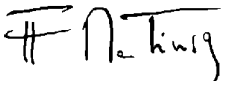
METHODS & TOOLS

Practical knowledge for the software developer, tester and project manager **ISSN 1661-402X**

Winter 2014 (Volume 22 - number 4) www.methodsandtools.com

The Virtue of Going Through Purgatory in Software Development

Having some decade of experience in software development behind me, I had the time to accumulate a lot of mistakes. One of the recurring patterns in these failures was the ambition to solve code issues too quickly. This was especially the case when the problem was related to code that I wrote, which made me feel responsible for the situation. When you detect a problem, sometimes you think that you just have to do a small change in the code to get things right. I trust myself and I want to solve the problem quickly, so I am not going to run a full regression test to check that my changes are not creating any negative side effects. This means that I can push the new code from the development to the production environment with limited testing. Don't we all want this problem to be solved quickly? It worked most of the time, but sometime the solution didn't work or just made things worse. This was mostly because I didn't took enough time to think about the actual issue or I had limited testing environment in development, mostly with a reduced set of data. This is how I start appreciating having a good integration testing environment between development and production. The purgatory is a catholic concept of a space between hell and heaven. People would be there in transit to make sure that they deserve to reach the paradise. I will say that we should make sure not to rush every time to production and allow our mind and our code to spend some time in purgatory. The first step is to make sure you understand the real cause of the issue and its consequences. If you are not able to reproduce a bug, how can you make sure you are correcting it? When facing a bug, we should "take a deep breath", taking a little bit of time to get away from this "I have to solve this quickly" feeling of urgency, but rather try to understand what the actual issue is. When we think that we have made the right corrections, the ideal way to confirm it is to have an integration testing environment that is as close as possible to the production environment. A place where you will find good test cases and maybe some automated testing help to speed the verification process. This could be difficult if you have a special production environment with links to external systems for instance or that contains sensitive data that business people are reluctant to let the IT people explore. Spending some time in the purgatory often make sure that your change is a good solution and your quick fixes don't makes the issue get worst and damage your software developer reputation. Just make sure that your code is qualified to reach the production stage. I wish all the Methods & Tools readers a Happy 2015 and would like to thank our longtime advertising partners for their continuous support.



Inside

Mobile Dev + Test Conference - Click on ad to reach advertiser web site

Analysis on Analysts in Agile

Leslie J. Morse, leslie [at] lesliej.net, @lesliejdotnet,
Davisbase Consulting, www.davisbase.com

Antagonizing Analysts

The story is common: the insecurities should be unwarranted, but the agony is reality.

- We're Agile now; we don't do requirements
- Agile teams don't need documentation
- There are only 2 roles on Agile teams: Customer and Developer
- Right-sized teams don't have space for Business Analysts

It is easy to make poor assumptions about how to involve business analysts in Agile practices. The Agile Manifesto [1] opens the door for frustration and confusion.

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Phrases like "processes and tools," "comprehensive documentation," and "customer collaboration" are often tightly coupled with the idea of business analysis. As a result, analysts derive much of their professional value from being involved in those activities.

A few of the Principles [2] behind the Agile Manifesto further the debate.

- *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*

  The reaction: Wait… changing requirements is a change request, and that's bad.

- *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*

  The reaction: It takes weeks or months to get on the stakeholder's calendar.

- *Business people and developers must work together daily throughout the project.*

  The reaction: They don't speak the same language. That is what a business analyst is for!

- *Working software is the primary measure of progress.*

  The reaction: The 274 pages requirements document is a whole heck of a lot of progress!

