, because we have found that it provides us with a strong framework for helping projects keep on track in a fast-moving, creative and highly technical environment. Here are some examples of how we apply the four different pillars of Agile:

We value individuals and interactions over processes and tools

Face-to-face really is the best way to communicate, especially when trying to describe the idea for a particular game or feature, so we encourage people to talk, not just send each other emails.

A lot of what we do in the early days of a project is to iterate and discover the elusive “fun factor” that is so important to the success of any game. That is not a process you could build a tool or process to achieve, it can only be accomplished with iteration and collaboration. Something may seem that it would work in theory, but often it is not until you play the game that you can really tell.

We use Scrum to one degree or another across the studio and generally stick to its core tenets, with regular meetings to review progress, blockers, and plans. Every sub-team generally has its own task board, which is useful for making daily progress

visible. That said, how much we use Scrum does vary according to each project and the stage at which it is at: we find it works particularly well in the middle stages of a project when it is easier to define the goals clearly.

We avoid having overly prescriptive design or technical documents, because things change so quickly here. However, good tools and processes – while not as important as individuals and interactions – are still very important and support our more collaborative, iterative approach, as long as they are chosen well. For instance, using Perforce software version management gives us the freedom to focus on creating code and assets and exploring new ideas, without the concern of losing the original version: we simply roll back if needed.

A feature we find very useful within the version management system is Protection Groups which, when coupled with well-managed Client Specs, gives people a solid framework for what they should be focusing on and prevents them from making accidental changes outside of their remit. It also means that they only need a subset of the whole project source, rather than synchronizing everything. For example, we might put all the audio team members on a project into one group and then limit the folders and files they can use to just the audio and other relevant project data.

We value working software over comprehensive documentation

We are constantly reviewing the game – towards the end of the production process this can be every day, or even more frequently – so having working software is vital. Again, tools have a major support role here: we have built a dedicated build server that is always integrating the latest data through Perforce, so we can create a build of the game in just a few hours at most, often as little as half an hour. To give that some context, ten years ago it often took all night to create a build, despite having far less data.

Another part of helping us ensure we have working software is the use of branching within version management. For example, if we want to create a demo version of a game for a show, we can branch that out and work on it in isolation, so any changes will not affect the working mainline, preventing the propagation of bugs.

We value customer collaboration over contract negotiation

Our customers are primarily our publishers, but can also be colleagues within the company. We have found that taking a brief, working with a rigid set of requirements, going away and then coming back several months later just does not work well in practice. Requirements change, and what was needed may not have been clearly understood by either the developer or the customer in the first place. We believe it is far better to sit down with the “customer” (for instance, the tools programmer who creates gameplay content tools sitting with a designer) for about half a day and really understand the context of what is needed. We find that this usually leads to a much more effective solution in the end.

We work closely with our publishers throughout the process of developing a game, particularly their producers and QA teams. While a lot is planned and agreed in advance, a regular face-to-face relationship is invaluable for effective collaboration.

Customer collaboration also has to happen at a technical level. For instance, one of our systems supports localization of the text strings and voice-audio for a game. For an interactive drama game like “Until Dawn”, this could be as many as 25,000 lines of dialogue which is all stored in an online database. From our side, we can then pull down new versions of the localized data into Perforce or push any script changes back up; from the client side, their translators can take a drop of anything that has changed and put their translations back in the database. It may not sound major, but it would otherwise have been a laborious, time-consuming, and error-prone process of making manual notes on a spreadsheet.

We value responding to change over following a plan

We place a lot of importance on solid planning (and we now even have dedicated production managers who are focused on that). But within that framework, it is essential to be flexible and able to respond to change, which is very much part of our company’s ethos. The challenge is that embracing change is not natural for everyone, so it makes it a lot easier for people to deal with if the right supporting mechanisms are in place. Again, version management is a good example, because people can play around with ideas and make changes, safe in the knowledge that they can revert back to the original whenever they want.

Conclusion

The games industry is a fascinating business; the studios that will survive are those that can innovate, and are able to react to consumer behavior and market trends. As we continue to grow, we believe that Agile will help Supermassive to maintain that flexibility and responsiveness. ■

> about the author

Jonathan Amor



Jonathan Amor is Director of Technology at Supermassive Games, a UK game development studio. Since starting his career in the games industry 19 years ago as a programmer on the racing game “Formula 1” developed for the Sony PlayStation console by Bizarre Creations, Jonathan has worked as a programmer and technical director on 15 published titles. He has been interested in the use of Agile methodologies, especially in their pragmatic application to game development, since working for Electronic Arts where he was introduced to Scrum.. Jonathan’s current role involves managing a team of over 20 programmers, and helping to support the teams and technology used for the studio’s two current projects: “Wonderbook: Walking With Dinosaurs” and “Until Dawn”.

Stumbling Blocks

by Gurpreet Singh

Ours was a typical Waterfall team that believed to the core in the SDLC (systems development life cycle). Our requesters were always adamant about sending countless changes to us at all stages of the project flow. This led to a lot of rework, a decrease in customer satisfaction, an increase in the number of bugs, and other problems.

Hence our leadership thought about bringing in a new methodology, which had proved fruitful when it had been applied a few years earlier in another location: Agile. As the name suggests, the whole framework is flexible. Our group started off as the pilot project, and we finally implemented Agile across the entire service line.

However, our team faced numerous stumbling blocks while we were attempting to adopt Agile. I will focus below on the most prominent ones:

1. Inertia

We are performing so well, we thought. Why would we change? Why are we giving liberty to the client to change the scope at any time? How will the projects be completed?

To be honest, it is very difficult to tame the inertia of a team. We ran several rounds of meetings to educate the team about the Agile (specifically, the Scrum) framework.

2. Lack of discipline

Discipline is the core of Scrum. We all needed to be on time for all meetings, and we soon found that this was a problem. So we set up a “Softy meter”, which meant that whoever was late for a meeting (without advance notice) would have to bring Softy ice creams for the entire team. This was advantageous: we enjoyed a lot of Softies in the initial phase, and gradually the Softies were abolished as there were no more latecomers.

We also faced a problem in not time-boxing the meetings. Our stand-ups lasted for as long as 30 minutes, as parallel discussions erupted and diverted the agenda. Eventually the Scrum Master learned to intervene so that the team followed Scrum stand-up practices.

3. Learning responsibilities

An Agile team is a self-organizing team, motivated enough to work towards a common goal for the company. However, during the initial adoption period, we realized people were not aware enough about their roles. A product owner should focus on achieving the client’s business piece of the project. A Scrum Master should act as a facilitator to help remove the impediments.

The team should be self-organizing and have the liberty to size the stories, pull them individually, and assign them in the product backlog of the sprint.

4. Learning the prerequisites

The product owner and the Scrum Master should not be the same person, nor should either of them be the direct reporting manager of teammates. A product owner needs to be a single person, not a committee. Both the PO and the Scrum Master should be dedicated 100 percent to the project. If someone has to work on multiple projects, he or she needs to draw the line to protect each individual project and team so that they are always taken care of.

5. Expectation setting

The expectations of the client need to be set wisely. That is, if the client is finicky about the budget, then we should communicate clearly to him or her that we will give a daily, approximate cost estimate. However, if there is a non-negotiable cost, we will clarify that we will forward the product in its current state to the client once that cost has been reached.

6. Learning empiricism

Scrum is empirical in nature. Never make it too calculated or mathematical, or you will destroy its core purpose. For example, always draw a rough burn-down chart to keep it simple and meaningful. If you make it too “accurate”, using rules and scales, you end up wasting a lot of time and missing the real work.

Sizing the story is another example. Never size a story for the estimated time involved; sizing is a measure of the complexity of the task. The size should not depend on whether a senior developer takes it or a junior developer takes it. Sizing should also be relative, meaning that stories should be sized relative to each other.

Finally, the “Definition of Done” needs to be set. Acceptance criteria need to be outlined so we can judge whether a particular story is done or not.

7. Kill the Manager

We need to “kill” the manager as there is no manager role in a Scrum Team. The entire basis of Scrum lies in the functioning of self-empowered and cross-functional teams. If the Scrum Master or Product Owner tries to act as a manager and pushes the work to the team beyond the commitment, or change the scope/UAC of the stories within the sprints, then Scrum fails.

8. Destroy the Resource

Team members are “live” human beings. These are not “dead resources” like office furniture, archived mail, or a desktop, etc. We need to place huge emphasis on people and their communication, and this is clearly shown in the Agile Manifesto as well.

9. Possible need for customization

Agile hates people working on multiple projects simultaneously. If a PO does not have the needed bandwidth, then a proxy PO must be set up to fill that vacancy.

A few companies do not have Scrum Masters, and so the PO handles the dual roles of being the “client’s person” and the “team’s person” (Scrum Master). This leads to internal conflicts, as those roles need distinct people.

Sometimes a person acting as PO for one project is Scrum Master for another project, which can work well as long as schedules are respected. Alternatively, one of the team members doubles as a Scrum Master for some period (along with his core work) and then the baton is passed to another team member like in a relay race.

Our team today

Now we are a purely Agile team. Truthfully, the transformation from Waterfall to Agile practices was extremely difficult. It

took us several pilots; extra hours; weekend training courses; regular and ongoing coaching; learning to deal with ego clashes and inertia conflicts; and more. But today we are a happy Agile team. Scrum, sprints, retros, review meetings, discipline – they are all now part of our DNA. ■

> about the author

Gurpreet Singh



Gurpreet Singh is a Certified ScrumMaster and Agile enthusiast. He has over 8 years of experience in the software development industry. His passion is bringing transformation of companies to Agile. He has worked in Scrum, Kanban, and XP practices.

He has worked as a ScrumMaster, Product Owner, and Agile coach with geographically diversified teams spanning different industries like US healthcare, mobile applications, content management systems, etc.

His hobby is writing articles for blogs and newspapers (on general topics). He has exposure in other fields like Six Sigma, PMP, risk mitigation, SLAs and scoping.

Accredited ISTQB® and IREB® training material through licensing!



Díaz Hilterscheid

The Díaz & Hilterscheid GmbH has created ISTQB® and IREB® training material incorporating best practices and ample training experience. It provides an aspiring trainer with the necessary resources to successfully offer a comprehensive training program.

Save time and money by licensing the needed training material from Díaz & Hilterscheid GmbH. Course material is available to prepare participants for the following certifications:

- ISTQB® Certified Tester – Foundation Level
- ISTQB® Certified Tester – Advanced Level (Test Manager, Test Analyst, Technical Test Analyst)
- IREB® Certified Professional for Requirements Engineering – Foundation Level

- Course material includes presentation slides and exercises.
- Receive automatic updates and reviews for the licensed material.
- Available in up to three languages: English, German, and Spanish.



**For pricing conditions and other related matters,
please contact us by e-mail or phone.**



Díaz & Hilterscheid
Unternehmensberatung GmbH
Kurfürstendamm 179
10707 Berlin
Germany

Phone: +49 (0)30 74 76 28-0
Fax: +49 (0)30 74 76 28-99
E-Mail: training@diazhilterscheid.com
Website: training.diazhilterscheid.com

Risk Management in an Agile Way

by Edwin van Loon

Within risk-based test approaches, product risks are defined once beforehand and mitigated within a predefined test approach. The consequence of this approach is that changes within scope (backlogs), new insides or calculated uncertainties are not or only limited taken into account. One of the principles of Agile is that requirements changes have to be welcomed. This means that Agile and risk-based testing not go hand in hand.

Since risks and requirements are related to each other, the risk management process should be set up dynamically. This article describes a good practice, in which Agile and risk management are combined.

Risk

The Oxford Dictionary defines risk as a situation involving exposure to danger. It also states that all outdoor activities carry an element of risk, in particular the possibility that something unpleasant or unwelcome will happen. I have come up with the following ICT definition when translating those descriptions: a possible THREAT related to one or more REQUIREMENTS (user stories) causing DAMAGE to an organization.

Risks and requirements go hand in hand. A requirement without any risk is a non-needed requirement (a gadget) and a risk without a requirement is a missing requirement (an incomplete backlog). Combining the risk management and requirement management processes will increase the efficiency and effectiveness of the project and will also increase the detection of issues at an early stage.

Combining User Stories and Risks

User stories are normally described using the syntax:

“As a <person>, I would like <a need>, so that <the added value of this need to the person>.”

An example is the following need for a department that is responsible for managing financial master data.

“As an employee of the managed services organization, I would like to be able to centrally manage the financial master data so that changes in master data only need to be submitted once.”

The primary business risk related to this user story is implementing this user story correctly.

This can be described using the following syntax:

“As a <person>, I fear that <a failure occurs>, due to <an occurrence>.”

The following risk can be matched the user story example: “As an employee of the managed services organization, I fear that inconsistencies will occur in the master data, due to the fact that the master data is not distributed correctly to the local administrators.”

The introduction of this user story can also introduce some additional secondary risks, such as: “As an administrator, I fear that I am not able to submit the financial transactions, due to the fact that master data is not available or outdated.” This risk should be the trigger for introducing another user story.

This example shows the added value of combining risk management with requirement management.

Risk Management Process

Let's zoom in to the risk management process. Within the regular risk-based test methodologies there are four stages. Within the first stage the risks are identified using techniques like interviews, workshops, and brainstorming. This stage will result in a list of risks, which is the basis for the whole project ... but how can we be sure that this is the complete list and what about dealing with changes? Nassim Nicholas Taleb has written a book called “The Black Swan”. The Black Swan theory is a metaphor for events that come as a surprise and have a huge impact. Our thinking is usually limited in scope and we make assumptions based on what we see, know, and assume. Reality, however, is much more complicated and unpredictable than we think. So, how can we be sure that this list of risks is complete? We, as testers, are by nature the best doom thinkers, because we have developed the ability to “think negatively” during the hunt for defects. That will help us to be as complete as possible, but we still need to be able to reconsider this list at different stages within the project.

The next stage of the risk management process is the assessment of the risks. A risk consists of two parameters, namely business impact (the impact of the occurrence of that risk)

and risk likelihood. The business impact is predictable. We can know what financial or other damage will occur when a risk with regular proportions happens, such as a department that is not able to work for one regular hour. We can, of course, not predict the impact of a “black swan” like the attack on 11 September or the bankruptcy of Lehman Brothers, but that is not required. In the event a black swan happens, the project should at least be reconsidered.

Predicting the likelihood is something else. How can we predict the likelihood of the occurrence of a risk using factors like complexity of technology, conflicts within a team, and legacy versus new approaches? And how can we be that sure? If we are that good in predicting the “future”, we should also be able to predict what soccer team is most likely to win the FIFA world cup in Brazil in 2014. I think, if we had the capability, we would already be rich through winning many lotteries. This stage results in prioritized/classified risks. Most of the time we use four classes from “high business impact and high likelihood” to “low business impact and low likelihood”. My practice is to only use a maximum of two classes: high business impact and low business impact. The classes are only used for differentiating the test approach and, so, the assured quality. In many organizations there is no need to differentiate and so it is sufficient to merely identify the risks (and not prioritize them). In regulated environments, for example, there is the need to differentiate between requirements that are critical and non-critical in terms of regulation.

Last year I attended a presentation by Randall Rice on Defect Sampling. He compared testing with gold digging. Gold diggers begin with dirt sampling before they start digging. In order to be as efficient as possible in testing, we should also learn from the defects we find (or other new insides) and revise our test approach as required. In order to be able to work in such a way, there is a need for flexibility within the testing approach/plan. This flexibility is created by removing the “likelihood factor” and by defining a test approach with variable test coverage per risk class.

Normally the test specification techniques, including the coverage per defined risk class, are defined in the test plan, which does not allow a tester to revise the approach based on detected defects. Therefore the minimum and average test coverage have to be defined instead of the real test coverage. The minimum test coverage is the coverage that needs to be achieved within the first test execution sub-cycle. Within this sub-cycle, the system under test will be “sampled” by executing pre-defined test cases or by exploring. The minimum coverage (for example requirement or risk coverage) will be achieved within this cycle and the defects are logged, including the system area in which the defect is detected. Within sub-cycle 2, the real “digging” is executed. The number of defects per area detected during sampling will determine the test approach per area. So the likelihood factor is calculated dynamically during the test execution instead of in the risk assessment stage. This factor might also need to be re-calculated after every test cycle/iteration. The average test coverage is needed to

estimate the required effort beforehand within the test plan (and allocate the budget).

The last stage within the risk management process is called risk management. Within this stage, the achievement of the mitigating measures is managed. Regularly it is a rather “static” process in which the test results including defects are collected and reported, including any deviations from the original plan.

The defined risks are not static and need to be reconsidered on a regular basis, as already mentioned in the previously described stages. The dynamic way is to iterate the risk management stages in combination with the test cycles or iterations. So, after every test run, you need to reconsider the completeness of the list of risks and the “digging approach”.

This dynamic approach to risk management can be called “Agile-compliant”, because it can deal with changes and be fitted in well in all Agile methodologies. ■

> about the author

Edwin van Loon



Edwin van Loon is an ISEB practitioner and Lean six sigma certified test consultant who has gained a lot of ICT experience in different jobs and different sectors. He started his career in 1994 and specialized in quality and testing in 1998. Because of his wide experience and ability to deal with different kinds of situations, he uses a pragmatic approach to arrive at tailor-made custom solutions. Edwin has fulfilled several different testing and QA roles within his career. During the most recent years of his testing career he held test management and test consulting roles and has seen organizations and test projects increasingly adopting Agile principles. Valid is Edwin's current employer. At Valid, Edwin is responsible for the development of the quality management competence. This competence consists of the sub-competences Requirements Management, and Process Improvement and Testing. Edwin is also an experienced speaker, having presented three times at Eurostar and several times at other conferences like TestNet and the Test Automation Day. In October 2013, he spoke at the Agile Testing Days conference in Potsdam. This article is based on that presentation.

LinkedIn: [edwinvanloon](#)

Twitter: [@Edloon](#)

E-mail: Edwin.van.loon@valid.nl

Agile Architecture Engineering: Dynamic Incremental Design Selection and Validation

by Tom Gilb & Kai Gilb

Agile project management offers us a whole new method for approaching architecture and design engineering, of both IT systems and software.

Agile is *iterative* (cyclical, repetitive), and *incremental* (cumulating stakeholder value delivery), and *evolutionary* (learning from experience, and changing plans). This means we have very useful opportunities to manage systems and software architecture, and design, better.

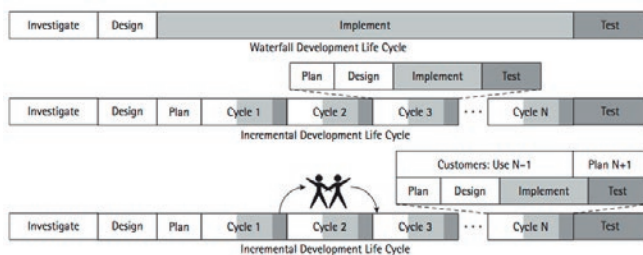


Figure 1

Architecture and design is complex, and we really know very little about the impact on values and costs of most of our initial design suggestions. They need to be considered as mere hypotheses – to be proven or disproven. The outcome is only roughly understood in advance, and our scope for estimation error is intolerably wide (at least in terms of order of magnitude) [8].