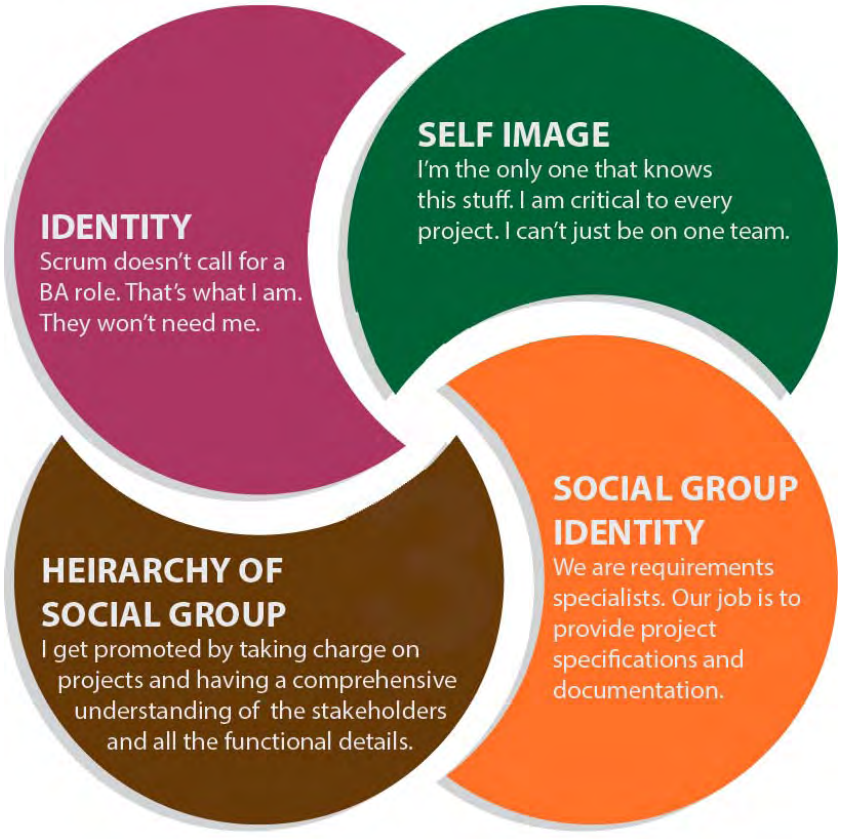
Agile reframes the way individuals add value, and the new construct for collaboration creates emotional resistance. Emotional resistance occurs when change disrupts how one derives his/her social capital. Performance systems that reward and incent attributes of a group's social capital exacerbate the magnitude of the resistance.

The Business Analyst culture values:

- Relationships (often the ownership of the relationship "with the business")
- Subject Matter Expertise (both business and functional)
- Facilitation skills
- Communication and Presentation skills

Figure 1 ties the culture of business analysts to the four aspects of social capital.

Figure 1, Emotional Resistance of a Business Analyst



This emotional resistance leaves analysts with 3 key concerns:

1. Job Security - Do I still have a job?
2. Documentation - That is how I produce my work; we need this. What will we do?
3. Time - How will it be possible to get all this analysis done so fast?

The good news is that these should not be worries. In fact, the role of a business analyst on an Agile team can be critical to the team's success. This article will explore four topics that shed light on how business analysts fit into Agile teams.

1. Analysis versus Analysts - The difference in the 'discipline' of Business Analysis versus the 'role' of Business Analysts.
2. Adopting Analysis - Key techniques and mental models for how the discipline is leveraged within Agile teams.

3. Analyst Actions - Specific recommendations for the way the role functions within an Agile team.
4. Appreciation & Acknowledgement - A tip of the hat to key thought leaders that are leading the way in the application of analysis in Agile.

Analysis versus Analysts

It is critical to distinguish between the *discipline* of Business Analysis versus the *role* of Business Analyst.

Business analysis is the set of tasks and techniques used to work as a liaison among stakeholders in order to understand the structure, policies and operations of an organization, and recommend solutions that enable the organization to achieve its goals. [3]

According to that definition, many software development practitioners could acknowledge involvement in business analysis. The International Institute of Business Analysis's definition of *business analyst* explicitly acknowledges that idea.

A business analyst is any person who performs business analysis activities, no matter what their job title or organizational role may be. Business analysis practitioners include not only people with the job title of business analyst, but may also include business systems analysts, systems analysts, requirements engineers, process analysts, product managers, product owners, enterprise analysts, business architects, management consultants, or any other person who performs the tasks described in the BABOK® Guide, including those who also perform related disciplines such as project management, software development, quality assurance, and interaction design. [3]

The rapid delivery of value requires Agile teams to perform analysis tasks and activities. In a perfect world, the individual playing the role of Product Owner will be deep in business analysis expertise. If that is not reality then the other team members must be competent in business analysis. Agile is not an excuse to skip over traditional software development activities. (See Figure 2, Traditional Activities of Software Development.) Instead, Agile approaches package those activities in a different way.

Figure 2, Traditional Activities of Software Development

SpiraTeam Complete Agile ALM Suite - Click on ad to reach advertiser web site

Adopting Analysis

Making a case for why Agile teams need to employ business analysis techniques is a topic in and of itself. Four approaches prove to be successful for teams that realize the need for business analysis.

1. Maintain a State of 'Ready'
2. Establish a Cadence for Refinement
3. Proactively Engage Stakeholders
4. Delineate between Work Products and Deliverables

Adopting any of these approaches will utilize a plethora of business analysis techniques.

Maintain a State of 'Ready'

Teams with unstable velocity and erratic Complete vs. Commit (CvC) rates may be suffering from a lack of runway or depth within the backlog. Maintaining a set of 'ready' stories in the backlog will decrease swirl within any given sprint. It should also help increase predictability and make prioritization easier. An extra sprint of 'ready' stories will allow the Product Owner to tweak order without disturbing flow.