Agile offers a useful practical solution. But most Agile cultures, as taught and practiced today (Agile Manifesto, Scrum, XP for example) do not take a position on the measurement and management of architecture and design. But some earlier Agile methods, such as Evo (1970–2013 [3, 6, 10]) and Cleanroom (1970–1980s [1, 2, 6]), have long since exploited the architecture management opportunity inherent in cyclical incremental evolutionary system delivery to successfully manage the architecture and design itself.

Harlan Mills, IBM Federal Systems Division, comments on the ability of the Evolutionary “Cleanroom” method to control projects perfectly: “LAMPS software was a four-year project of over 200 person-years of effort ... in 45 incremental deliveries. There were few late or overrun deliveries in that decade, and none at all in the past four years” – Harlan Mills, in 1980 [1].

His colleague in the “Cleanroom” method, one of the first “Agile” methods, Robert Quinnan, comments on the “dynamic design-to-cost” aspects: “The method consists of developing a design, estimating its cost, and ensuring that the design is cost-effective.” (p. 473, [2])

He goes on to describe a design iteration process that tries to meet cost targets either through redesign or by sacrificing “planned capability”. When a satisfactory design at cost target is achieved for a single increment, the “development of each increment can proceed concurrently with the program design of the others”.

“Design is an iterative process in which each design level is a refinement of the previous level.”

But they iterate through a series of increments, thus reducing the complexity of the task and increasing the probability of learning from experience, won as each increment develops and as the true cost of the increment becomes a fact.

“When the development and test of an increment are complete, an estimate to complete the remaining increments is computed” (Quinnan, [2]).

The “developers” in the current Agile culture are not going to do anything about this. The just want to “code”. The responsible architects, such as IT Architects, are going to have to figure out how to exploit Agile for *their* purposes.

The first stage of this is to recognize that *architecture* (the overall discipline of managing a system development through design) and *design* (which includes specialist disciplines such as Human Interfaces Design, Security Engineering, Performance Engineering, and other such disciplines supervised by the overall architect) need to be conducted as an *engineering* discipline. Not as art or poetry.

“*Engineering*” means managing a numeric set of objectives and constraints, which are our architecture requirements. *Engineering* then implies managing the corresponding numeric attributes of all design and *architecture* (defined as the things we do to achieve our performance and quality requirements, within our resource constraints).

One interesting side-effect of managing architecture as an *engineering* discipline, is that we not only get control over performance and quality aspects of the system, but we simultaneously get far better control over our budget and deadline [1,2, 6, 8], as Cleanroom experience proved long ago.

Technical Prerequisites

In order to do this, (and several groups have done it for decades, so this is not idle speculation, but observation of known methods!) we need to learn and practice the following architecture engineering disciplines:

1. Quantification of all critical *quality* aspects (security, maintainability, usability etc.) [9].
2. Design of suitably cheap processes for measuring at least leading indicators, then better final indicators, of the incremental delivery of technical qualities (like degrees of security or usability) and then of their intended knock-on effects to a higher level of stakeholder interest (e. g. stakeholder perceptions such as “saving time”, “feeling confident”).
3. Ability to decompose [10, 11] our larger high-level architecture ideas into smaller implementable components (so we can get earlier delivery of their value).
4. Ability to test design-component hypotheses in a safe, but realistic, way before confidently scaling up, once they are working as required.
5. Contracts for outsourcing that envision, allow for, and assist our ability to do all the above engineering and exploration; with the ability to be agile and exchange what does *not* work for that which *does*! The *real* heart of agility [7].

The Agile Architecture Engineering Process [10, 13]

The Architecture Engineering process using “Planguage” as a planning language, or any other way to express qualities quantitatively, goes like this. I use a week to get through these initial plans, before diving into cycles of delivering real value from the implementation of architecture components [5].

1. Quantify the top few critical performance and quality objectives for the system. This means a defined scale of measure and at least one level of performance expected in the future. The agenda is that the project will be done, and successful, when these levels of requirements have been reached. Day 1.
2. List and define in some detail (maybe a page each of ten major architecture ideas) [3, the CE book, Design Chapter template for detail] the major architecture components. These should be the set of ideas you believe will enable you to reach the critical requirement levels in the first step above. Day 2.
3. Rate the expected effectiveness of each architecture component on all critical objectives, as well as on critical resources such as money and time. Use an Impact

Estimation Table [14]. One rating is to estimate the % effectiveness expected by the deadline. (100 % means we reach numeric goals on time). Day 3.

4. Using the information in the Impact Estimation Table, find subsets of the architecture that are estimated to give very high value (performance and quality requirements level) in relation to resources used [15]. The most efficient designs. Schedule these for early value-delivery cycles [10]. Day 4. Day 5 is presentation to management.
5. Evaluate results (feedback from delivery cycles, on measures of value and costs). Decide what you need in order to improve or learn. Plan the next steps, with a view to maximizing fast progress towards your requirements levels.

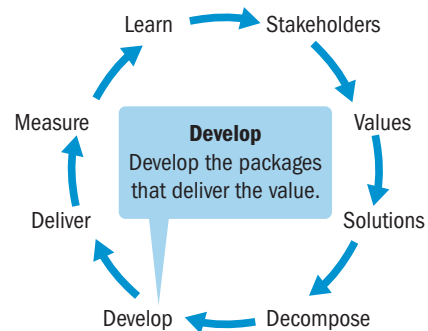


Figure 2. The Evo Cycle [16] – an extension of the Shewhart/Deming “Plan Do Study Act” cycle of SPC methods. Developed by Kai Gilb.

In Summary: we can *engineer* the architecture in *incrementally*. We get earlier results and better results, as a result [4].

References

- [1] Mills, H. 1980. “The management of software engineering: part 1: principles of software engineering”, *IBM Systems Journal* 19, issue 4 (Dec.): 414–420. Reprinted 1999 in *IBM Systems Journal*, Volume 38, Numbers 2 and 3.
A nice sample, in slides, of how Cleanroom reaches the highest military and space performance and quality levels “always on time and under budget”:
<http://www.gilb.com/dl602>, “Architecture Prioritization”, Oct 2013. See ref. [6] for source of Mills and Quinnan.
- [2] Robert E. Quinnan, “Software Engineering Management Practices”, *IBM Systems Journal*, Vol. 19, No. 4, 1980, pp. 466–77.
A nice example in slides for Oct 9 2013 1.5 hour talk at London “Software Architect” conference.:
<http://www.gilb.com/dl602> “Architecture Prioritization” Oct 2013. Quinnan goes into detail on his dynamic design to cost practice within Cleanroom. See ref. [6] for source of Mills and Quinnan.
- [3] Gilb, T. 2005. *Competitive engineering: A handbook for systems engineering, requirements engineering, and software engineering, using planguage*. Oxford: Elsevier Butterworth-Heinemann. **Free digital copy for first 50 Agile Record readers who email me with a request**

within one week after publication of this paper. After that, see 9 and 10 below.

- [4] Gilb: "What's Wrong with Software Architecture". Keynote Software Architect Conference, London 10 Oct 2013. <http://www.gilb.com/dl603>
Slides in PDF: <http://vimeo.com/28763240>
- [5] "Confirmit" Company Cases (use of Evo). Johansen, T., and Gilb, T. 2005. *From waterfall to evolutionary development (Evo): How we rapidly created faster, more user-friendly, and more productive software products for a competitive multi-national market*. Available at: http://www.gilb.com/tiki-download_file.php?fileId=32.

And: *The Green Week: engineering the technical debt reduction in the Evo Agile Environment* by Confirmit. <http://www.gilb.com/dl575>
A Gilb's Mythodology column published in May 2013 in Agile Record No. 14, www.agilerecord.com. This paper highlights a case of reengineering a legacy system to give reduced technical debt, using Evo, in a small Norwegian international market software package house.
- [6] Gilb, T. 1988. *Principles of software engineering management*. Boston: Addison-Wesley. See http://www.gilb.com/tiki-list_file_gallery.php?galleryId=15. "Some deeper and broader perspectives on evolutionary delivery and related technology", chapter 15 of the book.
- [7] *Agile Contracting for Results The Next Level of Agile Project Management*: Gilb's Mythodology Column Agile-record No. 15, August 2013, www.agilerecord.com. <http://www.gilb.com/dl581>
- [8] Estimation Paper: *Estimation: A Paradigm Shift Toward Dynamic Design-to-Cost and Radical Management*. SQP VOL. 13, NO. 2/© 2011, ASQ. <http://www.gilb.com/dl460>
- [9] CE book. Chapter 5: Scales of Measure: http://www.gilb.com/tiki-download_file.php?fileId=26
- [10] CE Book. Chapter 10: *Evolutionary Project Management: Evo Standard 2012 for DB, Non-Confidential* http://www.gilb.com/tiki-download_file.php?fileId=77
http://www.gilb.com/tiki-download_file.php?fileId=487
- [11] Gilb, T. 2010b. *The 111111 or Unity Method for Decomposition*, presented at the 2010 Smidig (Agile) Conference, Oslo. Available at: http://www.gilb.com/tiki-download_file.php?fileId=350
- [12] *Agile Contracting for Results The Next Level of Agile Project Management*: Gilb's Mythodology Column Agile-record August 2013 <http://www.gilb.com/dl581>
- [13] Evo standards Feasibility Study paper. "Project Startup" for agile architecture. <http://www.gilb.com/dl568>
Our column in Agile Record No. 13, www.agilerecord.com, as published 7 March 2013
- [14] *Impact Estimation Tables: Understanding Complex Technology Quantitatively*. <http://www.gilb.com/dl23>

- [15] On Decomposition. See ref. 11 above and: *Decomposition of Projects: How to Design Small Incremental Steps* <http://www.gilb.com/dl41>
- [16] See [gilb.com](http://www.gilb.com) for a dynamic version of the Evo cycle and for more explanation. <http://www.gilb.com/Site+Content+Overview>

> about the authors

Tom Gilb and Kai Gilb



Tom Gilb and Kai Gilb have, together with many professional friends and clients, personally developed the agile methods they teach. The methods have been developed over five decades of practice all over the world in both small companies and projects, as well as in the largest companies and projects. Their website www.gilb.com/downloads offers free papers, slides, and cases about agile and other subjects. There are many organisations, and individuals, who use some or all of their methods. IBM and HP were two early corporate-wide adopters (1980, 1988). Recently (2012) over 15,000 engineers at Intel have voluntarily adopted the Planguage requirements specification methods; in addition to practicing to a lesser extent Evo, Spec QC and other Gilb methods. Many other multinationals are in various phases of adopting and practicing the Gilb methods. Many smaller companies also use the methods.

Tom Gilb

Tom is the author of nine published books, and hundreds of papers on agile and related subjects. His latest book 'Competitive Engineering' (CE) is a detailed handbook on the standards for the 'Evo' (Evolutionary) Agile Method, and also for Agile Spec QC. The CE book also, uniquely in the agile community, defines an Agile Planning Language, called 'Planguage' for Quality Value Delivery Management. His 1988 book, *Principles of Software Engineering Management* (now in 20th Printing) is the publicly acknowledged source of inspiration from leaders in the agile community (Beck, Highsmith, and many more), regarding iterative and incremental development methods. Research (Larman, Southampton University) has determined that Tom was the earliest published source campaigning for agile methods (Evo) for IT and Software. His first 20-sprint agile (Evo) incremental value delivery project was done in 1960, in Oslo. Tom has guest lectured at universities all over UK, Europe, China, India, USA, Korea – and has been a keynote speaker at dozens of technical conferences internationally. Twitter: [@intomgilb](https://twitter.com/intomgilb)

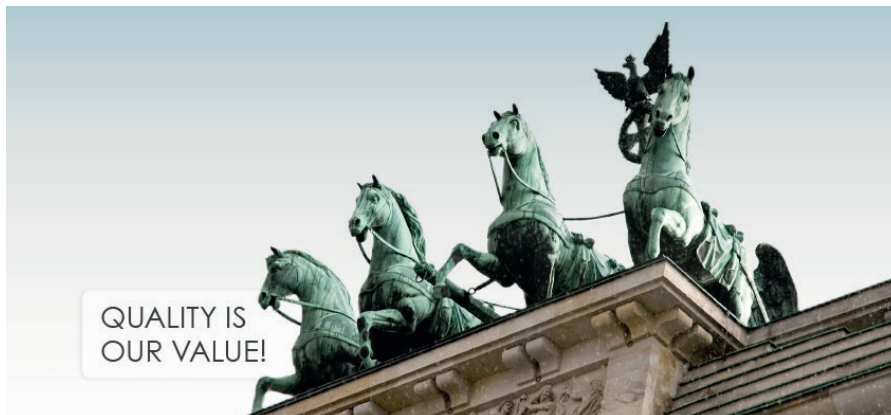
Kai Gilb

Kai Gilb has partnered with Tom in developing these ideas, holding courses and practicing them with clients since 1992. He coaches managers and product owners, writes papers, develops the courses, and is writing his own book, 'Evo – Evolutionary Project Management & Product Development.' Tom & Kai work well as a team; they approach the art of teaching their common methods somewhat differently. Consequently the students benefit from two different styles.



Book your training with Díaz & Hilterscheid!

Díaz Hilterscheid



CAT – Certified Agile Tester

CAT is no ordinary certification, but a professional journey into the world of Agile. As with any voyage you have to take the first step.



HP Quality Center



Incorporate the abundant tool knowledge of HP Quality Center in this workshop: From the module architecture to analyzing and customizing. You will be able to configure your test project, utilize requirement data, execute efficient test cases, and apply efficient solution to discovered issues.

HP QuickTest Professional

This workshop enables the quick and efficient Use of QTP. Discover the individual functions: documentation, reproduction, object repository, application of Syncpoints and Verifys, utilization of Active Screens, Features Data Tables, and the linkage to QC for automatized testing, producing and applying results.

Certified Agile Essentials

Certified Agile Essentials

This two-day course is aimed at anyone involved in software engineering who wants to become familiar with working in an Agile environment, giving candidates a solid understanding of roles, processes and techniques used in Agile projects. It provides an overview of the activities building on a basic understanding, reinforced through a heavy emphasis on discussion and practical application.

IREB® Certified Professional for Requirements Engineering – Foundation Level

This training course introduces you to the correct and complete specification, documentation, checking and management of requirements. You will learn techniques, procedures and tools, and you will be introduced to the fundamentals of communication theory. The training course is aligned to the syllabus of the independent International Requirements Engineering Board (IREB®).



ISTQB® Certified Tester

Foundation Level

In this training course you will learn about the most important test techniques and procedures that you can use to prepare and execute software tests efficiently and effectively, and which will help you to make a decisive contribution to your project's success.



Advanced Level – Test Manager

Building on the fundamental knowledge of the Foundation Level, this course teaches you all of the theoretical planning, steering and control tasks in test management and discusses your possibilities for their practical implementation using examples.

Advanced Level – Test Analyst

Based on the basic knowledge taught in the Foundation Level course, this training will teach you in-depth knowledge about technical testing tasks and review techniques. In this connection, your possible course of action in every-day work situations will be discussed with the help of numerous examples.

Advanced Level – Technical Test Analyst

Based on the basic knowledge taught in the Foundation Level course, this training will teach you in-depth knowledge for structure-based testing tasks and test automation matters. In this connection, your possible course of action in practice will be discussed with the help of numerous examples.

In-house Training: All of our training courses are available as private/in-house training. Please contact us for details.

For more training courses and current training dates, please visit our website or contact us:



Díaz & Hilterscheid Unternehmensberatung GmbH
Kurfürstendamm 179
10707 Berlin
Germany

Phone: +49 (0)30 74 76 28-0
Fax: +49 (0)30 74 76 28-99

E-mail: training@diazhilterscheid.com
Website: training.diazhilterscheid.com

Online Training

Save up to 60% with online training!
Visit our online shop to see what courses are available:
www.te-trainings-shop.com

-60%

Automated Blackbox Testing of New Age Websites

by Nilay Coşkun

Test automation is the process that executes generated test cases with automated testing tools and compares the predicted output with actual output. Although many defects can be found with manual testing, it is a time consuming (and, to be honest, too boring) process. Automated tests generate test cases automatically and execute all the test cases in the same way each time. This process reduces human error and costs, since automated tests can be run repeatedly and quickly.

There are various automated test tools used in the software test life cycle. In this paper I want to demonstrate an automated dynamic black box testing approach with Selenium IDE and Webdriver. Firstly, why Selenium? Selenium is a browser automation framework and Selenium IDE is the integrated development environment that allows recording, editing, and debugging tests. You do not need to learn a test scripting language to use Selenium. It is implemented as a Firefox extension and deploys on different Linux, Windows, and Macintosh platforms. Once you record a test case using Selenium IDE, the recorded test case can be exported in most programming languages such as Java, .net, Perl, Ruby, and HTML. You can record scripts automatically and edit manually. Editor provides you with autocompletion.

Selenium Webdriver is the automated testing tool for testing web applications. With Selenium 2, Selenium Remote Control has been deprecated in favor of Selenium Webdriver. It provides Java API for ease of use and understanding. Simply, it is the process of writing a JUnit or Test NG test case in a Java Project and executing the test case in a “main” method. Any condition in the test case can be controlled using if-else conditions. It enables the creation of dynamic test cases which means you can test dynamic web pages whose data can be different at any time, so the test case can fail. In Selenium 1, Selenium Remote Control was necessary to run the tests. With Selenium 2, Selenium Remote control has been officially deprecated in favor of Selenium Webdriver. Selenium Webdriver does not need a server to run the tests. It directly starts a browser and executes tests.

Selenese is the name of Selenium commands. You can develop tests and use Selenium IDE running on Firefox. So you can execute your tests against other browsers instead of only executing on Firefox. A test script is formed as a sequence of selenese. Selenium provides very rich command sets so you can completely test your web application. In selenese, you can test your UI elements based on HTML tags, list options, form submissions, or table data. Selenium provides testing windows size, alerts, and dynamic contents on your webpage,

mouse position, alerts, pop-ups, and so on. Selenium has three different command types, which are Actions, Accessors and Assertions. Actions indicate the “select this” or “click on this”, etc. Accessors are the commands that examine the state of the application and store the result. Assertions verify the state of the application and make comparisons with the expected result. You can control that the application is on the correct page using Assertions.

| Matrix Tree Sample Test Set | | |
|-----------------------------|---|------|
| include | testSetup.html | |
| open | \${GIPATH}/shell.html?jsxapppath=\${SAMPLEPATH}/32test-matrix | |
| waitForElementPresent | JsxToolBarButtonName=tbbTree | 5000 |
| assertElementPresent | JsxToolBarButtonText = Tree | |
| click | JsxToolBarButtonText = Tree | |
| verifyElementNotPresent | JsxMatrixTreeItemId=mtxTree,4 | |
| verifyElementNotPresent | JsxMatrixTreeItemId=mtxTree,5 | |
| clickJsxMatrixToggleTree | JsxMatrixTreeItemId=mtxTree,3 | |
| pause | 1000 | |
| verifyVisible | JsxMatrixTreeItemId=mtxTree,4 | |
| clickJsxMatrixToggleTree | JsxMatrixTreeItemId=mtxTree,4 | |
| pause | 500 | |
| verifyVisible | JsxMatrixTreeItemId=mtxTree,5 | |

Table 1. Selenese

Blackbox testing is the process of examining the functionality of an application without taking into consideration the internal progress. If we think of the application as a black box, the black box testing approach is about whether the black box gives the same output with the same input. It does not consider what happened inside the black box. You can use Selenium Webdriver for automated black box testing of your webpages.

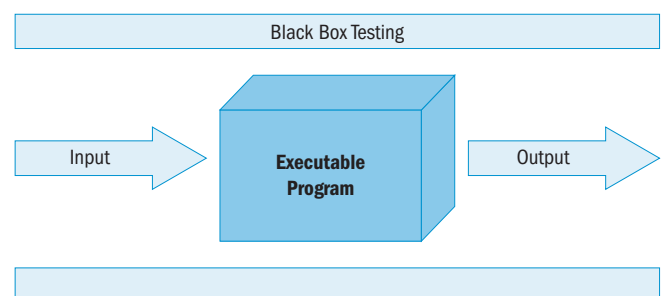


Figure 1. Blackbox Testing

With Selenium Webdriver you can check if you are on the right web page with Assertions. Selenese allows that to check the page elements. You can verify whether an element is on the page, if a text is on the page, or if a text is at the special location of the page. You have to give a target to the Selenium commands and this can be XPath. Xpath shows the location of nodes in an XML document. So you can use Xpath to refer to an element in the HTML. You can use a name or attribute for locating, but using XPath makes your tests more robust.

When developing a test, Selenium allows debugging and the use of breakpoints and starting points.

Let's assume that you are going to generate an automated test case for booking a flight from New York to San Francisco for today. When recording a test case with Selenium IDE, the steps are simply:

- Open the web page with a given url.
- Verify the page
- Select From
- Verify the From node *Optional
- Select To
- Verify the To node *Optional
- Select today's date
- Verify the date node *Optional
- Verify the Submit button *Optional
- Submit the form
- Check the result

The expected result would be the list of flights. The next step is selecting a flight and making a reservation. With Selenium IDE, we can select a flight and submit the form. But if the selected flight is not available or there is no available flight at all for this date, what will happen? We should control such a case dynamically so the test case executes successfully at any time. In this case, if the flight result list is empty our test case will fail. Webdriver allows controls to be made in the test case for any situation.

The reason why we use Selenium IDE and Webdriver together is because we record a test case using Selenium IDE and execute the test case using Webdriver API. It enables the exported test case to be edited instead of having to write the entire case right from the beginning. There are various cases in which exported Java code cannot be executed by Webdriver, since some codes are not compatible with Java. But this approach enables effective automated test cases to be generated for most of the cases. ■

> about the author

Nilay Coşkun



Nilay is from Turkey and has been living in Istanbul for 3 years. She received her Bachelor of Science in Computer Engineering at University of Kocaeli. She still continue Masters at Istanbul Technical University, Computer Engineering Department. She has been working for Eteration for 3 years. She had different roles on different projects for telecommunication companies such as Business Analyst, Quality and Test Manager and Project Manager.

Twitter: [@nlycskn](#)

Refactoring to Combinators

by Carlos Blé

The famous Refactoring book by Martin Fowler (martinfowler.com/books/refactoring.html) is focused on improving the design of object oriented code. Now, functional programming is becoming more and more popular for several reasons and modern programming languages are becoming multi-paradigm, supporting functional programming (FP) in addition to object-orientation (OO). In this article I present ideas on applying refactoring through functions with JavaScript, not necessarily using FP.

One of the popular multi-paradigm languages is JavaScript whose functional aspect was inspired by Scheme, a Lisp dialect. Like many other people, I started using JavaScript as an object-oriented language, trying to port my style from C# or Java, or even Python. And you can do it, but it is not natural, as you are not taking advantage of the powerful features of the language. I like the idea of OOP for high-level design and FP for lower level operations. They can be mixed up in any way, but I prefer OOP for the high level because I find it expresses better the metaphors from the business domain. It depends on the problem, though.

FP brings immutability which makes it suitable for concurrency and parallel algorithms, as the state of the variables does not change. This avoids race conditions and deadlocks, for example. The multi-core machines we have today are one reason for FP to become popular. Immutability saves us from defects related to the state of the objects, even if we do not have concurrency.

But the fact that we use functions in JavaScript does not mean we are using a functional programming approach. You can use functions and still change the state because it is a hybrid language. Thus, not only the multi-paradigm aspect is an advantage in JavaScript, but also the way it supports functions.

Functions in JavaScript are first-class citizens like any other type in the language, so they can be passed in as arguments or returned from other functions. Moreover, every function defines an environment that can have nested functions and *closures*. A closure is a function that has “free” variables, i.e. variables defined in an outer environment:

```
1 function() { // pure function
2   var x = 1; // local variable
3   function() { // closure
4     x = 2; // free variable
5     var y = 3; // local variable
6   }
7 }
```

The inner function in the example is a closure. It has access to the variable “x”. The outer function is a pure function because

it has no free variables. Variable “y” is only accessible in the closure, whereas “x” is visible in both functions.

Some time ago I used to think that, given an OOP design, I should not pass functions as arguments to other functions because that could break encapsulation. Well, JavaScript blurs the lines because functions are also a particular kind of object:

```
1 var f = function(a,b) { return a + b; };
2 f.length; // 2
3 f.someOtherDynamicProperty = 5; // created on the fly
```

It turned out that I was wrong. To decide whether you are breaking design principles like encapsulation or the Single Responsibility Principle, you have to look at each particular case.

And the same applies to other multi-paradigm languages like C# or recent versions of Java.

Douglas Crockford in his book “JavaScript: The Good Parts” shows an alternative way to design classes, called *functional inheritance*. This is the style I use now in my code:

```
1 function someClass() {
2   var self = {};
3   var privateMember;
4   function privateMethod() { /*...*/ }
5   self.publicMethod = function() { /*...*/ }
6   return self;
7 }
8 function someChild() {
9   var self = someClass();
10  self.publicMethod = function() {
11    /* overriding method... */;
12  }
13  return self;
14 }
```

It is clear and saves me from making typical mistakes with the “this” and “new” keywords. I would like to say “thank you” to my friend Guillermo Pascual for telling me about this and also about the JavaScript Allongé book (leanpub.com/javascript-allonge), which is the one that gave me the following ideas on refactoring.

Let’s see how combinators provide a very interesting way to refactor duplicated code.

As Reginald Braithwaite explains in the book, a basic definition of combinator is a function that takes only functions as arguments and returns a function. I cannot think of *combinators* straight away when I start off a new piece of code, my brain just does not work that way. The code is too smart for me to start with. In this case, smart does not mean hard to read. So, reading the book could be a bit frustrating until you see how powerful combinators can be in removing duplication. Say you have these two functions:

```

1  function doSomething(arg1) {
2      if (arg1 !== null && arg1 !== undefined)
3          doTheStuff(arg1);
4      /*... some code ...*/
5  }
6  function doOtherThing(arg1) {
7      if (arg1 !== null && arg1 !== undefined)
8          doTheOtherStuff(arg1);
9      /*... some code ...*/
10 }

```

As you can see, there is duplication. The first thing we can do is to extract a method with the condition we are checking. I always encapsulate complex conditions into methods, and I consider them complex when there is a logical operation or anything that makes me think in order to understand the condition. But, even if we create a method and the code reads better, we still have some kind of duplication. Let's get rid of it with a combinator:

```

1  function maybe(fn) { // our combinator
2      return function(arg1) {
3          if (isSomething(arg1)) // the extracted method
4              containing the former conditional
5              return fn(arg1); // invoking the target function
6          }
7      }
8      // The refactored code:
9      function isSomething(arg1)
10         return arg1 !== null && arg1 !== undefined;
11     function doSomethingWith(arg1) {
12         doTheStuff(arg1);
13         /*... some code ...*/
14     }
15     var doSomething = maybe(doSomethingWith); // new function
16     function doOtherThingWith(arg1) {
17         doTheOtherStuff(arg1);
18         /*... some code ...*/
19     }
20     var doSomeOtherThing = maybe(doOtherThingWith);
21     // new function

```

There is no duplication now! The behavior is exactly the same and we have not broken anything. Now, this “maybe” combinator borrowed from Haskell is not generic, it will not work for all the cases. But you might not need it to be generic at this point. Especially if you are test-driving the code, you know it will get more generic as the tests get more specific so you don't have to take too big steps. If all your usages of the “maybe” combinator are the ones above, you can leave it like that as long as the combinator is placed where other developers understand that it is not generic enough for other cases. The good part is that this code is simpler and easier to understand than the generic maybe combinator (www.leanpub.com/javascript-allonge/read#maybe) you can find in the book. It is up to you and your needs. I would keep it as a private function if it is not a general purpose combinator.

Another example is the “fluent” combinator for creating fluent APIs. Fluent interfaces (www.martinfowler.com/bliki/FluentInterface.html) are often used in the Test Data Builder (www.c2.com/cgi/wiki/TestDataBuilder) pattern as well as test doubles libraries and, of course, production code.

This is the code without the combinator. Can you find the duplicated behavior?

```

1  function someClass() {
2      var self = {};
3      var someProp, otherProp;
4      self.withProperty = function(val) {
5          someProp = val;
6          return self;
7      };
8      self.andProperty = function(val) {
9          otherProp = val;
10         return self;
11     };
12     return self;
13 }
14 var someInstance = someClass().withProperty(5).
    andProperty(7); // usage

```

As you can see, both setters return the object itself in order for the API to be fluent. We can consider this as a kind of duplication. Let's remove it with the “fluent” combinator:

```

1  function someClass() {
2      var self = {};
3      var someProp, otherProp;
4      function fluent(fn) { // our combinator
5          return function(arg1) {
6              fn(arg1);
7              return self;
8          }
9      }
10     self.withProperty = fluent(function(val) {
11         someProp = val;
12     });
13     self.andProperty = fluent(function(val) {
14         otherProp = val;
15     });
16     return self;
17 }
18 var someInstance = someClass().withProperty(5).
    andProperty(7);

```

This gives the same behavior with no duplication, and it also documents the code by making it explicitly part of a fluent API. As with the previous example, the combinator is not generic enough. A generalization could be this:

```

1  function fluent(fn) {
2      return function() {
3          fn.apply(self, arguments);
4          return self;
5      }
6  }

```

These two combinators can give you an idea of how powerful functions can be in removing duplication, which is one of the major aims of refactoring. Notice how the second code example is object-oriented and we still use functions to remove duplication without breaking encapsulation.

When working with collections, the benefit of the combinators is more obvious. Repeating the same kind of loop over and over may be a symptom of the fact that we can extract a combinator to solve the problem for us. The combinators I have seen so far that operate on collections have their basis in the “Array.prototype.map” function that comes with ECMAScript 5, now supported by all modern browsers (for older browsers there are powerful libraries like Underscore (underscorejs.org)).

Imagine I have a list of objects and I want to collect only a certain attribute:

```

1  var items = [
2    {propA: 1, propB: 2},
3    {propA: 7, propB: 8}
4  ];
5  function collectPropA(items) {
6    var result = [];
7    for (var i = 0; i < items.length; i++)
8      result.push(items[i].propA);
9  };
10 function collectPropB(items) {
11   var result = [];
12   for (var i = 0; i < items.length; i++)
13     result.push(items[i].propB);
14 };

```

You can remove duplication by passing the name of the property as a parameter to the function:

```

1  function collectProp(items, propName) {
2    var result = [];
3    for (var i = 0; i < items.length; i++)
4      result.push(items[i][propName]);
5  };

```

But you can also use the “map” built-in function:

```

1  function collectProp(items, propName) {
2    items.map(function(i) { return i[propName]});
3  }

```

Now imagine I need to implement another functionality in the array of objects:

```

1  function calculate(items) {
2    items.map(function(i) { return i.propA + i.propB});
3  }

```

We can extract that particular behavior using the “mapWith” combinator:

```

1  function mapWith(fn) {
2    return function(items) {
3      return items.map(fn);
4    };
5  };
6  var calculate = mapWith(function(i) {
7    return i.propA + i.propB; });
7  var collectPropA = mapWith(function(i) {
8    return i.propA; });

```

We are avoiding duplicating the loop, where I usually make mistakes because I tend to write “lenght” rather than “length” and it does not complain about this. Even if we use the map function for both implementations, that is duplication. If we need to replace the “map” implementation with Underscores, we would need to change several lines. On the other hand, if it is encapsulated in the “mapWith” combinator, we only have to change it in one place.

I am looking forward to reading the book “Functional JavaScript” by Michael Fogus.

I believe someone will write a book on “Refactoring for JavaScript” in the near future. In the meantime, enjoy the great books mentioned in this article, I totally recommend them. ■

> about the author

Carlos Blé



Carlos Blé started using computers at the age of six, when his father bought a PC with the Intel 8086 and some books on the Basic programming language. Since then, he has been learning how machines can be used to build a better world. He started earning money by writing software back in 2000. In 2008, Carlos started using TDD for pretty much every piece of software he needed to write. In 2010, he published the first book on TDD in the Spanish language. Over the last three years, he has been training developers in several countries and writing code with them. Apart from that, he has been investing his time and money in several web start-ups, always applying what he teaches to his own crafted software.

Twitter: [@carlosble](https://twitter.com/carlosble)

Website: www.carlosble.com

Build Your Immune System and Maintain a Healthy Codebase

by Augusto Evangelisti

I like metaphors, they help people see things from different perspectives and also stimulate thinking that normally wouldn't happen when discussing a subject in a traditional one-dimensional way.

My metaphor is between the human body and the code base of an application you are building and maintaining. Let's think about your application as a living organism, in particular I will compare it to the body of a teenager.



Artwork by Xavier Salvador

Teenagers grow quickly and so does our software when our business partners constantly seek the delivery of new requirements to satisfy our customers. Teenagers grow fast, their bones get longer, their hair grows in places that were bare before, and their muscles become bigger and stronger. Agile teams often have to cope with ever-changing requirements and a fast-growing code base. Now think of a user story like a meal or a drink that our teenager will have. Food and water (some alcohol, we're in Ireland) will be transformed in the teenager's body and will turn into larger bones, stronger muscles, facial hair, etc. All good until now, right?

Then what happens when the teenager eats food that is out of date, or maybe one night he has too much to drink? Well

the human body is equipped with an amazing self-defence tool called the immune system that will do its best either to eject the poisonous food and drink or to combat it and make a dangerous situation into a simple stomach ache or at worst a hangover.

Imagine what would happen if the human body did not have such a defence mechanism. Out of date food could kill the teenager, and an excess of alcohol could poison his blood and cause a lot of damage to our poor silly teenager who had one too many.

Our body is really good at dealing with these situations, but is our code base?

What happens when some really bad code gets into the system? Well the first to know will be our customers. They will not be able to get the value that they normally expect and this might mean that they go and get that value somewhere else. Following that sequence might mean that one day we end up with no customers and our software and company will die, just like a sick teenager with no antibodies.

What do I mean by really bad code? Really bad code is code that:

1. Doesn't have any tests.
2. Has bad naming conventions.
3. Contains duplications.
4. Is tightly coupled.
5. Is not maintainable.
6. When a developer looks at it for the first time he says "What the heck?" (thanks uncle Bob), holds his head in his hands, and asks himself: "Why didn't I go to business school?"

What can a developer do to stop the rot? What is the software's immune system? Fear no more, REFACTORING comes to the rescue. You might say: "Yeah, sure, easy said, but what if I break something else?" Well, look at the above list once again. What do you read in point 1? Tests, uhm ...

Tests are the first defence; they are your vaccine for the next time that you inject similar bad code. The vaccine will tell you straight away not to push that change into your body, how about that?

If you are dealing with a bad piece of code with no vaccine and you want to cure it, the first thing you need to do is to write a lot of tests. To guarantee that you will not break anything, you need to build a safety net around your poisonous code. If the code is so coupled that resembles an Italian dish also known as Bolognese, then you might not be able to write good unit

tests at first until you untangle it a bit. You could write some integration tests that cover a group of tangled classes; they will not be pretty but they will represent some kind of safety net when you start refactoring. The deeper you get into refactoring, the more you will discover that writing tests becomes easier, because clean code is also more testable code. See, we are getting there.

Step back for a minute now and you will be thinking: “OK this is painful and I don’t want to do it, so what can I do to avoid it all together?” Well, there is a solution. Start writing only clean code with loads of tests and refactor every time it is necessary; don’t think somebody else will do it for you. Writing a lot of unit and integration tests will give you the confidence of refactoring without having to worry about having unleashed a chain reaction that will ignite a nuclear war. Because your code is clean, simple, and has a lot of tests that will warn you very quickly when something is not quite right. Make sure that you keep that beautiful teenage body in perfect shape, so that if he eats a poisonous user story, you can be the immune system and refactor him into good health!

A few months ago we were releasing the MVP (Minimum Valuable Product) for a very important and risky project on the Sunday and at the stand-up on Friday one of our most senior developers said: “Lads, I found a small issue with component X and I think we need to change the way we operate there. I will need to do a refactor, I will be finished today anyway.” Nobody said anything and I thought I had reached “agile development nirvana” because the fact that nobody said anything and worried about anything demonstrated clearly that our code was in such good state and had such a great automatic regression suite that our friend’s refactor was not going to break anything and we would be releasing on schedule on Sunday with no problems.

Now ask yourself, how many times have you worked in a team that has had to postpone a small change without even mentioning refactors because it was too close to delivery day and considered too risky? Well, in my career about a thousand times. And now imagine a team that trusts so much with its codebase that you can change it 10 minutes before going to production. In which team would you rather be?

Defend your teenager codebase, write loads of tests, and always refactor! ■

> about the author

Augusto Evangelisti



Augusto “Gus” Evangelisti is a software development professional, blogger, foosball player with great interest in people, software quality, agile and lean practices. He enjoys cooking, eating, learning and helping agile teams exceed customer expectations while having fun.

Twitter: [@augeva](#)

Website: mysoftwarequality.wordpress.com



The Magazine for Testers, Developers and Managers

Testing Experience DE is our first German-language magazine that deals with current topics of the software development industry. Subscribe to the **free online edition** and download the first three issues, “Testing made in Germany! Was machen wir gut?“, “Der Weg zum agilen Testen“, and “Mobile App Testing”.