What does 'ready' mean? It likely includes the following 4 elements:

- Shared Understanding - Everyone on the team can paraphrase the intent and approach for the backlog item. (Be sure to include both development and testing aspects of the work.)
- Know Enough - Team members have enough knowledge about the story to plan tasks. (i.e. If they took the item to Sprint Planning tomorrow, they already have a solid idea of what those tasks should be).
- Sized Appropriately - The team has triangulated the story's size with known factors, and verified the accuracy of the points assigned.
- Dependencies Fulfilled - Required incoming dependencies associated with the backlog item are complete.

Encourage teams to build their own Definition of Ready (DoR). Organizations with significant complexity and team interdependency may need to maintain 2 sprint's worth of stories in a 'ready' state.

Establish a Cadence for Refinement

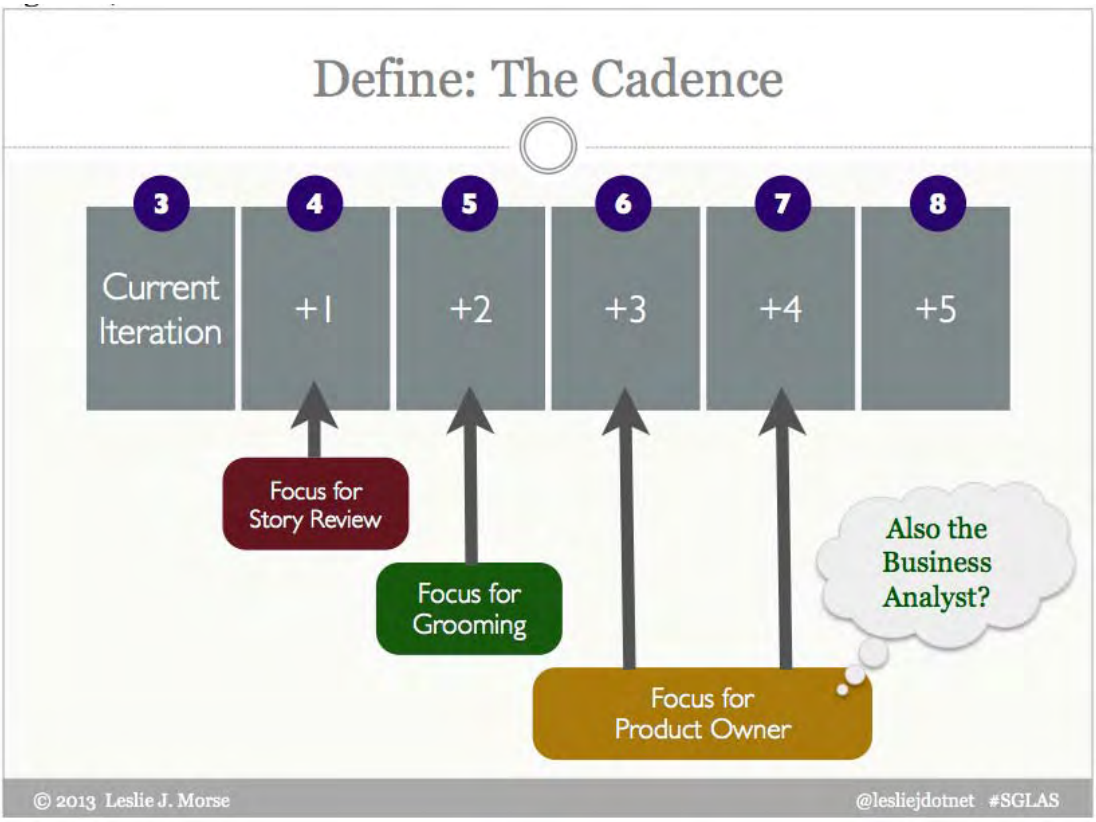
There is a touch of "robbing Peter to pay Paul" that goes into the balance of an Agile team's productivity. Every moment the team spends looking ahead, they have less time to devote to building the working software. However, without doing so the team will never maintain a state of 'ready.'

Building 2 sessions into a team's schedule can establish a cadence for refinement.

1) **Story Review Sessions.** These are brief 15-30 minutes sessions held during the first half of the iteration (perhaps on day 4 of 10). The intent of the session is to look at backlog items planned for the next iteration to ensure they are 'ready'. Holding this session early in the iteration allows the team time to resolve any open issues and proactively manage expectations.

Story Refinement Sessions. These are 2-4 hours sessions held once (maybe twice) during each sprint. If these sessions run shorter than 2 hours, the team barely get started before it is time to finish; longer than 4 hours, the team can run into analysis paralysis. During this time, the team collaborates on the detailed approach for delivery.

Figure 3 suggests a focus for these sessions: using it can help teams maintain a healthy runway within their product backlog.