Call for Articles

Calling all German software wizards and gurus! Your article submissions are now being accepted for the fourth issue of Testing Experience DE, on the topic of “**Testautomatisierung**” (test automation). Share your insights and submit your articles by November 15, 2013!

www.testingexperience.de

Reuse of Unit Test Artifacts

Allow Us to Dream

by Yaron Tsubery & Dani Almog

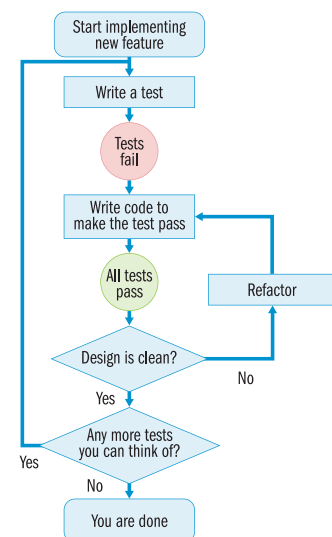
One of the main reasons for refactoring software frameworks is the need to enable reuse of the software developed. This process often occurs as a generalized act where a specific solution becomes an abstract superclass [1]. This refactoring not only clarifies the design of the framework, but ensures better consistency by defining the abstraction in one place. The concrete classes retain the behavior, although it is now inherited rather than being locally defined. This wisdom and other observations on the nature of software refactoring were first presented by WILLIAM F. OPDYKE in his doctorate thesis (1992). TDD involves short, rapid iterations of “write a test, write the code to make the test pass, and refactor”. These short iterations provide rapid feedback. Refactoring of both the test and code ensures that everything is performed to ensure simplicity and readability of the emerging code. In other words, refactoring is a transformation that preserves the external behavior of a program and improves its internal quality[2], and there is some concern to ensure refactoring is not actually rework. Needless to say, while writing the unit test the developers have the white-box testing techniques that assist them in ensuring a wide coverage of the written code. Nowadays the world is much more advanced through the recent TDD approach where Refactoring of the code has become a mandatory part of the development process. The goal of this article is to suggest a similar approach to unit test code. Here we try to shift the reuse into expanding the functionality of the unit test. The importance of refactoring the unit test is increasing and there is the need to use the White-box Test Techniques, because in some cases it is only after you have implemented the code that the developer understands which technique needs to be implemented (e.g. decision coverage or branch coverage).

Test-driven development (TDD) is a disciplined development practice that involves writing automated unit tests prior to writing the unit under test [3]. By writing a test first, the software developer must make detailed design decisions such as determining the interface and expected behavior of a unit before actually implementing the unit.

Traditionally TDD focuses on unit tests (methods and classes) and occurs primarily in the software construction phase, often following some level of requirements engineering and software architecture definition.

Sometimes unit test patterns are not enough. As a result, developers have to refactor their code to make it ‘testable’. Examples of code that needs to change include:

- Singleton classes
- Calls to static members
- Objects that do not implement an interface (as required for the mock object pattern)
- Objects that are instantiated in the code being tested
- Objects that are not passed to the method (as required for mock object pattern)



The main problem is that refactoring without unit tests to make the code testable, e.g. for the benefit of writing tests for it, does not make sense. It is risky and costly.

Exploring the origin for the unit test will make it easier to understand the current situation and status of tools, and its usage:

Some history. At the beginning of the software development, when attempting to test a certain functionality, the testers were faced with the challenge of needing to have the program ready and operational before attempting to actually execute the test (the compiler would not let you execute the code before completing all the necessary declaration and build of all affiliated infrastructure). Only then was the tester able to perform the specific test. Apart from running the program in a debug mode, when we needed to test the precise functionality we had to develop an isolation mechanism to ensure we were testing a specific code behavior. In order to do so we had to develop new code to mask the tested unit and inject artificial information into the tested object so the program would be executed

(this sometime called mock mechanism). In many cases, the tested program and the actual final code were very different. It was only when the world moved into interpreter mode program execution that isolation was enabled more naturally and unit test infrastructure appeared.

Unit test infrastructure was designed as a key element to enable isolation of the tested code prior to the full implementation of each object. Today we have literally hundreds of similar tools and add-ons for almost every software development environment. The following is just an example of the many Unit-derived test tools [4].

xUnit Family Members

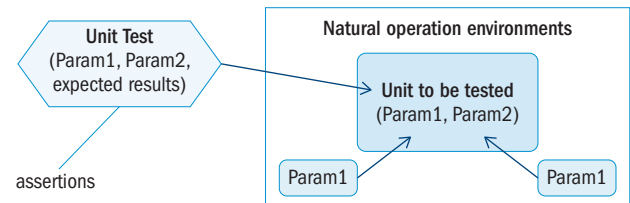
| | |
|---------|--|
| JUnit | The reference implementation of xUnit, JUnit is by far the most widely used and extended unit test framework. It is implemented in and used with Java |
| CppUnit | The C++ port of JUnit, it closely follows the JUnit model. |
| NUnit | The Unit for .NET. Rather than being a direct port of JUnit, it has a .NET-specific implementation that generally follows the xUnit model. It is written in C# and can be used to test any .NET language, including C#, VB.Net, J#, and Managed C++. |
| PyUnit | The Python version of xUnit. It is included as a standard component of Python 2.1 |
| SUnit | Also known as SmalltalkUnit, this is the original xUnit, and the basis of the xUnit architecture. It is written in and used with the Smalltalk language. |
| vbUnit | vbUnit is xUnit for Visual Basic (VB). It is written in VB and supports building unit tests in VB and COM development. |
| utPLSQL | utPLSQL is xUnit for Oracle's PL/SQL language. It is written in and used with PL/SQL. |
| MinUnit | A great example of a minimal but functional unit test framework. It is implemented in three lines of C and is used to test C code. |

xUnit Extensions

Beyond the xUnits themselves, many add-on tools are available that extend the functionality of existing unit test frameworks into specialized domains, rather than acting as standalone tools.

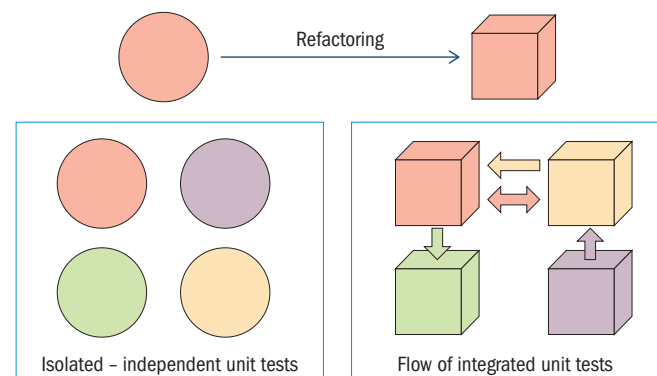
| | |
|------------|---|
| XMLUnit | An xUnit extension to support XML testing. Versions exist as extensions to both JUnit and NUnit. |
| JUnitPerf | A JUnit extension that supports writing code performance and scalability tests. It is written in and used with Java. |
| Cactus | A JUnit extension for unit testing server-side code such as servlets, JSPs, or EJBs. It is written in and used with Java. |
| JFCUnit | A JUnit extension that supports writing GUI tests for Java Swing applications. It is written in and used with Java. |
| NUnitForms | An NUnit extension that supports GUI tests of Windows Forms applications. It is written in C# and can be used with any .NET language. |
| HTMLUnit | An extension to JUnit that tests web-based applications. It simulates a web browser, and is oriented towards writing tests that deal with HTML pages. |
| HTTPUnit | Another JUnit extension that tests web-based applications. It is oriented towards writing tests that deal with HTTP request and response objects. |
| Jester | A helpful extension to JUnit that automatically finds and reports code that is not covered by unit tests. Versions exist for Python (Pester) and NUnit (Nester). Many other code coverage tools with similar functionality exist. |

Regardless of the specific tool, the unit test principle is described in the following diagram:



Many organizations have already adapted the unit test as a standard development procedure. So a lot of effort is allocated to this task. It is our claim that most of this effort is made for one-time use regardless of the reusability nature of the unit test infrastructure. Actually most of these tools make it relatively easy to repeat the test action automatically, especially if the organization has adapted TDD as well. It is understandable, since there is no reason to repeat the tests provided nothing has changed. Remember – most of the unit will not change after the refactoring action. Since unit test is testing isolated code, there is no reason to add unit test artifacts into your regression test suite if the code has not been changed.

Does it have to be like this? Ideally we would like to see the refactoring process for the unit test code enable it to integrate with other unit test artifacts. Hoping for this to happen is like saying “We want a circle to transform into a square”. In other words, I want to transform the isolated unit test item (represented by an independent sphere) into a cubical shape where each edge connects to another cube. This means refactoring the unit test into a different type of test – one that interacts with external entities.

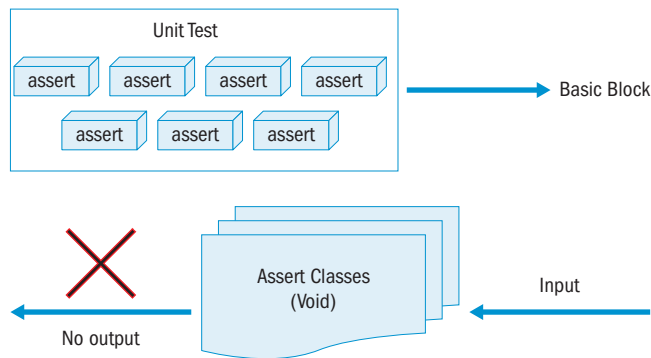


You could say: “This is a contradiction of all we have achieved; the unit test was created to isolate not integrate.” we do believe we are mature enough to try to overcome this contradiction and present two modes of operation for the unit test.

In order to better understand the implication of this change, we must explore the unit test mechanism a bit more deeply.

Assertions are the key to unit testing

We probably all know about the importance of a good unit test suite for our projects and about the Arrange-Act-Assert (AAA) pattern we should be using in our tests. The one thing we do consider a must in each and every unit test is *an assertion*.



An assertion not only decides whether the test passes or fails, it also makes the purpose of the test clear and ensures it is verifiable. A test without an assertion is nothing more than a random snippet of code. The key to this revolution is centered in the unit test isolation and assertion principles. The assertion mechanism is articulate in different forms in the different tools and infrastructures. In order to demonstrate this aspect, we will elaborate and present the two leading tools and their assertion mechanisms in the following chapter.

Assertions in NUnit

In a similar way to some small differences in NUnit (for C#), the Assert classes only deal with static arguments. As explained in: www.nunit.org/index.php?p=assertions&r=2.2.8

In the NUnit test tool, the Assert class provides the most commonly used assertions. Assert methods are grouped as follows:

Equality Assertions

These methods test whether the two arguments are equal. Overloaded methods are provided for common value types so the languages that do not automatically box values can use them directly.

Identity Assertions

- `Assert.AreSame` and `Assert.AreNotSame` test whether the same objects are referenced by the two arguments.
- `Assert.Contains` is used to test whether an object is contained in an array or list.

Comparison Assertions

The following methods test whether one object is greater than another. Contrary to the normal order of Asserts, these methods are designed to be read in the “natural” English-language or mathematical order. Thus `Assert.Greater(x, y)` asserts that `x` is greater than `y` (`x > y`).

Type Assertions

These methods allow us to make assertions about the type of an object.

- Condition tests
- Methods that test a specific condition are named for the condition they test and take the value tested as their first argument and, optionally, a message as the second.

Utility methods

Two utility methods, `Fail()` and `Ignore()`, are provided in order to allow more direct control of the test process.

StringAssert class

The `StringAssert` class provides a number of methods that are useful when examining string values.

Assertions in JUnit

In JUnit the Assert class is a static one. Formal declarations and usage example can be found at [junit.sourceforge.net/javadoc/org/junit/Assert.html#assertArrayEquals\(byte\[\], byte\[\]\)](http://junit.sourceforge.net/javadoc/org/junit/Assert.html#assertArrayEquals(byte[], byte[])) and provide a set of assertion methods useful for writing tests. Only failed assertions are recorded. These methods can be used directly.

Assert classes are defined as Void (not returning specific or internal values), so the only information we get is a pass or fail verdict.

In addition to the isolation of the unit test, there are three irritating qualities with the assertion APIs used by the major unit testing frameworks:

1. You can't use expressions.
2. We have difficulty in achieving descriptive failure messages.
3. Unit testing frameworks are not as smart as our compilers.

The root problem of all these difficulties

All of these issues are caused by the same root problem. Conceptually, unit testing frameworks are an extension of your compiler. Compilers report static errors such as malformed code and type mismatches, and unit testing frameworks report run-time errors such as unexpected values. The reason why unit testing APIs are so clunky is that our unit testing frameworks do not have access to as much information as our compiler does. Compilers have access to the expression trees generated by our code. They can analyze the code and determine what type of comparison we are attempting, and whether it is a value or reference comparison. If the compiler encounters an error it can print the exact line of code or expression responsible.

An example of current development towards reusability and ease of use for unit test is the Should Assertion Library (github.com/erichexter/Should/blob/master/README.markdown). This provides a set of extension methods for test assertions for AAA (Assemble, Act, Assert) and BDD (Behavior Driven Development) which promote `/given //when //then` style tests. It provides assertions only, and as a result it is Test-runner-agnostic. The assertions are a direct fork of the xUnit (unit.codeplex.com) test assertions. This project was born because test runners *should* be independent of the assertions!

Following this description and some experimentation that has been done, it seems the operational and internal design of unit test, having the test deals with one assert at the time, does

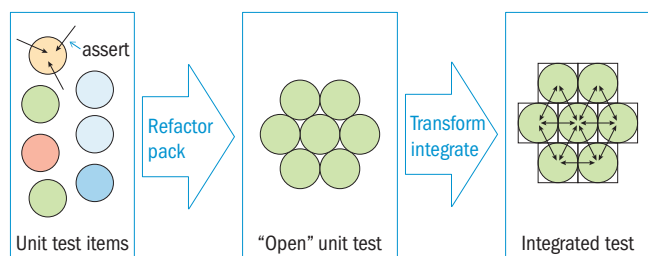
not enable us to integrate them. When attempting to reuse unit test in integrated manner, the only possible solution we currently have is to address the order and the hierarchy of the tests when trying to use and execute them, so we have a logical cover of tests for the several units and, therefore, this shows some sense in the application. *Currently we are not testing integration using unit test artifacts.*

In order to achieve this we need to *revolutionize our way of thinking* – let's call this thinking *RTF* (Right First Time). Developing test automation is like developing any other software. If the requirement of the software we are designing includes integrating tested elements – consider it from the beginning. It is time to throw down the gauntlet to the X-unit test makers.

We challenge the tools makers to develop a new generation of *I-unit* test family tools. These new tools will show new types of assertions:

- Double toggle ones between
 - The original static null assertion classes
 - Integrated assertion where partial parameters used during the assertion are replaced by the real application responses.
- Integrated assert classes where the insertion of parameters is done dynamically and the outcome of the previous test could be used at the next one.

Ideally we can see a new process resembling the software development. At the beginning you take a quick and dirty approach and then you refactor your work towards a better standardized shape.



Of course, refactoring enabling the “open” quality can come later (after performing the initial unit isolated unit test). But results from the new strategy, like the code refactor, could have been implemented earlier.

Summary

This article proposed a revolution in addressing the unit test artifacts from an isolated and single purpose (used mostly by developers) to being an integrated part of reusable testing artifacts used by all levels of development and quality assurance teams.

All new technologies started with an impossible mission – let us dream the impossible.

References

- [1] Opdyke, W.F., *Refactoring object-oriented frameworks*. 1992, University of Illinois.
- [2] Soares, G.S. Automated behavioral testing of refactoring engines. in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. 2012: ACM.
- [3] Hayes, J.H., A. Dekhtyar, and D.S. Janzen. Towards traceable test-driven development. in *Traceability in Emerging Forms of Software Engineering, 2009. TEFSE'09. ICSE Workshop on*. 2009: IEEE.
- [4] Hamill, P., *Unit Test Frameworks: Tools for High-Quality Software Development*. 2009: O'Reilly Media, Inc. ■

> about the authors

Dani Almog



Dani Almog: From Ben Gurion University – senior researcher and lecturer on Software Quality and test automation. Dani has very vast experience in the industry – former test automation managing director for Amdocs product development division. Dani is a speaker in professional and academic

international conferences such as, STARW, CAST2009, STP professional, Cise2010, STEP2012, ExpoQA-2012, Agile Testing Days, 2012.

Yaron Tsubery



Yaron Tsubery has been working in software since 1990 and has more than 20 years of experience as a test engineer and test manager. His original profession was system analyst.

Yaron is the current VP Testing Division Manager at Ness Technologies. He worked in product and

project management and development for over 3 years before becoming director of QA and testing manager at Comverse from 2000 until 2013, as well as the CEO and founder of Smartest Technologies Ltd from 2010 to 2012. Yaron was in charge of a large distributed group of testing team leaders and test engineers dealing with functional and non-functional, load and performance tests.

Yaron has wide experience in banking business processes. For the last 10 years he has implemented best practices in a field where he is in charge of producing complex systems for telecommunication companies. Yaron is the current president of ISTQB® and is also the president and founder of the ITCB. Yaron has been invited as a speaker to a number of international conferences to lecture on subjects related to testing. He has written articles that were published in professional magazines.

By the way ...

by *Tanja Schmitz-Remberg & Werner Lieblang*

Autumn had arrived. The trees were wearing a colorful dress of leaves, the sky was blue, and Jack enjoyed his walk through the fallen leaves that the wind has arranged in piles along the footpath. He felt a bit like a little boy again.

When he arrived home he met Lisa who was – as usual – having a cup of herbal tea. It smelled delicious.

“Hi,” she said, “How was your day?”

“Fine, actually great. We did some good team work. You might remember, we are currently designing the new holiday web site “off you go”. And because my colleagues did not open their mouths during the meeting with the customer, I could bring up all my ideas. Et voilà – we are going to do it exactly the way I want it to be done!”

“Well, that sounds like a real successful day for you. But what about your team, and the customer – how was it for them?”

“Actually, I think the colleagues were happy that I cut through that unpleasant silence and made some creative suggestions. And the customer – well, they listened to everything, they took my notes and we agreed on a follow up meeting for next week.”

“So you are happy and the customer as well – that’s great!” Their eyes met and Lisa could tell by the sound of his voice that not everything was fine.”

“Well...” Jack replied, “Simon, my senior colleague seemed to be a bit annoyed, but he has mood swings anyway. And Sarah, our young one, was very quiet – but that is typical for her. And the customer – hmm, that’s difficult for me to judge. But I think it’s a good sign that they want to see us again next week in order to see how we are doing as a team.”

Lisa sipped her cup of tea. She was wondering why Jack was not able to see that he had lost track of the others. In his world everything seemed to be okay.

“By the way, we still have to prepare our garden for the cold season,” she said, looking up from her cup. “Some of our plants will make it easily through winter, whereas others cannot handle temperatures below zero, so we urgently have to shift some of these into the cellar before it gets colder.”

Jack felt this change of subject had taken him harshly out of his state of self-contentment. He turned towards Lisa and dutifully asked, “Right – but how do we know which plant needs what?”

Lisa smiled, “Well, we cannot ask them, can we? It’s up to us to find out. I know what some of them need, but for others we have to investigate carefully. We want to make sure that none of them gets harmed accidentally. They are all very different.”

“That’s right”, Jack thought. It would be so much easier if they could all be treated the same. He opened his notebook and started his research. And in less than one hour he had all the facts together and knew what to do.

Lisa brought him a new cup of tea. She was pleased with her psychology lesson that had made Jack realize his fault. “So, what have you learnt?” she asked.

“How do gardeners do their job”, Jack sighed, “Every plant needs a different treatment. I could never keep all of that in my head. I’m glad that handling my people is not that difficult!”

Lisa dropped her cup and it shattered on the paved floor. ■

> about the authors

Tanja Schmitz-Remberg and Werner Lieblang



Tanja Schmitz-Remberg and Werner Lieblang combine half a century of experience in software engineering and training various groups. Werner is working as a tester trainer and agile coach, Tanja works as a communication and group work trainer.



Being friends for ages and sharing an enthusiasm for working with agile groups, they passionately play impro-theatre and love to dare out-of-the-box stuff. Together they have developed several trainings for members of the agile community.

Three Tips for Test Refactoring

by Gil Zilberfeld

Refactoring is well understood today. It is done either seamlessly as we write with automatic refactoring tools, or we can refactor our production code manually with the aid of tests, so it will be easier to read and maintain over time.

Production code gets all the refactoring glory and fun, but what about its sibling – the test code? Tests need refactoring too!

Production vs. test code

They look similar, since we use the same tools and languages. However, production code and test code are different. The difference lies not in semantics, but rather in the purpose of the code. With production code we solve a functional problem: sort an array, complete an operation, or shoot birds to hurt unsuspecting pigs.

Tests have a different goal altogether. In fact, two goals. The first is to tell us if something went wrong – functionality that worked before has stopped working. The second goal is to help us analyze what went wrong and fix it as quickly as possible.

TDD tests, additionally, help us in the design of the production code, but this benefit is irrelevant to our refactoring discussion. Even in TDD, you start refactoring when all tests pass, not when you have a red test. But then we still attribute the refactoring to the production code, now protected by the tests.

So if tests serve a different purpose, do they perhaps have different refactoring rules?

What's in a name?

Test names are important. In fact, they are the second most important thing about the test. (If you are wondering what is #1, it is that the test is testing the correct behavior). “Why?” you ask.

A test lives as long as your application does. That means years, if not decades. And five years from now it is going to happen that a programmer who is not even working in this company yet will break existing functionality. All he has to start with is a list of test names. Once a test fails, it does its job best getting you from “found a bug” to “fixed a bug” as quickly as possible, and the test name is the first clue. It should tell you as much as it can.

For example: `test13` is not a good name for a test. It also brings bad luck.

The existence of refactoring is the acknowledgement that we cannot write our best code straight off. We need to iterate until we find the best suitable design. We can apply the same acknowledgement to test naming, too. We do not usually find the best name the first time we write it. In fact, it is only when we have a bunch of tests surrounding different cases sur-

rounding a functionality that we have enough information to craft their names.

So the obvious conclusion is that renaming a test (much like renaming methods and variables) is something we want to do iteratively, until we believe that it will give us enough information to close the bug investigation quickly.

How do we know that the test name works? There are a couple of questions you want to ask yourself:

- Is this name readable to me?
- Does it describe the test in terms of context and expected result?
- Is the name of this test differentiated enough from its other brothers that test close yet different cases?
- If this test alone fails, and its brothers pass, can I understand the problem?

If you answered “yes” to all of the above, you can move to the actual exam. Grab someone who does not work on the feature and ask him the same questions. The longer we wait, the greater the chances are that we are not going to be the ones who have to deal with the failing tests. A second set of eyes can give you the feedback you need.

Divide and conquer

If your test name looks like `Recover_Password_Form` there is a smell. Unless you are really starting out, this name usually means that you are testing big workflows or long scenarios, and that there is not just one thing to assert. That is good for integration tests and bad for unit tests.

The way tests have worked for ages is that once something is wrong (an assertion failure, or a thrown exception), the test does not continue. So if you have multiple assertions and the first one fails, you are left with this knowledge alone, without any information about the rest of the test.

```
1 [Test]
2 public void Authenticated_Index() {
3     var authenticatedUser = new User();
4     authenticatedUser.Email = "authenticated@email.com";
5     var m_Controller = CreateController<AccountController>
6         (authenticatedUser);
7     var authorized = ControllerActionInvoker<ViewResult>().
8         InvokeAction (m_Controller.ControllerContext,
9             "Index");
10    Assert.AreEqual(true, authorized);
11    var result = m_Controller.Index() as System.Web.Mvc.
12        ViewResult;
13    Assert.AreEqual(result.ViewName, string.Empty);
14    var user = result.ViewData.Model as ERPStore.Models.
15        User;
16    Assert.AreEqual(user.Email, "authenticated@email.com");
17 }
```


As you can see in this example, we are testing three different assert criteria: authorized is true, view name is empty, and user's email is what we originally put in. No wonder we cannot find a good name for the test!

And, if the first Assert fails, we do not have results for the other two. We are missing information that could have been enough for us to understand the problem. Our way to "fix the bug" just got longer.

Instead, we can refactor our encompassing test into separate tests, like these:

```
1 [Test]
2 public void ControllerInvokedIndex_AuthenticatedUser_
   True() {
3     var authenticatedUser = new User();
4     var m_Controller = CreateController<AccountController>
       (authenticatedUser);
5     var authorized = ControllerActionInvoker<ViewResult>().
       InvokeAction(m_Controller.ControllerContext,
           "Index");
6     Assert.AreEqual(true, authorized);
7 }

1 [Test]
2 public void Controller_ViewNameOfNewUser_IsEmpty() {
3     var authenticatedUser = new User();
4     var m_Controller = CreateController<AccountController>
       (authenticatedUser);
5     var result = m_Controller.Index() as System.Web.Mvc.
       ViewResult;
6     Assert.AreEqual(result.ViewName, string.Empty);
7 }

1 [Test]
2 public void Controller_AuthenticatedUser_EmailIsFilled(){
3     var authenticatedUser = new User();
4     authenticatedUser.Email = "authenticated@email.com";
5     var m_Controller = CreateController<AccountController>
       (authenticatedUser);
6     var result = m_Controller.Index() as System.Web.Mvc.
       ViewResult;
7     var user = result.ViewData.Model as User;
8     Assert.AreEqual(user.Email, "authenticated@email.com");
9 }
```

Now if one of them fails, we will know how the others fare. This will give us more information that can help pinpoint the bug and fix it quickly. And we have managed to find more accurate names in the process.

The WET principle

We all know about the DRY (don't repeat yourself) principle. We know that duplicated code is evil, and for good reasons. When you need to change code, it is in one place – no need to look for it, or remember (or usually forget) where you put it last.

But if tests are not like production code (we know that already), does that mean that DRY does not apply to them?

I have passed through a couple of learning iterations going through this topic. I finally decided that WET is better than DRY: Write Expressive Tests.

What does that mean? Consider our three refactored tests from the previous section. They have the same setup:

```
1 var authenticatedUser = new User();
2 var m_Controller = CreateController<AccountController>
   (authenticatedUser);
```

DRY fanatics will tell you to extract the initialization into the setup method, like this:

```
1 [Setup]
2 public void Setup() {
3     authenticatedUser = new User();
4     m_Controller = CreateController<AccountController>
       (authenticatedUser);
5 }
```

In real life, tests will be long, setup is long, and there are many tests in that file. Now let's imagine a test failing. You look at the failing test, but the initialization is not there! It is in the Setup method. To compare, it is like reading a mystery book starting from the middle. Now you need to search for the beginning, then maybe jump and scroll around the code until you get a picture of what the context is.

There is a better way.

Good tests lead you quickly and closely to the problem. Applying DRY may not be the right choice. Ask yourself and, preferably, a colleague whether the test is readable, whether it tells the story clearly. Expressive tests that tell the story are better. If not, prefer extracting initialization to private methods, rather than moving them into separate setup methods which are used by the framework. Private methods create some continuity, while framework setups break it.

Conclusion

Refactoring is a means to an end: better code. With production code, it is clearer and meaningful. In test code, it is about making tests as helpful as possible, so when they fail they can quickly lead you to the bug and its solution. ■

> about the author

Gil @gil_zilberfeld



Gil Zilberfeld has been in software since childhood, starting out with Logo turtles. With twenty years of developing commercial software, he has vast experience in software methodology and practices. Gil is the product manager at Typemock, working as part of an agile team in an agile company, creating tools for agile developers. He promotes unit testing and other design practices, down-to-earth agile methods, and some incredibly cool tools. Gil speaks in local and international venues about unit testing, TDD, and agile practices and communication. And in his spare time he shoots zombies, for fun. Gil blogs at www.gilzilberfeld.com on different agile topics, including processes, communication and unit testing.

Twitter: [@gil_zilberfeld](https://twitter.com/gil_zilberfeld)



CaseMaker® SAAS

Quit struggling
with the decision
and enjoy test
case design in
the cloud today!

CaseMaker SaaS systematically supports test case design by covering the techniques taught in the ISTQB® Certified Tester program and standardized within the British Standard BS 7925. The implemented techniques are: equivalence partitioning, boundary check, error guessing, decision tables, pairwise testing, and risk-based testing.

CaseMaker SaaS is your perfect fit between requirement management and test management/test automation.

saas.casemaker.eu



Subscribe today
and enjoy a 14-day
trial period for free!

Your license starts at 75 €/month (+ VAT).



Business First, Not Test First: How to Create Business Value from Acceptance Tests

by Dr. Chaehan So

1. The Problem

Acceptance Test Driven Development (ATDD) has emerged and rapidly gained popularity in recent years. Acceptance tests serve to verify the system interaction from the user perspective (“what”) and thus abstract the technical implementation level (“how”).

Although the advantages of ATDD are uncontested in the agile software community, many IT managers are still struggling to convince the business side to fund ATDD. The reason for this business reluctance is twofold:

First, customers view ATDD as scope with no business value that conflicts with features which they do associate with business value. Second, it is often the timeline that dictates the trade-offs between scope and quality decisions. Most customers have a traditional QA team in the plan that covers quality issues at project end.

As a result, many customers trade off a sustainable level of quality in favor of more features because they associate more business value with the latter. If IT managers or agile teams want to convince business to decide the other way, the solution is quite simple: they need to create business value from ATDD.

Although this goal can be set quickly, it is not easily attained. The reason why business people do not see the business value in ATDD lies in its nature. It is designed to provide test engineers and software engineers with the technical details for debugging. It hence follows that the only way to convince customers to give higher priority to ATDD is (a) to make them see something that they haven’t seen in it before and (b) they associate with business value.

2. The Solution

The key to the solution is to view ATDD from a customer’s perspective. As customers are not involved in the technical processes of continuous integration and debugging, they do not see the immediate benefit of ATDD. To them, the only visible parts of ATDD are the test reports.

However, ATDD proponents often neglect the fact that many customers find the format of many tools too difficult to understand. For example, many tools rely on a tabular format like FitNesse, or an abstract syntax like “Gherkin” as used by the Cucumber tool. These tools provide an easy way to read and add new

cases with the same requirement in the corresponding tables. Yet, people perceive this requirement representation more as a *configuration* of the requirement than the requirement itself.

The next step to the solution is to apply a psychological view to the problem. A human mind can only perceive things that are already represented in its mental model. If test reports are the only visible part of the ATDD process to a customer, and the information contained in these test reports are not part of her mental model, it is evident that she does not attribute any value to it. Nevertheless, it is easy to find the customer’s mental model what matters most to her – requirements – and whether they have been or will be implemented.

Under condition that we can map the requirements world to the testing world in such a way that it reflects what the customer wants to know – the status of the requirement implementation – it follows that we can actually create business value which the customer has previously not perceived.

Presenting this requirement status information in an intuitive manner could further increase the new business value of such a testing framework. In addition to business and IT top management, the development team could benefit from such a testing framework because it alleviates their burden of generating and aggregating information solely for top management reporting.

I propose a solution for such an acceptance test framework that embodies the principle “design to communicate” and consists of two building blocks:

- a. Mapping requirements to the testing world
- b. Simple design

3. Mapping Requirements to the Testing World

One way of realizing ATDD is to implement test cases in user story syntax. This concept is called “executable acceptance tests” and communicates the acceptance tests in the form of user stories and linked user scenarios to the customer. To implement executable acceptance tests in an agile project, we need to follow two steps:

1. Build different requirement abstraction layers
2. Synchronize the acceptance test model with the requirements model

Step 1: Build Different Requirement Abstraction Layers

The majority of SCRUM projects model requirements in user stories. However, for business-critical features it is more practical to model requirements in use cases because use case scenarios provide a level of process detail that user stories do not specify.

Figure 1 shows a naming convention that allows to extend user stories into use cases in a seamless manner:

- The title remains identical (e.g. “as an iPhone user, I want to read an article”)
- The original scenario of the user story is titled “main success scenario” (e.g. “iPhone user reads news article”)
- The extensions of the main success scenario are labeled “alternative success scenarios” and “failure scenarios” according to conventional use case modeling

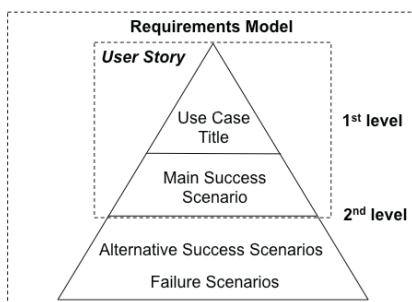


Figure 1. Requirement Model Linking User Stories to Use Case Scenarios

This naming convention creates a robust requirements architecture that accounts for all levels of requirement maturity and business criticality.

A team models and implements a first version of a user story that contains the user story title (1st level) and the main success scenario (2nd level). This, in essence, represents a stripped-down version of a use case that the team can extend with alternative and failure scenarios in subsequent iterations.

Step 2: Synchronize the Acceptance Test Model with the Requirements Model

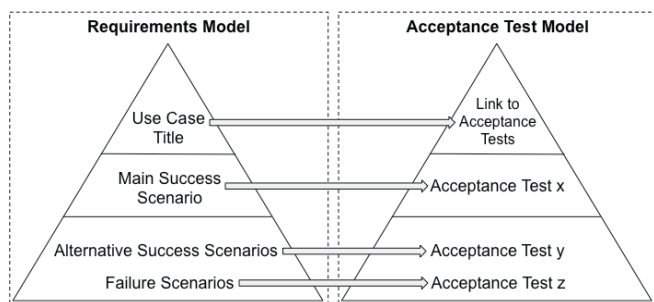


Figure 2. Synchronized Acceptance Test Architecture

After Step 1, we have all the requirements in a user story-use case scenario structure. Step 2 consists of aligning the accep-

tance test model exactly with the structure of the requirements model (see Figure 2).

To ensure the alignment, we need to label the acceptance tests identically to the corresponding requirements. This is critical because the customer can then perceive the resulting acceptance tests directly as requirements due to the identical labeling. In other words, the acceptance tests do not merely *reference* the requirements, they *are* the requirements from the customer's perspective (see Figure 3).

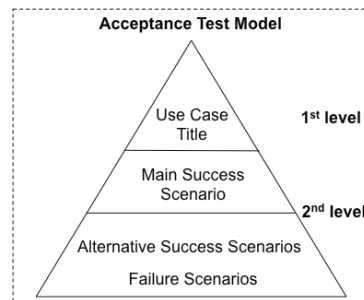


Figure 3. Customer's Perception of the Acceptance Test Model

4. Simple Design

After building the synchronized acceptance test model, the crucial factor for creating business value for customers is to abstract and hide all layers of complexity until we can present them with the big picture of all the requirements in the simplest way.

The 1st level (titles of user stories for each use case) communicates an aggregated status (green or red) that reflects the summarized view of all underlying use case scenarios of the 2nd level. The latter is used in product demos after additional scenarios have been added to the initial requirement.

The advantage of building acceptance tests on the requirements abstraction hierarchy of Step 1 is that the team can easily integrate the evolving maturity states of requirements along a project's timeline (cf. Figure 4) with the presentation of business value in the form of implemented acceptance tests.

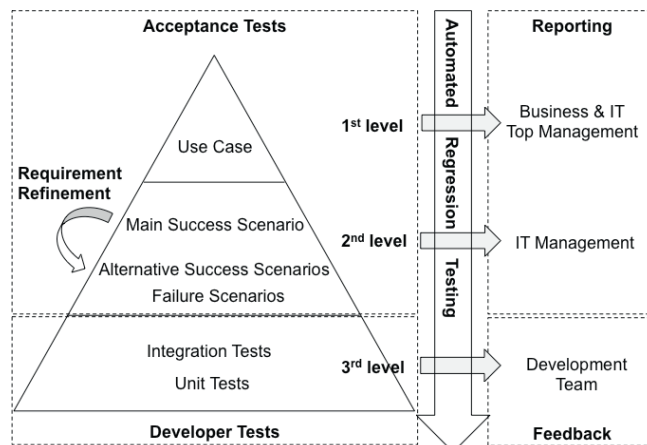


Figure 4. Conceptual Model

5. Results

The depicted testing framework and its underlying principle “design to communicate” quickly began to facilitate management reporting on project progress, and served as a monitoring tool after go-live.

Customers at all management levels, even including one board member, stated that they found the framework’s design very intuitive to understand. Some emphasized the value of using natural language without any technical terminology. Other team members attributed the ease of understanding to the simple design: green and red icons structured into a simple matrix of use cases vs. channels. This overview page only contained the names of the user stories, which in turn linked to the use case scenarios.

IT top management saw the framework’s benefit as delivering top-level status reports about project progress. As a consequence, the CIO approved the project proposal of another internal department to implement the framework and tailor it to their needs.

We found that building the test framework was easy and did not take much time to build in its initial stage. Yet, we encountered drawbacks in getting new-hires up to speed, so we decided to switch from the Robot framework to Cucumber because most developers found Cucumber integrated better into Eclipse. The team tried but eventually did not follow the “test first” approach in ATDD, i.e. start by writing an acceptance test for a given requirement entity (user story, acceptance criterion), and then write the corresponding implementation of the code until the acceptance tests passes.

6. Summary

This article addresses many companies that have customers who do not see a direct benefit of ATDD and thus are reluctant to approve the corresponding budget. For such customers, the proposed solution may be a vehicle to demonstrate business value in ATDD. The underlying principle “design to communicate” is a step-by-step recipe how to transform acceptance tests until the customer perceives them as requirements.

I conceptualized the solution and had it implemented in an industrial setting in a major IT project with over 50 project members. Not only did the customer approve that we used this framework to communicate project progress in terms of implemented requirements. It also served as a major feedback tool for all software engineers. By keeping everybody focused on delivery, the framework became a major project success factor.

It is important to keep in mind that “design to communicate” goes beyond the implementation of executable acceptance tests. Without a deep understanding of cognitive information processing, it is not possible to conceptualize “simple design”. Only then can we shield the complexity and structure of different

abstraction layers, cross browser tests, and multiple mobile device tests from the big picture view that is most beneficial to top management.

Last but not least, the visibility of the acceptance test results was a major driver for business approval. Placing the acceptance test result view near the project requirement documentation in a wiki may have served as the final twist to increase business approval. ■

> about the author

Dr. Chaehan So



Chaehan So is an agile coach and consultant since 1999. He specializes in transforming large organizations to agile.

His emphasis on real psychological insight is how he differs from mainstream agile consultants. He draws on his psychological research (tinyurl.com/agileTeamwork) in how agile software development teams become

effective through motivational and learning mechanisms. Dr. So developed the first psychological measurement instruments for agile practices (www.psychologie.hu-berlin.de/personal/7777269/PAM) which won him a best paper award at the XP 2009 conference. Apart from his Ph.D. in Psychology, he holds master’s degrees in engineering (Technical University Berlin) and business (Ecole Supérieure de Commerce de Paris).