Proactively Engage Stakeholders

Product Owners act as the single source of truth about *what* the team should deliver, which means they need to be available, knowledgeable, and empowered. It is a huge morale hit when teams demo working software and the feedback from stakeholders is essentially, “No way - that's not what we wanted.”

Such stakeholder feedback leaves the team in a situation where the Product Owner asked for the wrong thing. The inaccuracy could be within the content of the story or the acceptance criteria. You can prevent this situation by developing a strategy for engaging stakeholders to get their input in advance of the sprint.

There are three steps to building a proactive stakeholder engagement strategy:

- Step 1: Analyze Stakeholders - Who are they, and how close are they to the work?
- Step 2: Define the Cadence - How often should you meet?
- Step 3: Follow-Through - Keep the discipline for the sessions [4]

Grouping stakeholders into 2-3 categories (e.g. Advisors, Supporters, Sponsors [4]), allows the team to establish a patterns for meetings. For example, teams may define a weekly meeting cadence (e.g. Mondays from 10:00 - 11:30am) and rotate which group(s) attend the meeting.

A four-week rotation pattern could be:

- Week 1 - Advisors
- Week 2 - Advisors & Supporters
- Week 3 - Advisors
- Week 4 - Advisors, Supporters & Sponsors

Discussions should focus on work the team is about to do. The goal is final validation that the intent and acceptance criteria are truly accurate. It is important to include both business and IT stakeholders in these discussions.

Delineate between Work Products and Deliverables

Working software is the primary measure of progress. . That means that all artifacts produced in advance of the working software have the potential to add bloat to the process. Agile teams know that *simplicity - the art of maximizing the amount of work not done - is essential.* Finding ways to lean the process and documentation is key.

Think of “deliverables” as the final artifacts the team produces, the highest priority of them being the actual working software. Rarely are teams afforded the luxury of only needing to produce working software: training, production support, and governance groups typically need more than the software alone. Consider any supplemental documents a “deliverable” or long-term artifact.

Organizations try to use requirements documents, specifications, and technical design documents as long-lasting artifacts. The challenge is that those documents are always produced *before* writing the software. This results in situation where the software works in a different way; hence, why people often look at the code to know how something is supposed to work.

Think of traditional artifacts (requirements, specs, designs) as “work products”. These short-term artifacts only exist for the purpose of building the right deliverables. In an Agile world work products should capture the bare minimum the team needs and should exist as a result of collaboration (not a replacement for collaboration). User stories, scenarios, acceptance tests, diagrams, prototypes and models are the techniques often used.

In most cases, “deliverables” should be standard from team-to-team. (Hint: Capture the need for deliverables as part of a team’s Definition of Done.) “Work products”, on the other hand, should be flexible and as simple as the teams choose. The goal is to clearly delineate between work products and deliverables. Allowing flexibility around the detail captured in “work products” enables teams to be creative and apply a wide variety of analysis approaches. Delineating work products and deliverables also opens the door for redefining what deliverables should look like: if traditional work-products don't meet the needs of down-stream constituents, then what will?

Analyst Actions

There are countless ways for someone with a title like 'Business Analyst' to engage with Agile teams. Individuals with jobs focused on analysis are often rich with facilitation and negotiation skills, which are critical to navigating through naturally occurring team conflicts. They are also quite often adept at systems thinking and problem solving. What Agile team doesn't need that expertise?

Finally, BAs often have a wide network of relationships within the organization, and they may hold the key to resolving an impediment or getting clarity on a situation. It's important to play to these strengths when guiding Analysts to operate within an Agile environment.

Analysts may engage in the following ways:

- Act as a proxy Product Owner
- Engage stakeholders, aggregate opinions, present options
- Collaborate on authoring acceptance & functional tests
- Execute tests
- Facilitate refinement and story review sessions
- Assess backlog depth
- Conduct impact analysis
- Experiment with different analysis techniques to find those that resonate with the team
- Embrace the role of "Team Member" and take on any tasks necessary for the team to be successful

Analysts are often the knowledge and glue that can make a set of disparate team members gel. Think of all of the knowledge and information about a product the team is building as the crystals at the end of a kaleidoscope: analysts are the ones equipped to turn the kaleidoscope just the right way (by using the right analysis technique). The right techniques at the right time can reveal the piece of information needed to make the right decision and move forward.