Chaehan So’s first encounter with agile software development dates back to his Silicon Valley experience at Netscape in 1997 and to research conducted at Stanford University in 1998.

Explorative C# Web Scripting

Using *scriptcs* and *FluentAutomation*

by Vagif Abilov

The dilemma of automating unstable UI

If you write acceptance tests for Web applications, no matter how you arrange your work, you have to spend some time on figuring out how to reach certain UI elements and how to make them perform the requested action. While you can navigate and command these elements directly from your tests, this is considered to be bad practice that makes tests both brittle and hard to read. The better approach is to build a test automation framework around the Web application under test, so acceptance tests will only access it via its automation API.

Test automation frameworks for Web applications have their own challenges. Firstly, if the framework is developed in one of the traditional object-oriented languages, such as C# or Java, the process of inspecting Web UI elements and embedding their CSS or XPath selectors into program code may not be very efficient. Programmatic management of DOM elements involves some trials and failures, and these languages usually lack REPL tools that let developers focus on single statement execution without wrapping those statements in classes and modules. Secondly, in an iterative development process the structure of a Web application under development will continuously change, so attempts to build an automation API around it will face not just revisions but sometimes full rewrites.

Here comes the dilemma. If you practice BDD, ATDD, or any flavor of development methodology that tries to reduce the gap between requirements, specifications, and programming code, then you will try to automate early validation of UI, perhaps by making specifications directly executable (e.g. writing them in Gherkin and running in Cucumber or one of its flavors). So you will need to provide UI automation for your Web sites. But again and again you will come to work in the morning to find out that full nightly test of your system failed: div IDs changed, CSS classes were renamed, or maybe the entire page has been turned into a tab inside another page, so the way you structured your automation API no longer makes sense. And such test failures do not indicate a lack of following TDD principles because these are *not* unit tests. UI design, Web programming logic, and acceptance tests are usually in the hands of different people, so the best that can be done to improve the stability of UI tests is to make them more responsive to frequent and, sometimes, radical changes. And the changes may come both from higher (specifications) and lower (UI elements) abstraction levels of the system under development.

From pages and elements to expression of intentions

So how can we make a Web test automation framework more responsive to frequent changes? Let's have a quick look at a simple C# code example that automates a Google search:

```
1 public class SearchPage {
2     private readonly IWebDriver _driver;
3     public SearchPage(IWebDriver driver) {
4         _driver = driver;
5     }
6     public void Goto() {
7         _driver.Navigate().GoToUrl("http://google.com");
8     }
9     public IEnumerable<IWebElement> Search(string text) {
10         var search = _driver.FindElement(By.Id("gbqfq"));
11         search.SendKeys(text);
12         var button = _driver.FindElement(By.Id("gbqfb"));
13         button.Click();
14         var resultsPanel = _driver.FindElement(By.
15             Id("search"));
16         return resultsPanel.FindElements(By.XPath("./a"));
17     }
18 }
```

The code above uses Selenium Web Driver and should be familiar to anyone who has worked with Web UI test automation. The code uses a Page pattern that encapsulates details of specific UI elements and exposes action results. If we want to avoid tests and deal with *IWebElement* objects directly, we could change the return type of the *Search* method so it would extract strings from the *IWebElement* collection before returning results. The test validating the Google search page functionality might contain the following code:

```
1 var driver = new ChromeDriver();
2 var searchPage = new SearchPage(driver);
3 searchPage.Goto();
4 var results = searchPage.Search("Agile Testing Days");
```

Is this code simple? Yes it is. Does it encapsulate UI element access details? Yes it does. Then what can be wrong with this code?

Well, there is nothing *wrong* with this code, especially since it works. But let's imagine we are automating access not to Google (where the key UI elements in its front page are not going to change any time soon), but to a page that is a part of a Web site developed by a startup company planning to offer services to its members within a few months. How good are the chances that tomorrow this page will still have an element

with id "gbqfq"? How good are the chances that the Web application will still have a dedicated search page instead of a placing a search section into a common toolbar accessible across the whole Web site?

Since we are dealing with the search functionality, the application under development most likely has a user story describing the search feature. Expressed in Gherkin, it might look like this:

Feature: Search for professional events
In order to keep updated about professional events
As a service subscriber
I want to be able to search for conferences

Scenario: Search for professional conference Web sites
Given I am a service subscriber
When I search for "Agile Testing Days"
Then I should receive results starting with
"www.agiletestingdays.com"

At first glance, the difference between the Gherkin specification and the exposed Web test automation UI is not significant, because human readable text and programming code will always be different and it will not take that many lines of code to bridge Given-When-Then statements and SearchPage class. But what if our programming code had a closer match of both our intentions and steps required to achieve them? Would we need the SearchPage class at all if we could write something like this:

```
1 I.SearchForConferences("Agile Testing Days");
2 I.ExpectSearchResult("www.agiletestingdays.com");
```

... and the SearchForConferences method implementation would simply list the sequence of steps to perform this action:

```
1 public void SearchForConferences(string text) {
2     I.Enter(text).In("#gbqfq");
3     I.Click("#gbqfb");
4 }
5 public void ExpectSearchResult(string text) {
6     I.Expect.Text(text).In("#rso li cite");
7 }
```

Now, we have not eliminated the Web automation layer – but we have nearly eliminated the *translation* layer that we had to keep between our specifications and the Web driver that transformed the steps needed to achieve our intentions into an API exposed by the test automation framework. Now it looks like what we do is the automation API. And what we want to achieve in user story scenarios is expressed as the sequence of steps performed by *us* – as opposed to a *driver* or a *page*. So we can record a script of what we are doing and group script lines to match scenario steps of user stories – and we will have an internal DSL that we can use to command our Web application. So what can bring us such scripting capabilities?

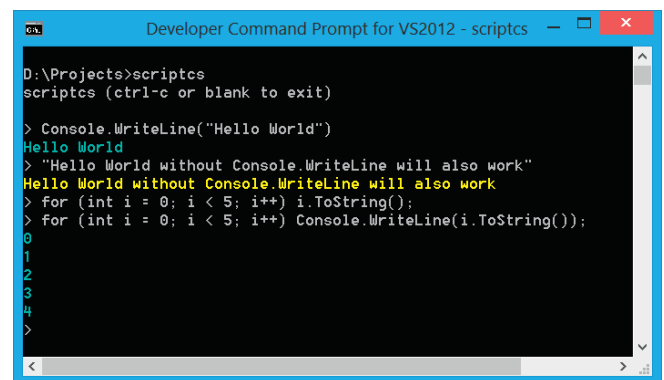
Enter **scriptcs**: an open source project that enhances C# with an REPL (read-eval-print loop) tool. Enter **FluentAutomation**: an open source project that wraps Selenium Web Driver API in a fluent interface closely resembling user interactions and extensible to expose higher level user intentions.

Getting started with scriptcs and FluentAutomation

A detailed introduction to scriptcs and FluentAutomation is outside the scope of this article, so we will just list the steps required to get both products up and running. The easiest way to install scriptcs is by requesting it from *Chocolatey* – a Windows analog of apt-get. If you do not have Chocolatey, grab it from chocolatey.org – it only takes a few minutes. Once Chocolatey is installed, open the command line Window and write the following:

```
1 cinst scriptcs
```

This will install scriptcs on your machine, so you can start scripting in C#. To start a scripting session, type *scriptcs* in a command line prompt (you may need to open a new command line window to get scriptcs into PATH environment variable). Once scriptcs is up, you can start invoking C# statements:



```
D:\Projects>scriptcs
scriptcs (ctrl-c or blank to exit)

> Console.WriteLine("Hello World")
Hello World
> "Hello World without Console.WriteLine will also work"
Hello World without Console.WriteLine will also work
> for (int i = 0; i < 5; i++) i.ToString();
0
1
2
3
4
>
```

Scriptcs is both simple to use and powerful. By using C# compiler services (codenamed Roslyn) internally, it turns C# and the whole of .NET with any third-party libraries into a scripting engine.

Our third-party library for this story will be FluentAutomation. In the previous section we already showed a few examples of its API modeled after user interactions ("I.Open", "I.Click", "I.Enter(...).In(...)", "I.Expect"). The easiest way to obtain the FluentAutomation library is via NuGet, and scriptcs uses NuGet to load additional components.

To add FluentAutomation to scriptcs, type the following command in a command line window (since scriptcs will be downloading some packages, it is recommended that you do it in a directory dedicated to the scripting session):

```
1 scriptcs FluentAutomation.SeleniumWebDriver
```

FluentAutomation requires you to choose a specific Web driver to command Web pages, so we installed a NuGet package for Selenium. In addition, we will need to specify a browser to be used with the Web driver. This can be done from the actual scripting session.

Scripting the Web

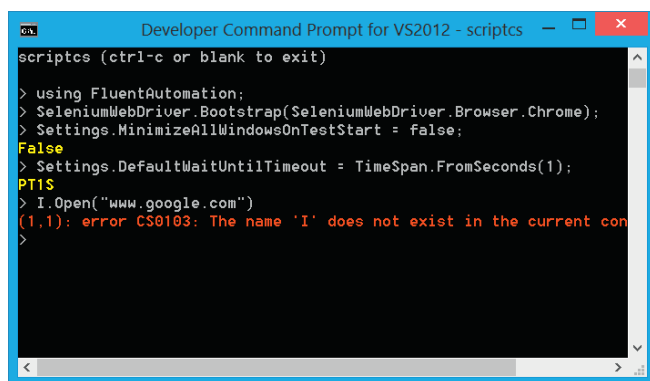
After restarting scriptcs type the following commands:

```
1 using FluentAutomation;
2 SeleniumWebDriver.Bootstrap(SeleniumWebDriver.Browser.
  Chrome);
3 Settings.MinimizeAllWindowsOnTestStart = false;
4 Settings.DefaultWaitUntilTimeout = TimeSpan.
  FromSeconds(1);
```

The first line of the script will import FluentAutomation (just like in a regular C# file, so we do not need to use fully qualified class names). The rest of the lines configure browser session parameters. Once these preparations have been made, we might be tempted to start using fluent Web API and write the following statement:

```
1 I.Open("http://www.google.com")
```

But this will not work yet, scriptcs will complain:



Don't worry – the fix is simple. It is just that FluentAutomation expects its fluent API to be used from inside a class that inherits from a FluentTest base class. But we are scripting, we do not need to declare any classes, so we will simply define a variable I inside our script:

```
1 var I = new FluentTest().I;
```

This will do – now we can start commanding Google!

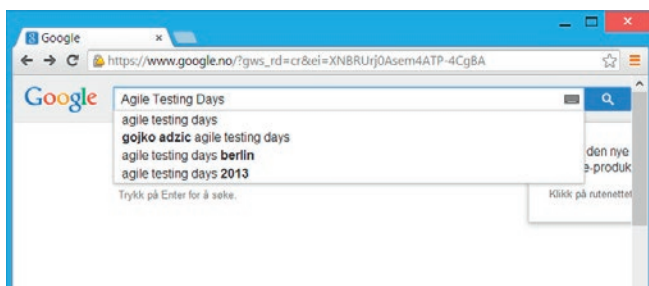
```
1 I.Open("http://www.google.com")
```

... and a new Chrome window is created and points to a Google front page.

Now we just need to send C# commands – using FluentAutomation API. We type

```
1 I.Enter(text).In("#gbqfq")
```

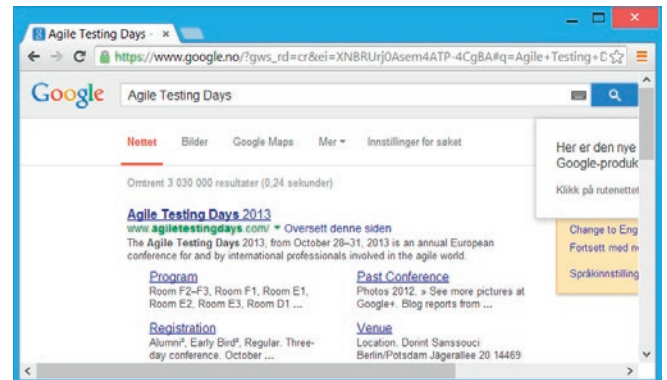
... and Google opens its search text combo:



We continue with the button click:

```
1 I.Click("#gbqfb")
```

... and the result page appears:



We can try now to assert for a match to “www.agiletestingdays.com”:

```
1 I.Expect.Text("www.agiletestingdays.com").In("#rso li cite")
```

But the scriptcs engine will show an exception raised by FluentAutomation: the specified element does not contain the expected text. If we have a closer look at the results, we will understand why. In the resulting text “www.agiletestingdays.com”, Google highlighted the “agiletestingdays” part so the element contains a combination of parts of the URL and formatting options. But we may refine our match criteria, and we do not need to restart or recompile anything. Even after the thrown exception we can continue sending new script commands. The following will succeed:

```
1 I.Expect.True(x => x.Contains("agiletestingdays")).
  In("#rso li cite")
```

The statement above includes a lambda-expression, this is because our expectation is based not just on a simple text comparison, but requires an invocation of a delegate function (string.Contains). So instead of passing a string literal, we need to pass a delegate method containing the match logic.

You can see how easy it is to explore and automate Web sites without leaving a scriptcs REPL window. We can send FluentAutomation commands, validate execution results, and, in the event of failure, inspect UI elements and re-try commands with corrections.

From scripts to internal DSL

But perhaps the greatest efficiency is reached at the stage of building a domain-specific Web test automation framework. Because our script statements are written in C#, we can just copy and paste them as a body of higher-level methods. Remember the SearchForConferences shown in the first section? Its code was very close to a real one that needs to be implemented as extension methods for the INativeActionSyntaxProvider interface:

```

1 public static class ExtensionMethods {
2     public static void SearchForConferences(
3         INativeActionSyntaxProvider I, string text) {
4         I.Enter(text).In("#gbqfq");
5         I.Click("#gbqfb");
6     }
7     public static void ExpectSearchResult(
8         INativeActionSyntaxProvider I, string text) {
9         I.Expect.True(() => I.Find("#rso li cite")().Text.
10            Contains(text));
11     }
12 }

```

Use of C# extension methods makes it possible to extend original actions available for the “I” role with new ones that come from our product specifications and express actions and expectations from user story scenarios. The set of extension methods forms an internal DSL and effectively becomes our Web test automation framework. Forthcoming changes – whether they come from specifications or user interface design – should have a smaller impact on such DSL than if it exposed its own proprietary API. And updating the DSL should be less time-consuming – thanks to scriptcs and FluentAutomation.

Conclusion

In this article we had a quick look at fairly new (launched in 2012–2013) open source projects: scriptcs and FluentAutomation. Scriptcs one provides a REPL environment for extensible C# scripting and is highly recommended as a canvas for acceptance and integration tests. FluentAutomation is specific to Web testing but is a perfect match for the scriptcs environment because its syntax resembles complete user interactions. FluentAutomation API can also easily be extended with custom user actions to match steps from user story scenarios. In combination, scriptcs and FluentAutomation provide an efficient tool for .NET developers and testers to implement executable specifications and acceptance tests.

Resources

[1] Scriptcs. <http://scriptcs.net/>

[2] FluentAutomation. <http://fluent.stirno.com/> ■

> about the author

Vagif Abilov



Vagif Abilov is working for a Norwegian company Miles. He has more than twenty years of programming experience that includes various programming languages, currently using mostly C#, F# and Gherkin.

Vagif writes articles and speaks at user group sessions and conferences. He is a contributor to several open source projects, such as SpecFlow and Simple.Data, and a maintainer of Simple.Data.OData, Simple.OData.Client and MongoDBData.

Twitter: [@ooobject](https://twitter.com/ooobject)



twitter
Follow me @vanTesting

**Prof. van Testing
recommends**



Díaz Hilterscheid

IREB® Certified Professional for Requirements Engineering Foundation Level

Course Details

The three-day training course "IREB® Certified Professional for Requirements Engineering" is given by Díaz & Hilterscheid in German and English and can be completed with an independent certification exam.

Incomplete or inconsistent requirements engineering leaves scope for interpretation in software development and makes verification and validation difficult. Corrections during the project can have serious consequences on time and cost planning.

The training introduces you to the correct and complete capture, documentation, checking and administration

of requirements. You will become acquainted with procedures, techniques and tools and will shown the fundamentals of communication theory. The training is based on the independent syllabus of the International Requirements Engineering Board (IREB®).

Requirements

Information regarding the required knowledge can be found in the IREB® syllabus, which can be downloaded from the IREB® website: www.certified-re.com.



In-house Training

All of our courses are available as private/in-house training. Please contact us for details.

Target Audience

The training is intended for requirements managers, project managers, quality managers, software testers and software developers who preferably have experience in IT projects and some initial experience with the handling of requirements.

See all dates at
training.diazhilterscheid.com.

**For more information, please visit our website
or contact us:**



Díaz & Hilterscheid Unternehmensberatung GmbH
Kurfürstendamm 179
10707 Berlin
Germany

Phone: +49 (0)30 74 76 28-0
Fax: +49 (0)30 74 76 28-99

E-mail: training@diazhilterscheid.com
Website: training.diazhilterscheid.com



Ensuring Sustainable Quality of the Product in an Agile Environment with Automated Test Generation

by Anahit Asatryan

Abstract

Nowadays software products/applications indisputably form an integral part of our business, day-to-day activities and social life. With the continuously growing importance of contemporary applications, it has become business critical to establish an effective and efficient process for ensuring the sustainable quality of the product.

With the Agile software development methodologies and continuous delivery practices widely used these days, software development cycles have been reduced dramatically to maximally shorten time-to-market and achieve customer satisfaction. This makes it even harder to ensure the sustainable quality of applications which are becoming ever more complex, and this requires the review and adoption of existing functional and regression techniques.

Motivated by this challenge, this article proposes test scenario and test script generation algorithms that are based on the imitational model of application from the high level description of its functionalities. The suggested methodology can be used for black box functional and regression testing of the product in Agile development environment where neither application model nor functional specification documents are available.

Practicing Agile development

These days, numerous companies apply incremental software development methods such as Agile software development [4]. The Agile Manifesto has a few principles that can be summarized as: “Individuals and interactions over processes and tools”, “Working software over comprehensive documentation”, “Customer collaboration over contract negotiation” and “Responding to change over following a plan”.

The Agile development process assumes short development life cycles and reduced time-to-market by achieving the main goal: customer satisfaction.

Continuous integration and continuous delivery

Continuous integration (CI) is the practice, in software engineering, of merging all developer working copies with a shared mainline several times a day [5]. Having continuous integration allows you to have all the newly implemented features integrated into the product. However, as a result, you may lose the ability to have a shippable product at every point in time. In the same vein, the practice of continuous delivery (CD) further extends

CI by making sure the software checked in on the mainline is always in a state that can be deployed to users and makes the actual deployment process very rapid.

Continuous Delivery (CD) is a pattern language used in software development to automate and improve the process of software delivery. Techniques such as automated testing, continuous integration, and continuous deployment allow software to be developed to a high standard, easily packaged, and deployed to test environments, resulting in the ability to rapidly, reliably, and repeatedly push out enhancements and bug fixes to customers [6]. However, with CD you will not always have all the new features integrated (only the ones that are ready).

As it is hard to achieve both at the same time, your company may choose either continuous integration or continuous delivery, depending on your business strategy.

Improving the testing process

Whether your company has decided to go with continuous delivery or not, it is critical to establish the quality assurance process so that the quality is built into the product.

Once you have released the product, the sustainability of the product quality starts to play a critical role in customer satisfaction. When it comes to the testing phase, you should ensure that the existing product functionality does not suffer as a result of integrating new features.

In the testing phase of product development, quality can be controlled by:

1. Maximally covering the functionality of each new feature with automated tests
2. Using the continuous integration tool to run the whole regression suite after each new feature integration

Usually automated tests are written based on the test scenarios/paths prepared beforehand. Test scenario preparation and automated test development is done manually in many companies. This makes the process time consuming and human-dependent, so leaves gaps in functional path coverage.

To make the testing process more efficient by reducing time required for scenario/path preparation and test automation and eliminating the human factor, this article proposes test scenario and test script generation algorithms based on the imitational model of the application from the high level description of its functionalities.

Algorithm for test scenario generation

Suggested test-generation methodology can be used for black box functional and regression testing of the contemporary applications in an Agile development environment where neither the application model nor the functional specification documents are available. The proposed test scenario and test script generation algorithms are based on the high level description of the application functionalities [1][2][3].

The test-scenario-generation algorithm consists of the following steps:

1. Splitting the full set of functionalities into equivalence classes.
2. Sorting the functionalities by relations count.
3. Selecting not equivalent functionalities.
4. Generating the adjacency matrix based on the set of non-equivalent functionalities.
5. Generating test scenarios in three different ways based on the adjacency matrix:
 - a. Test scenarios generated from a set of non-equivalent functionalities.
 - b. Test scenarios generated from a full set of functionalities (including equivalent ones).
 - c. Test scenarios generated for the given functionality.

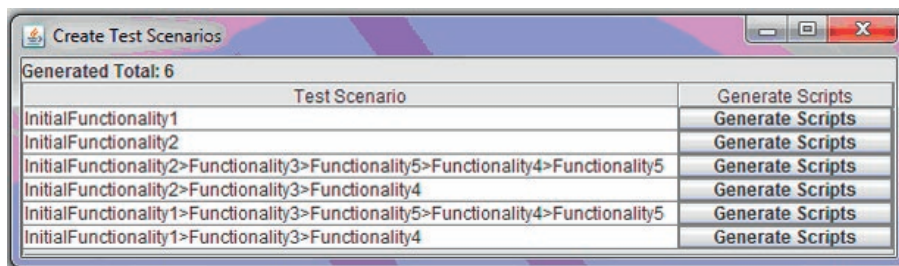
When test scenarios are generated, test scripts can be generated for the test scenarios.

A test tool implementing the suggested algorithms is developed. This supports easy extension and integration with contemporary testing frameworks such as Selenium Web Driver.

An application window displaying the generated list of test scenarios is shown in Figure 1.

Conclusion

The proposed algorithms make the black box functional and regression testing of the applications under test more efficient by reducing the time required for test scenario preparation and test automation, and by eliminating the human factor. The proposed test generation methodology can be used for testing different types of applications and can be applied in an Agile development environment where the application model and functional specification documents are not available.



| Test Scenario | Generate Scripts |
|---|------------------|
| InitialFunctionality1 | Generate Scripts |
| InitialFunctionality2 | Generate Scripts |
| InitialFunctionality2>Functionality3>Functionality5>Functionality4>Functionality5 | Generate Scripts |
| InitialFunctionality2>Functionality3>Functionality4 | Generate Scripts |
| InitialFunctionality1>Functionality3>Functionality5>Functionality4>Functionality5 | Generate Scripts |
| InitialFunctionality1>Functionality3>Functionality4 | Generate Scripts |

Figure 1. Test scenarios generated

Further reading

- [1] Sayadyan G.A., Arakelyan A.A., Aleksanyan N.A. "On the automation of synthesis of functional tests for blocks of micro-computers", *Automation and Computer Engineering*, 1981, N1, pp. 29–33.
- [2] Arakelyan A.A., Sayadyan G.A., Ohanjanyan S.R. "Algorithms for automatic synthesis of functional control LSI Firmware", *Automation and Computer Engineering*, 1983, N1, pp.55–59.
- [3] Boshyan K.G., *Development of methods for the automated synthesis of functional control tests of microprocessors*, Yerevan, 1992.
- [4] http://en.wikipedia.org/wiki/Agile_software_development
- [5] http://en.wikipedia.org/wiki/Continuous_integration
- [6] http://en.wikipedia.org/wiki/Continuous_delivery ■

> about the author

Anahit Asatryan



Anahit is a Senior QA Automation Engineer at AtTask, the SaaS leader in project management solutions. Anahit has more than nine years' experience in quality assurance and test automation in different areas. She graduated from the State Engineering University of Armenia and has the degree of Research Engineer. She is currently working on her PhD thesis, which is devoted to the "Development of an imitational model for testing Web-based applications". Her hobbies are photography, dancing, and hiking.

Twitter: @Anahit_Asatryan

Refactoring Your Best Asset – Your People – Through Mentoring

by Peter Saddington

Over half of all Nobel Prize winners were once apprenticed to other Nobel laureates.

As a volunteer counselor who is passionate about growing other great talent in our Agile community, I often take on opportunities to mentor others. I believe in the power of mentoring others. I believe in the power of helping people grow and begin to taste their potential. It is so very exciting for me to help others. Isn't this what servant leadership is all about?

Let's talk about mentoring for a bit ...

What exactly is mentoring?

- To help mature someone in a practice or discipline
- To show them how you walked the path and to lead them through their own path
- To teach them to *mentor someone else*, what you are doing to them.
- Make a distinction between mentoring and teaching. If you are teaching, you are telling. If you are mentoring, you are walking with them through it (high-touch).

What mentoring is *not* primarily concerned with:

- A methodology and exact praxis of how to do something. It is not prescriptive. The person you are mentoring is *not* your disciple.
- Mentoring is not a two-way street like friendship. Mentoring is not accountability, but it is focused, unlike friendships.
- Personal agendas.
- You. The focus is on them, not you. We are to pour our life into someone else.

6 Tips for Mentors

1. A mentor takes time to know people and reveal to them new possibilities and realities.

- Mentors are good listeners and they have the ability and willingness to step over familiar ground to get to know people and bring them into the circle.
- If you are mentoring someone for a particular role, help an individual by inviting them into communities of that

practice. Always try to bring people not in the inner circle into the circle.

2. A mentor gets excited when good things happen to others.

- One of the wonderful, nourishing characteristics of a mentor is the ability to get excited about the good things that happen to other people.
- A mentor is someone who constantly celebrates the wins, while also giving firm guidance where necessary about areas of potential trouble. Mentors need to be situationally aware and experienced, so they can point to examples where trouble can (potentially) happen...but provide enough freedom for the individual to experiment and even fail.

3. A mentor takes the initiative to help others.

- Take the first step. Have margin in your life to reach out to those that you believe could use your help. This is not about ego, it is about a willingness to help.
- I have never been turned down when I have spoken to someone and let them know that I would "love to intentionally spend more time with you to help you grow your craft". Offer your services. You will be even more rewarded than the person you mentor!

4. A mentor raises up leaders.

- You raise up others so they can pass you in leadership. Of all the things we will talk about on mentoring, this may be the best part. You see, the reality is that the one you mentor can (and will) be more successful than you are now. This is a *great* thing. You have created a legacy.
- We need more leaders. Do your part by helping leaders grow. This is how you 'scale yourself'. Great mentors develop leaders who are better than themselves. Wouldn't that be your definition of success as a mentor: *to pour your life into someone until they pass you?*
- A mentor's goal. We have all heard the statement: "There's no success without successors". But how about this? "Real success is having a successor who does a better job than we do." This is the highlight. This is what mentors live for. They live to be bypassed by somebody they have taught.

5. A mentor is willing to take a risk with a potential leader.

- Take risks with the one you mentor. Put them in positions where they can grow and even put them in positions where your reputation may be at risk if they fail. This imbued trust that you give the one you mentor is a huge step. But it will be the biggest win for all when he does well!
- You want to be able to say: *"You know what (mentoree)? I've mentored you – you're bigger than I am and it's time for me to find someone else to mentor. I'm going to take a risk for (a new person) as I did for you ... and I'd like your help. Want to help me grow another person?"*

6. A mentor is not position-conscious.

- Another Agile coach once said to me that *"servant leadership is influencing upwards and influencing outwards since no one is below you"*. He was right. You will always (in a sense) be a peer to others ... and there will always be people who position themselves higher than you. That is ok. You are growing others to be higher than you, with the hopes they will not have an egotistical attitude about it (agilescout.com/agile-coaching-is-about-being-available-to-help-others). That is a risk indeed!
- Your fanfare and rewards will be seen in others. You will have to be ok with that. Period.

6 Areas of a Mentor Relationship

Some practical guidelines for those who are interested in mentoring others. I always want to go over principles first, and then move on to practical guidelines, as it allows us to know why we are doing what we are doing.

- Authority/desire
- Intensity
- Duration
- Format/structure
- Intentionality
- Goals

My experience

1. Authority/desire – These are focus areas – what are you focusing on?
2. Intensity – Low key. How often are we going to engage?
3. Duration – 1 year. Length of mentorship program.
4. Format/structure – Book, talk, workshops, or problem solving?
5. Intentionality – Observe them. Yes, watch them in action if possible.
6. Goals – *They can mentor others.*

What Will You Live For: Titles or Testimonies?

In Tony Campolo's book *Who Switched the Price Tags?*, he talks about a Baptist preacher who was speaking to a group of collegians in his congregation. The following are a couple of paragraphs I want to read to you:

"Children you're going to die. One of these days they're going to take you out to the cemetery, drop you into a hole, throw some dirt on your face, and go back to church and eat potato salad (it'll be kimchi and duk in our case). When you were born you alone were crying and everyone else was happy. The important question I want to ask is this: when you die are you alone going to be happy, leaving everyone else crying? The answer depends on whether you live to get titles or whether you live to get testimonies.

When they lay you in the grave are people going to stand around reciting the fancy titles you earned, or are they going to stand around giving testimonies of the good things you did for them? Will you leave behind just a newspaper column telling people how important you were, or will you leave crying people who give testimony of how they've lost the best friend they ever had? There's nothing wrong with titles. Titles are a good thing to have. But if it ever comes down to a choice between a title or a testimony, go for the testimony."

He is talking about leaving a legacy.

Start leaving yours now. ■

> about the author

Peter Saddington



Peter Saddington owns a successful research and analytics consultancy and has been integral in multi-million dollar Agile Transformation projects with some of the biggest Fortune 500 companies, including Cisco, T-Mobile, Capital One, Blue Cross Blue Shield, Aetna, Primedia, and Cbeyond. He is a sought-after speaker at many industry events and is a Certified Scrum Trainer (CST). He has also received three master's degrees, one of which is in counseling, and provides life-coaching services in addition to his consultancy.

Twitter: [@agilescout](https://twitter.com/agilescout)

WE WANT YOU!



Testers, Developers, Consultants or Senior Consultants

for Consultancy Services (m/f) for agile software development
and software testing – Germany and Europe

Grow with us – we are recruiting!

We are looking for dedicated and qualified colleagues to strengthen our Consultancy Services team at the next possible date.

We are a market-oriented and innovative company based in Berlin with 40 employees. Apart from our high-value consultancy services in the areas of Financial Services, IT Management & Quality Services and Training Services, our customers

also appreciate the international conferences and publications on the subject of IT quality, which are organized by our Events & Media division.

Profit from our experience. We offer:

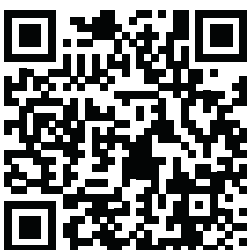
- interesting and challenging IT projects in the area of IT processes and projects, staff qualification and coaching, software testing and test management, as well as architecture and security

- a direct working relationship with the division leader and the team leaders of IT Management & Quality Services
- career development within a flexible company

You're looking for something special? So are we!

Let yourself be infected by the friendly working atmosphere in a strong and motivated team.

our job offers



For more job details, please visit our website or contact us:

Díaz & Hilterscheid Unternehmensberatung GmbH
Kurfürstendamm 179
10707 Berlin
Germany

Phone: +49 (0)30 74 76 28-0
Fax: +49 (0)30 74 76 28-99

E-mail: hr@diazhilterscheid.com
Website: jobs.diazhilterscheid.com



Díaz Hilterscheid

Writing Testable Use Cases Using Enterprise Architect

by Sander Hoogendoorn

One of the most frequently used modeling techniques for describing, designing, and testing requirements is the use case diagram. A use case diagram consists mainly of actors and use cases, either traditional or smart use cases. Other than the nowadays more popular user stories, use cases present teams with the opportunity to structure their requirements.

This little bit of additional structure then allows testers in teams to also test the requirements using more standardized techniques than apply to user stories. This article shows how to model and design your use cases so they become testable.

But before looking into the use case diagram, it is worthwhile delving into the format for describing use cases.

A use case template

The specification of the individual use cases in the model follows a template that at least should contain:

- **Name.** Every use case should have a clear descriptive name that will, in most use cases, suffice to make clear what this use case is all about. Please name use cases in the active sense, most often using a combination of an active verb and a noun.
- **Goal.** What is it that the use case should do? This can also be used for a brief description of the use case and is often a first draft of the use case.
- **Pre-conditions.** The set of conditions that must be met before the use case can be executed.
- **Post-conditions.** The set of conditions that must be met by the use case on completion. In my opinion, these can be both positive and negative conditions.
- **Basic flow.** A sequence of steps that describes the interaction between the actors and the system that leads to the desired result. The actors can be both the executing actor (often a person or role) as assisting actors (in many cases other systems, or services).
- **Alternative flows.** Deviations from the basic path may lead to one or more alternative flows. Each alternative flow again describes a sequence of steps that starts at a certain step in either the basic flow or another alternative flow, and returns to possibly the same step, or to another step (in the same scenario), or even possibly ends the use case.

Constraints

In addition, we often model other types of constraints (next to pre- and post-conditions) with a use case:

- **Input.** Parameters that are passed into the use case when the use case is started. Note: these are *not* pre-conditions.
- **Validation.** With certain steps validation can occur. In most cases validation concerns the domain objects or view models that are handled by the use case. These should be modeled with the object they concern. However, sometimes additional validation is typical for the use case at hand. These validations are modeled as constraints on the use case.

Describing use cases in Enterprise Architect

Enterprise Architect is a commonly used modeling tool, targeted at modeling UML and BPMN diagrams. In Enterprise Architect every use case comes with a property window which is opened, for instance, on double-clicking a use case in the diagram. Unfortunately this property window is implemented as a modal window, which means it is not possible to navigate away from the window without pressing OK or Cancel. Thus it is hard to check on other diagrams or elements in the model while specifying a use case.

On the main page of the property window the use case is named. We usually use the Notes property to specify the goal of the use case. Keep this description short, as it is not intended to capture one or more of the flows in the use case. In the example below, the Notes property is clearly far too long.

UseCase : View Relationships

Name: View Relationships

Stereotype: view Status: Approved Complexity: Easy

Keywords: Author: Sander Hoogendoorn Version: 1.0 Phase: 15.0

Database:

Notes:

Contact -> Select, select a contact and open it. At the bottom of the screen you see a tab Relationships.
 You can see with which other contact the current contact has (had) a relationship and what kind of relationship this is, such as an employer / employee relationship between an organisation and an individual or a parent / child relationship between two individuals.
 Note that you will see the relationship also on the Relationship tab of the related contact.
 Clicking on the other contact opens the Manage Contact form, so this contact can be consulted or edited. Clicking on the role of the contact in a relationship opens the Manage Relationship form.
 Clicking New Relationship leads to the Manage Relationship form.

OK Cancel Apply Help

UseCase : Manage Relationship

Constraint: Other Contact is not empty Type: Validation Status: Approved

The other contact for the relationship should be valid.

| Constraint | Type | Status |
|--|----------------|----------|
| Current Contact is passed in | Input | Approved |
| Current Relationship is passed in | Input | Approved |
| OK - Current Relationship is saved | Post-condition | Approved |
| OK - Current Relationship is removed | Post-condition | Approved |
| Cancel - No changes of Current Relationship are made | Post-condition | Approved |
| Other Contact is not empty | Validation | Approved |

New Save Delete

OK Cancel Apply Help

Describing constraints in Enterprise Architect

We consider a number of types of constraints with a use case, as described above. Enterprise Architect allows you to define your own constraint types, in addition to the pre-defined types, so we have added *Input* and *Validation*.

General Types

Constraint: Validation Description: Validation on use-case

Note:

New Save Delete

| Name | Description |
|----------------|---|
| Input | Parameters that are passed-in to the use case |
| Invariant | A state the object must always be in |
| Post-condition | An ending state that must be met |
| Pre-condition | A starting state that must be met |
| Process | A process that must occur |
| Validation | Validation on use-case |

Close Help

The constraints for a specific use case can again be described in Enterprise Architect in the property window. We usually specify both a name and a description for each of these constraints.

In the example here, the use case has defined a pre-condition, a post-condition, requires both a valid Contact and a Relationship to be passed in (constraint type *Input*), and describes a number of validations (constraint type *Validation*).

Please note that names for validation constraints should be clear and descriptive, as they will also appear in code on the implemented use case, resulting in a method of the use case class, as suggested in the C# code example below.

```

1 [BusinessRule]
2 public ValidationResult OtherContactIsNotEmpty() {
3     return !Relationship.To.IsEmpty
4         ? ValidationResult.CreateError(this.Prop(t =>
5             t.Relationship.To), "Other contact in relationship
6             may not be empty")
7         : ValidationResult.Success;
8 }

```

Different types of validations

When describing validations, it is vital to understand which validations are handled by the use case and which validation are handled by the domain object or view model that is handled by the use case.

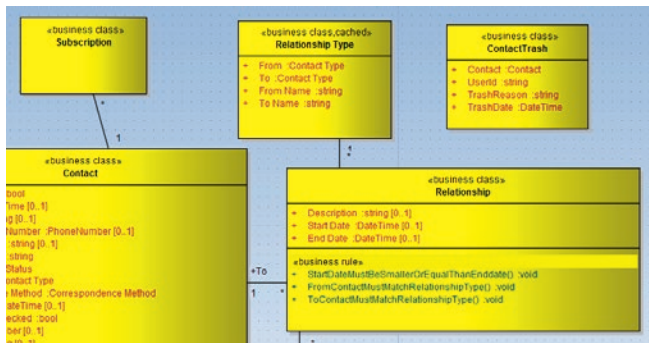
Validations on domain objects

During the execution of a particular use case, the state of a domain object (or view model) might be altered, e.g. due to user input, but also due to specific steps in the use case. When this happens, the domain object can be considered as *dirty*. This is a state where it is uncertain whether the domain object is actually valid, or that it has become invalid. Validations may now be performed on the domain object, for e.g. before persisting it.

Such validations may include required fields, validation properties that are value typed (think of email, zip codes, social service numbers), but can also be business rules (start date for a contract must be before the end date of a contract). These validations are always true when the domain object is in a valid state.