Appreciation and Approach

Fantastic thought leaders and subject matter experts are defining Agile analysis approaches. These approaches are paving the way for analysts in organizations adopting Agile practices. If you're looking to learn more turn to the following references and resources:

- Book: Discover to Deliver by Ellen Gottesdiener & Mary Gorman
- Book: Impact Mapping by Gojko Adzic
- PDF: Agile Extension to the BABOK® Guide from IIBA®
- YouTube Video: Unleashing the Power of Behavior Driven Development (BDD) by Jeffrey Davidson: https://www.youtube.com/watch?v=FL9OhjO9U2k
- Blog: The IT Risk Manager (http://theitriskmanager.wordpress.com/) by Chris Matts

A final recommendation for applying analysis techniques to Agile: consider leveraging user stories to determine what deliverables a team should produce. Examples include:

- As a new team member, I need a guide that summarizes how our systems work, so that I can get up to speed and start adding value quickly.
- As a support analyst, I need an operations guide, so that I can triage defects and determine the severity of incidents.

These stories, partnered with acceptance criteria, will ensure deliverables are tailored to meet the need of the groups requesting them.

Business Analysts have much to offer in Agile environments. Honor the value they add to Agile teams.

Build Useful Solutions that Inherently address the Needs of users… By Embracing the Spirit of Agile's Simplicity and being…	*Awesome at* *Nailing it down* *All while* *Looking for ways to increase* *Your productivity and* *Still maintain* *True balance* [5]

References

[1] http://www.agilemanifesto.org/

[2] http://www.agilemanifesto.org/principles.html

[3] A Guide to the Business Analysis Body of Knowledge® (BABOK® Guide) Version 2.0

[4] http://www.davisbase.com/proactive-stakeholder-engagement/

[5] *BA/SA/BSA: What do you call yourself? Technology-Enabled Business Solutions* 4/3/2013 blog.fusionalliance.com (Attribution only applicable to the "ANALYST" portion of the mental model)

Self-Selecting Teams - Why You Should Try Self-Selection

Sandy Mamoli, Nomad8, www.nomad8.com, @smamol
David Mole, david [at] mole.uk.com, @molio

Fast growth has a way of forcing organisational change on a business, but it also presents opportunities to try new ways of working. When one of New Zealand's biggest eCommerce providers hit a new level of growth, we saw an opportunity to drive productivity by reorganising its technology department into small, stable, Agile teams. And we decided the best way to go about it was using Self-Selection: in other words, to trust the people who work in the department - the engineers, testers, BAs, designers, and Uxers - to come up with the best solution.

1. What is self-selection and why should you care?

Self-selection is a facilitated process of letting people self-organise into small, cross-functional teams. We think it is the fastest and most efficient way to form stable teams, based on the belief that people are at their happiest and most productive if they can choose what they work on and who they work with.

To avoid confusion, we are not referring here to self-*organising* teams. Self-*organising* teams are groups of motivated individuals who work together toward a shared goal and have the ability and authority to take decisions and readily adapt to changing demands. We like self-organising teams, but that's not what this article is about.

This article is about self-*selection*, which is the process you use to set up self-organising teams in the first place. Self-selection happens at an organisational rather than at a team level and is a way to get everyone *into* teams. Another term for a self-selected team is a self-designed team.

We have done it and so can you

In this article - the first of a two-part serie - we will tell the story of how we used self-selection to decide on the structure and composition of 22 new Agile teams, a process that involved more than 150 people. We will not only share a case study but also a repeatable process for facilitating self-selection at scale - and maybe even convince you that self-selection is not only valid but highly rewarding and in all likelihood a successful approach for many organisations.

Who are we?

We are consultants who have spent several years doing transformational work with one of New Zealand's biggest eCommerce providers. If you aren't a New Zealander chances are that you've never heard of the company we are talking about. If you are a 'Kiwi', you most definitely will have: three quarters of the population have a member account on its transactional site.

The site is a popular place for Kiwis to buy, sell and trade everything from cars and antiques to clothes, crafts, property and farm gear. It is a Kiwi success story, having grown over the past 15 years to a unique position where it commands two-thirds of New Zealand's internet traffic with 1.5 billion page serves a month. And it is a fast-growing business: a year ago there were 110 employees in the technology department, now there are 190. In all, there are 400 staff and no sign of the growth slowing down.

What problem did we need to solve?

We reached a point about two years ago where the company was increasing its staff by roughly one person a week but adding new people no longer meant we were necessarily getting any more done; if anything delivery was slowing down. Somewhere along the way a web of dependencies had evolved where every person and project was reliant on someone else and there were a large number of handovers between professional groups. Projects were constantly being left on hold because there was no one available to work on them; everyone was busy somewhere else.

We wanted to avoid these delays of waiting for people to be freed from other projects, and we wanted to minimise handovers with their associated loss of tacit knowledge and create small units where the whole is greater than the sum of its parts. We decided we needed to pull people out of a complex matrix and get them into fixed, stable teams where we could make sure that one person would work on only one team, and one team would work on only one project at any time.

Happier, more productive teams

Anyone who has ever been on a high-performing team will know what it feels like when a team begins to truly gel, when everyone is committed to and enthusiastic about a shared goal and when people know each other well enough to support and hold each other accountable for great performance. These high-performing teams exist not only in software development but also in sports and in any area where a group of people need to manage their interdependencies while working towards a shared, compelling goal [1].