Modeling validations on domain objects

Such validations are best modeled and described with the specific domain object.



In the example above we have modeled different types of validations:

- **Required.** The properties *Description*, *StartDate* and *EndDate* of the domain object *Relationship* are not mandatory for the domain object to be valid. The multiplicity of these properties is set to *[0..1]* – which implies that zero or one instances of this property are present.
- **Value object.** Although not visible in the example, properties may have types that are defined as value objects, such as *Email* or *Isbn*. This means that the validation rules for this particular value object will apply to this property.
- **Enumeration.** In the class *RelationshipType*, the property *To* is modeled as *ContactType*, which is an enumeration of possible values. This enumeration is modeled elsewhere in the domain model.
- **Association.** The class *Relationship* described three associations with other classes. The properties *From* and *To* are modeled as associations with *Contact* and, given the multiplicity, are mandatory for an instance of *Relationship* to be valid.
- **Rule.** Additionally, we have modeled some business rules as methods on the domain object *Relationship*, such as *ToContactMustMatchRelationshipType()*. We have used the stereotype *business rule* to make sure developers will implement these accordingly, such as in the code example below.

```

1 [BusinessRule]
2 public ValidationResult ToContactMustMatchRelationshipType() {
3     return !To.IsEmpty && RelationshipType.To != To.ContactType
4         ? ValidationResult.CreateError(this.Prop(r => r.To), "Contact type '{1}' for {0} does not match required contact type '{2}'.", To, To.ContactType, RelationshipType.To)
5         : ValidationResult.Success;
6 }

```

Validations on use cases

With some use cases validations may occur during the execution of the use case steps. Such validations will validate the correct behavior of the use case and are not specific to the domain objects that are handled by the use case.

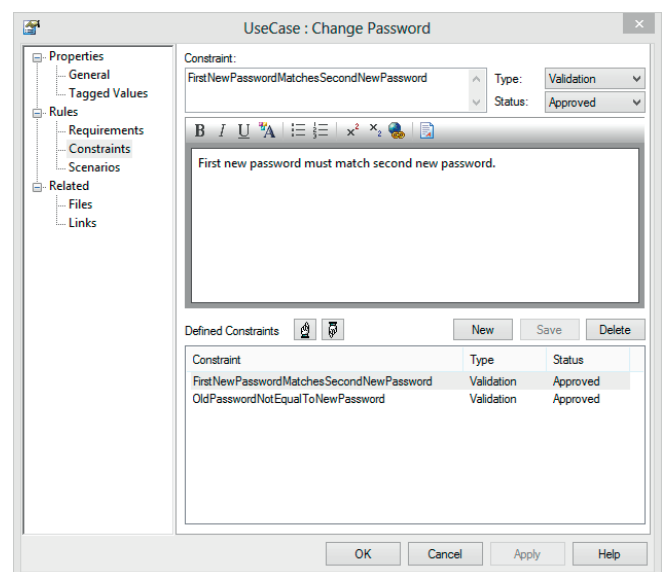
A good example would be a use case *Change Password*, where the user has to enter his old password for his *Account*, and a

new password twice. Typically, *Password* is a property of the domain object *Account*. But entering a new password twice is only temporary and checking whether the new password entered first matches the new password entered second does not validate the state of the attached *Account* object.

Therefore both new passwords are not modeled and implemented as properties on *Account*, but rather it is the use case's responsibility to validate equality on the new passwords. It is only after these appear to match, and the old password entered matches the *Password* property of the *Account* object, that this property gets set, and the *Account* object is validated and saved.

Modeling validations on use cases

Such validations can be modeled in Enterprise Architect in the property window of the use case.



Use the (custom) constraint type *Validation*. Give the validation a name that will match the name of the method on the use case in code to guarantee traceability. Also write an indisputable description for the validation.

Use case scenarios

There are different ways of modeling and describing what action a use case is comprised of. In general we consider use case scenarios. Each scenario describes a way the use case can be executed. There are three types of scenarios:

- **Happy day scenario.** Executing this scenario leads to the desired result, the goal that the actor wants to achieve, via the optimal path.
- **Fail scenario.** Running this scenario results in not reaching the desired result, but an alternate result where the actor does not achieve his goal.
- **Recovery scenario.** During this scenario some anomaly is encountered that leads away from the optimal path. But, by using some additional action, in the end the positive result is reached and the actor's goal is achieved.

Each scenario consists of a number of consecutive steps. As you might expect, every result that can be reached is in fact

a post-condition of the use case. Thus, in my opinion, use cases can (and will most often) have multiple post-conditions.

Scenarios are very useful when it comes to testing the functionality of a use case, as testers are used to describing test cases that match one of the scenarios.

Use case scenarios, basic flows, and alternative flows

In Enterprise Architect and most other modeling tools, scenarios are not literally described as such, but rather use a different terminology based on flows. Here a *basic flow* represents the happy day scenario. Next to the basic flow, modeling tools represent *alternative flows*. Unfortunately these do not always map to either fail or recovery scenarios.

An alternative flow is described as a replacement of one or more of the steps of another flow – either the basic flow or another alternative flow. This differs from scenarios. An alternative flow does not describe a particular scenario, but only a part of it.

Combining the basic flow with one or more alternative flows again leads to a scenario – always a fail or recovery scenario – if it ends at one of the post-conditions of the use case. To facilitate the use of basic and alternative flows, the steps they describe (in a similar way to scenarios) are usually numbered.

Alternative flows and exception flows

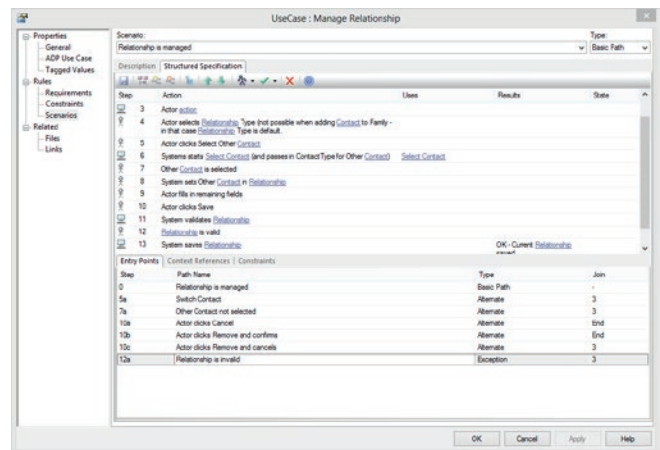
Enterprise Architect distinguishes alternative flows and exception flows. Although the concept and differences are not explained, in general an exception flow is modeled when something goes wrong during the execution of a use case, for instance if some validation fails. An alternative flow is modeled when the execution is correct, but takes a slightly different path from the basic flow. For instance, the user presses a button that executes some additional action.

In general, when the execution of an alternative or execution flow is finished, the flow returns to the calling flow. In theory this can be any other flow. However, Enterprise Architect (using structured specifications) only allow for a single level of alternate and exception flows – which is good because it disallows complex scenarios. After a flow is finished, the flow *can* return to the calling flow, but it might also end the use case, resulting in reaching one of the post-conditions.

Structured specifications

Use cases are usually specified in Word or other text editors, because there are few tools that allow for a more structured way of defining numbered flows. Although writing plain text allows for maximum flexibility, trouble arises as soon as the numbering of the steps needs to change, due to newly inserted steps or steps that are deleted. Here, references to these steps from other flows need to be updated too.

From version 8.0, Enterprise Architect offers the possibility of writing flows in a more structured way using the structured specifications tab in the use case properties window.



Although not perfect, this structured specification offers a number of other interesting additional features:

- **Generate activity diagram.** The tool can generate an activity diagram that displays all scenarios for the use case based on the structured specifications. This is quite similar to how it is described in my UML book.
- **Generate test documentation.** Using a document template, the tool allows for quite extensive generation of the test scenarios for the use case.

So, structured specifications are an interesting alternative to writing use case scenarios. Here are some tips.

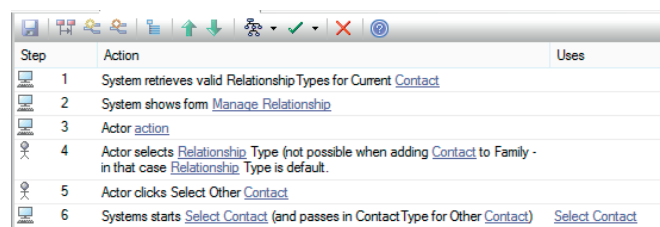
Tip. First write a basic flow

Start with describing the steps for the basic flow. Do not add any of the alternate or exception flows yet. A basic flow contains all steps that are required to reach the goal for the use case, or the success post-condition.

A flow does *not* include if-then constructs (these will be alternate or exception flows). However loop constructs can appear.

Tip. Actor or System

Start each step with the actor executing the step. If there is only one actor, I would prefer to use the term *Actor* to avoid issues when the name of an actor changes in the model. Steps that are executed by the system always start with the term *System*.



Note that Enterprise Architect specifies an icon in front of the steps indicating actor or system. This icon can be flipped from the context menu (although it should recognize it when you type *Actor* or *System*).

Tip. Note called use cases

When modeling smart use cases (<http://www.accelerateddeliveryplatform.com/SmartUseCase.aspx>), in some steps of your

flow other use cases might get called. Describe calling the other use case as a single step, executed by the system.

| | | |
|---|--|--------------------------------|
| 5 | Actor clicks Select Other Contact | |
| 6 | System invokes Select Contact (and passes in Contact Type for Other Contact) | Select Contact |
| 7 | Other Contact is selected | |
| 8 | System sets Other Contact in Relationship | |

As shown in step 6, we use the syntax *System invokes Use Case Name*, and sometimes add the parameters we pass to the called use case. In the *Uses* column of the structured specification, we note the name of the use case, as this column immediately shows when looking at the specification.

Tip. Check post-conditions of called use cases

Right after a use case is called, such as *Select Contact* in step 6 in the example above, you should check for the post-conditions of this use case. In most cases, the positive post-condition for the called use case is the desired result. Note this post-condition in the next step.

In the example above, step 7 mentions that Other Contact is selected. This literally is one of the post-conditions of use case *Select Contact*. Although this step is not necessarily an action, such as regular steps are, it is of vital importance that post-conditions are checked. Moreover, adding the positive post-conditions as a step in the (basic) flow allows for validating the other post-conditions of the called use case in alternate or even exception flows that, in this case, branch off from step 7.

| | Relationship is managed | Basic flow | |
|-----|----------------------------------|------------|-----|
| 5a | Switch Contact | Alternate | 3 |
| 7a | Other Contact not selected | Alternate | 3 |
| 10a | Actor clicks Cancel | Alternate | End |
| 10b | Actor clicks Remove and confirms | Alternate | End |
| 10c | Actor clicks Remove and cancels | Alternate | 3 |

Here the alternate flow *Other Contact not selected* describes what happens in that particular case. After finishing, it re-joins the basic flow in step 3. Do not forget to add these alternate flows.

Tip. Synchronize user actions

Many use cases have a strong component of interaction with the user. In this case, the user can often start many actions independent of their order. Think of starting another use case by pressing a button, starting some action by pressing a button, filling in fields on a form, submitting it, or pressing the cancel button.

The trouble with these user actions is that, in general, they do not follow a specific order. This makes describing user actions in a sequence of consecutive steps a bit awkward.

I therefore recommend the following. First, create a step in the basic flow that is simply called *User action*. This step is used to allow loops and trigger actual user actions in random order. All paths now return to this specific step.

| Step | Action | Uses |
|------|--|------|
| 1 | System retrieves valid RelationshipTypes for Current Contact | |
| 2 | System shows form Manage Relationship | |
| 3 | Actor action | |
| 4 | Actor selects Relationship Type (not possible when adding Contact to Family - in that case Relationship Type is default. | |
| 5 | Actor clicks Select Other Contact | |

In this example, step 3 represents this step and is called *Actor action*.

Next, add the user actions that are essential for reaching the positive post-condition in the basic flow. In this example, steps 5 to 7 represent such an essential user action.

And thirdly, add all other user actions as either alternative or exception flows. All of these branch off from the step right after the Actor action step. If these flows return to the basic flow, they always return to step 3.

Tip. Put post-conditions in results

The post-conditions of a use cases can be reached in several locations. Of course, the positive post-condition is reached at the end of the basic flow, but alternative and, certainly, exception flows can also result in ending the use case if one of the post-conditions is met.

It is a good habit to note these post-conditions as results in Enterprise Architect's structured specification, as shown in the alternative flow *Actor clicks Cancel* below.

| Step | Action | Uses | Results |
|------|-----------------------------------|------|---------------------------------------|
| 1 | System does not save Relationship | | Use Case is cancelled |
| 2 | End | | |
| 3 | new step... | | |

We usually precede the actual post-condition (No Relationship is saved) by either *OK* or *Cancel* to show whether or not a positive result has been achieved.

Tip. Skip details in fields in forms

When the user needs to fill in a number fields on a form, it is very tempting to add details on these fields to the structured specifications, such as:

- **Label.** Name of the label associated with the field.
- **Mandatory.** Can the field be left empty?
- **Display format.** Is the field displayed as a text box, a radio button list, a drop down list, a checkbox, a link?
- **Edit format.** Does the field have a specific edit format, such as with currencies or bank accounts?
- **Validations.** All kinds of field validation, such as amounts not allowed below zero, or dates not before today.

However, adding these details to the structured specifications clutters the flows. It is better to add these field details in another location, either in a user interface diagram, or to an additional document added to the use case. Enterprise Architect uses a linked document for this purpose. We use a formatted template to create these linked documents.

For the flows it is then sufficient to simply add a single step that defines that the user enters the fields in the form, as in step 9 in the example below.

| | | |
|----|---|--|
| 7 | Other Contact is selected | |
| 8 | System sets Other Contact in Relationship | |
| 9 | Actor fills in remaining fields | |
| 10 | Actor clicks Save | |
| 11 | System validates Relationship | |

Tip. Skip details on validation

A similar approach can be used to describe validation. As mentioned earlier, validation can be part of the use case or can be part of elements of your domain – your domain objects, value objects, enumerations, repositories, services. In the latter case, validation has already been modeled and described there. There is no need to repeat these validations in your basic and alternative flows. Validation that is specific to this use case should be specified with the use case, again not in the flows, but in the conditions.

Thus, in the flows of your use case it suffices to describe that validation takes place. But, due to the fact that validation can have different outcomes, add an additional step that handles the validation results. Always put the positive result directly below the validation step, such as in step 12 below.

| | |
|----|-------------------------------|
| 10 | Actor clicks Save |
| 11 | System validates Relationship |
| 12 | Relationship is valid |
| 13 | System saves Relationship |

Model any other outcome either as an alternative or exception flow that branches off from the positive outcome. In this case it branches off from step 12, such in alternative flow 12a below.

| | | | |
|-----|----------------------------------|-----------|-----|
| 10b | Actor clicks Remove and confirms | Alternate | End |
| 10c | Actor clicks Remove and cancels | Alternate | 3 |
| 12a | Relationship is invalid | Exception | 3 |

Tip. Check results for save and remove actions

Although with the current state of technology it appears that implementing save or remove actions on a domain object is trivial, in fact both actions can end in an undesirable result. For instance, a remove from the database might fail due to the fact that other records depend on the record you are deleting. A save might fail due to concurrency, or due to rules in the database failing.

So, again, in a similar way to checking the result of validations, it is good practice to also monitor the result of a save or remove actions with an additional step.

Concluding

Working with use cases, their basic flows, and alternative flows allows teams to more easily design and test the requirements for a system. The modeling tool Enterprise Architect facilitates modeling and designing flows and validation. Testers can use these scenarios, flows, and validations to easily create test scenarios and test cases.

We have successfully applied this approach with small fine-grained use cases to many, mainly agile projects, where a joint effort between analyst, developer, and tester thus creates well developed and tested requirements. ■

> about the author

Sander Hoogendoorn



In his roles of principal technology officer and global agile thoughtleader at Capgemini, Sander is involved in the innovation of software development both at Capgemini and its many international clients. Sander has coached many organizations and teams, has written books on UML and agile, and published over 200 articles in international magazines. He is an appreciated and inspiring speaker at many international conferences, and he presents seminars and training courses on a variety of topics such as agile, Scrum, Kanban, software estimation, software architecture, design patterns, UML, .NET, writing code, and testing. Sander is also a member several editorial and advisory boards, and he is the chief architect of Capgemini's agile Accelerated Delivery Platform (ADP). More at www.sanderhoogendoorn.com, www.smartusecase.com, and www.speedbird9.com.

Twitter: [@aahhoogendoorn](https://twitter.com/aahhoogendoorn)

Masthead

EDITOR

Díaz & Hilterscheid Unternehmensberatung GmbH
Kurfürstendamm 179
10707 Berlin
Germany

Phone: +49 (0)30 74 76 28-0
Fax: +49 (0)30 74 76 28-99

E-mail: info@diazhilterscheid.com
Website: www.diazhilterscheid.com

Díaz & Hilterscheid is a member of "Verband der
Zeitschriftenverleger Berlin-Brandenburg e. V."

EDITORIAL

José Díaz

ARTICLES & AUTHORS

editorial@agilerecord.com

In all of our publications at Díaz & Hilterscheid Unternehmensberatung GmbH, we make every effort to respect all copyrights of the chosen graphic and text materials. In the case that we do not have our own suitable graphic or text, we utilize those from public domains.

All brands and trademarks mentioned, where applicable, registered by third-parties are subject without restriction to the provisions of ruling labelling legislation and the rights of ownership of the registered owners. The mere mention of a trademark in no way allows the conclusion to be drawn that it is not protected by the rights of third parties.

ISSN 2191-1320



Díaz Hilterscheid

LAYOUT & DESIGN

Díaz & Hilterscheid
Lucas Jahn
Konstanze Ackermann

WEBSITE

www.agilerecord.com

ADVERTISEMENTS

sales@agilerecord.com

PRICE

online version: free of charge

The copyright for published material created by Díaz & Hilterscheid Unternehmensberatung GmbH remains the author's property. No material in this publication may be reproduced in any way or form without permission from Díaz & Hilterscheid Unternehmensberatung GmbH, including other electronic or printed media.

The opinions mentioned within the articles and contents herein do not necessarily express those of the publisher. Only the authors are responsible for the content of their articles.

Picture Credits

© DouDou – Fotolia.com C1

Index Of Advertisers

| | | | | | |
|---------------------------------------|----|--------------------------------|----|-----------------------------|----|
| Agile Dev Practices | 24 | Díaz & Hilterscheid GmbH | 28 | Mobile App Europe..... | C2 |
| Agile Record | 8 | Díaz & Hilterscheid GmbH | 34 | Mobile App Europe..... | 11 |
| CAE – Certified Agile Essentials..... | 5 | Díaz & Hilterscheid GmbH | 40 | Testing Experience..... | 15 |
| CaseMaker SAAS..... | 56 | Díaz & Hilterscheid GmbH | 64 | Testing Experience..... | 17 |
| CAT – Certified Agile Tester..... | 6 | Díaz & Hilterscheid GmbH | 69 | Testing Experience DE | 48 |