These teams have also been more productive. Recent research by Rally Software showed an almost a 2:1 difference in throughput between software development teams that were 95% or more dedicated compared with teams that were 50% or less dedicated. The 2014 white paper "The Impact of Agile Quantified" [2], based on the analysis of the process and performance data of nearly 10,000 teams, indicated that stable Agile teams result in up to 60% higher productivity.

One reason for greater productivity in stable teams is that they don't have to go through the team-building stages of forming, storming, norming, performing - as defined by Bruce Tuckman in 1965 [3] - over and over again. People working in small, stable teams are also happier and more content. Certainly our own internal surveys show that job satisfaction has increased since we started working in fixed teams.

The need for speed

When we first started implementing stable teams it was on a small scale, introducing one team at a time in a controlled manner. But as the company grew ever faster we realised we were going to have to be able to scale up, and quickly. People by now were starting to want to be part of this new way of working. We'd also seen measurable benefits and knew we wanted to extend the 'stable' approach right across the technology department. So the only question now was how to make it happen in a fast and efficient way.

The art of team design

We started looking into the design of teams, which some research suggests is *the* most important factor in team performance. Studies [4] conducted by J. Richard Hackman, for example, found 60% of the variation in team effectiveness is attributable to the design of the team, 30% to the way the team is launched, and 10% to leader coaching once the team is under way.

We believe that designing a team doesn't necessarily mean picking the best people, but rather deciding on the best *combination* of people based on their interdependent skills, preferences and personalities. We looked at two methods of designing teams:

- Managerial selection: Managers decide by executive decree
- Self-selection: People decide for themselves which team they want to work in

Managerial selection breaks when organisations grow

Managerial selection is the traditional way of deciding who should be in which team. Managers design teams based on their knowledge of people's skills, personalities and who they think would get along with whom. In a small company this often works well - a good manager is aware of relationships between people and knows the skills, personalities and preferences of each of them. Often they come up with team compositions that can be mostly right and it is a quick way to get team selection done.

Where this model breaks down is when a company grows. Managers might still know their direct reports' skills and personalities, but it becomes increasingly difficult to understand the intricacies of relationships between people as the number of relationships increases exponentially. Managerial selection made sense in its historical context of industrial factories where workers' tasks were relatively simple and repetitive, but makes much less sense in the complex and collaborative workflows of a tech department today.

The same is true of the carrot and stick approach to motivating staff, which emerged from Frederick Winslow Taylor's [5] theory of management in the 1900s which said that people charged with repetitive and boring tasks were best incentivised by monetary rewards. US author Daniel Pink turned the tables on that idea in his 2009 book Drive: The Surprising Truth About What Motivates Us [6], pointing out that today's work tasks are mainly creative and complex, and citing research that shows the best motivators in such an environment are autonomy, mastery and purpose:

- Autonomy [7] provides employees with autonomy over some or all of the four main aspects of work: when they do it, how they do it, who they do it with and what they do.
- Mastery allows employees to become better at a subject or task that matters to them and allows for continuous learning.
- Purpose gives people an opportunity to fulfil their natural desire to contribute to a cause greater and more enduring than themselves.

In the light of Pink's book, knowing that highly motivated people perform best, and considering our own observations of managerial selection breaking down as a company grows, we started to look more closely at self-selection as a team design tool. What better place to start offering autonomy than by letting people decide for themselves which team they wanted to work in?

Self-Selection has a good (and interesting) track record

Self-selection is not a new or unproven idea. Leo McKinstry [8] described one of the earliest and most successful self-selections at large scale in his 2009 book "Lancaster - The Second World War's Greatest Bomber" [9] about the RAF's Lancaster bomber crews in the early 1940s. During World War II new flight crews had to be formed after short training periods and the creative solution to forming these teams quickly and efficiently was to have them self-select into teams. The result was one of the most effective, well put-together teams [10] in the history of the war.

Fast-forward to 2004 and Atlassian [11], an Australian IT solutions company, created the Ship-it day concept [12]. A 24-hour hackathon, Ship-it Day became highly successful and was cited by Daniel Pink in Drive. Ship-it Days give people 24 hours to work on whatever they want - as long as it is not part of their regular jobs - and the aim is to complete something within a 24 hour period. The idea was originally named "Fedex Day" after Fedex's 1980's slogan "When it absolutely, positively has to get there overnight".

We have had many Ship-it Days over the years at our Kiwi eCommerce provider and it is always been a joy to see an entire organisation self-organise into small teams and work away on projects of their own choosing. During our last Ship-it Day we had roughly 80 people in 15 teams working on 15 projects that all benefited the company in one way or another. We saw Ship-it Day as a study in what happens when we give a group of people complete freedom to work on what they think is important, with whomever they like, and using any approach they think will get the job done.

Here are some of our observations about what happens when people self-select:

1. **People naturally form small, cross-functional teams**. Teams are between three and six people and team composition is based on skill rather than role. There is no one person per skill and t-shaped people [13] who are good at collaborating are in high demand.
2. **No one chooses to work on more than one team or project.** Time and again organisations fall into the trap of optimising "resources" rather than focusing on outcomes. People often believe that multi-tasking, having people work across several projects, and focusing on resource utilisation is the key to success, when in reality it is not. It is interesting to note that when people are determined to ship, no one thinks it is best to do more than one thing at a time and nobody believes they are more valuable as specialists across teams than as generalising specialists within one team.
3. **People communicate face-to-face.** There are barely any discussions about process or how to communicate. People just talk and co-ordinate and collaborate as needed. Things are much faster that way.
4. A **shared, clear goal makes everything so much easier.** When people buy into the goal and know which problem they are solving and why, things become a lot easier. It is easy to make decisions and reach consensus when people understand and support the objectives and constraints around a project or product. Selecting which project to work on worked wonders for ensuring that the team had a shared and compelling goal.
5. **People are highly motivated, enjoy the experience and get lots of work done.** Some of the projects people built, such as a "Is someone in the shower?" app, a virtual receptionist, or the room booking app "Get A Room", were simply incredible and are still in use.

The solution to our problem

It was partly our Ship-it Day successes that inspired us to try self-selection on a bigger scale and gave us confidence that people could in fact work like this every day, and not just in 24-hour hackathons. We realised that the most likely way to solve our challenge of getting people into well-designed teams quickly was to take the problem to the people involved. To leverage their knowledge, motivation and enthusiasm to come up with a better and more widely accepted solution than managerial selection could have given us.

2. How to Conceive and Carry Out a Self-Selection Event

Having decided that self-selection was the most efficient way to establish stable, happier and more productive teams the next question was: how do we go about it at scale? Should we follow the Lancaster Bombers' lead and just get everyone into a giant hall and tell them to get on with it? Or was there something more workable for us than that?

We tried to research the concept and ran the usual Google searches but it appeared that either no one had carried out a self-selection exercise at this scale or if they had they hadn't published the process or results. So we had to come up with our own process from scratch - one that's since been revised, tested and improved and which we are now sharing with you here. We have also shared these steps with a number of other organisations that have since run successful self-selection processes.

Before Self-Selecting: The Preparation Checklist

Preparation is incredibly important when you're taking on a self-selection exercise at scale. We have seen self-selection events fail spectacularly when preparation wasn't done well. In fact, we suggest erring on the side of over-preparation, not least because it will make you feel more confident and relaxed during the event. A relaxed facilitator is more likely to achieve a good outcome.

Readiness Check

The first step in preparation is to ask the question: "Are you in a strong enough position to do this?" Do you have an environment where this could work? Do you have the kind of people who are flexible and open enough to try a process like this? Are there any festering problems that could derail the process? Are there things you need to establish *before* you start on self-selection?

You may not be able to pull off self-selection if other things are too problematic. For example, if you don't already have the concept of fixed teams you may struggle to get people to choose a new home. So you might decide that you need to solve one or two problems before you start on self-selection. You can see how we went about that by reading/watching about our experience with Portfolio Kanban. [14] You might also consider running a Ship-it Day ahead of time, which will allow you to observe behaviour and see whether people naturally self-select and how well they work in teams.

Run a trial

In addition to (or instead of) a readiness check you can run a trial self-selection event to gauge the process and manage your risk. A trial can give you a lot of information at little cost and with little or no downside. We opted to carry out a trial self-selection event at our satellite office, which gave us a more controlled environment and fewer people to test our process on.

We ran the event with 20 people and started the day with just a carrier bag of sticky notes and a lot of good intentions. By the end of the day those 20 people had formed into the three teams that we had aimed to end up with and we knew we had a process that worked. Of all the things we learned that day, one of the most powerful was that our worst fears were unfounded: there were no fights, no crying in the corner, and no empty teams at the end of the day. We don't think we could have carried out our full-scale selection process without first having done this successful trial run.

Another way to trial the process could be to run a *practice* self-selection event where you can test the concept and refine it without any real risk. We have known companies to do this and it can work well. But be aware that people behave very differently when they are not making 'real' choices - it is like playing poker for no money: everyone goes all in because they have nothing to lose.

Define the Teams to Select

Ahead of the self-selection event, it is important to clearly define the teams that are required. This could simply be based on your current structure or gaps, or it could be a more complex proposition and even require a company-wide prioritisation to get the right teams established. Each team should have a name, a clear mission for what it will do and a product owner established in advance. When people choose what team they want to work in, they will want to understand what they are likely to be working on, the problem they are trying to solve, and who will provide guidance.

This is a potentially lengthy step but the work can be delegated to the product owners. The better they can explain their team, its mission and how they themselves work, the better chance they have of attracting a great team on self-selection day and in turn being successful. You don't want people walking out of the self-selection event feeling like they are not sure what's happening or thinking 'this is not what I signed up for'.

Logistics: where, when and who

Establishing early where and when your self-selection event will take place can help your communication effort, build trust that it will actually happen, and give confidence to people that everything is planned and under control.

The following details should be defined as early as possible:

- What time/date will your event take place? (It is probably a good idea to have a backup date in case of problems or illness.)
- Where will it take place? (Note, you will need a big, open collaborative area with lots of wall space.)
- Who will be invited?

Communication: early, often and honest

Communicating well might be the single most important thing you can do to pull off a successful self-selection event. We have seen a pattern emerge when people hear about self-selection for the first time. The first reaction tends to be positive but it can move quite quickly to fear and resistance. Fear of something new and different, fear of what might happen, fear of being stuck with someone you don't get on with or of being stuck in a team that you can't change your mind about later.

People will throw a lot of questions and what-if scenarios at you and you need to be prepared to answer them honestly. Getting the communication strategy right could be the difference between your self-selection event going badly (or not taking place at all) or being a roaring success.

Based on our experience, we recommend the following approach to communication:

- Talk to as many people as possible, from start to finish
- Actively listen to their concerns
- Be patient with people as they work through their fears
- Record people's fears and what-if scenarios
- Manage risks actively
- Paint a very honest picture about the worst-case scenario, which is never as bad as people think
- Talk to people individually *and* present in groups
- Show real examples from your Ship-it Days, trials and from other companies
- Ask people whether it is they, or their manager, that knows more about where they should be placed

Establish the Rules and Constraints

We recommend keeping rules and constraints to a minimum. The more there are, the harder it will be to solve the problem and the more likely that people could feel they are not self-selecting at all but simply moving into - or worse, being manipulated into - pre-chosen allocations.

We had just three rules during our self-selection events. Teams had to be:

- Capable of delivering end-to-end
- Made up of 3-7 people
- Co-located

Prepare your materials and stationery

This needs to be an interactive and visual event, where everyone can see what's going on and participate throughout the process. You're going to want to start the event with some of the details already on the wall: a visualisation of the status quo, a team diagram for each team you're aiming to create, checklists for the skillsets required, a visualisation of the rules and so on. On the day, people will indicate which team they want to join by sticking their photo on that team's diagram.

So you're going to need a lot of stationery and materials including:

- Photographs of the people participating
- Skills checklists
- FAQs
- Team diagrams for the wall
- The rules written up to display around the room

The day before our self-selection event we had what can only be described as an arts and crafts day. Cutting out photographs of everyone involved, preparing colour-coded skills checklists and preparing large team diagrams pre-populated with any information that we already knew such as team names and/or the product owner photo.

Little details are important here. You need to leave enough room on each team diagram to add the required number of people's photographs, and be careful where you put the product owner's photo in the diagram so you don't inadvertently imply an unwelcome hierarchy.

It is also important to decide whether your self-selection event will be starting from scratch or whether you will start your problem-solving process from the status quo. The advantage of starting from scratch is that it can be simpler and easier to throw out current constraints and start from a blank sheet. It can reduce the complexity of the problem and allow people to consider options they may not have otherwise realised would be possible. On the other hand, starting from the status quo makes the process very real, and it helps those people who want to stay exactly where they are (and there will probably be some) to do so and not feel like they are being bumped.

3. On the Day: Running the Self-Selection Event

Set up the Room

We recommend getting there early so you have plenty of time to set up the room, and have all your stationery, photographs and materials neatly laid out. The last thing you want is for a team to solve a problem during the day but not be able to reflect it on the finished product because you're short of blu-tac or sticky notes.

Since self-selection is a physical event, the emphasis when setting up the room should be on space and collaboration. It may be a good idea to get the group to help set up the room by moving chairs and so on, because then they will start the day as you hope they'll carry on: by moving around and talking to each other.

Product Owner Presentations

People need to know what they will be signing up for, they also need the opportunity to ask questions, so the day should start with the product owners standing up to pitch their team to the group. They should explain the purpose of the team and the type of things that they will work on. If they know their current projects those should be explained too, but projects can change and be aware that this could be the wrong level of detail. The type of product the team will support and the type of work (front end vs. back end, for example) might be more appropriate.

Explain the Rules

Talk people through the rules for the event, and have the rules displayed prominently around the room.

Go!

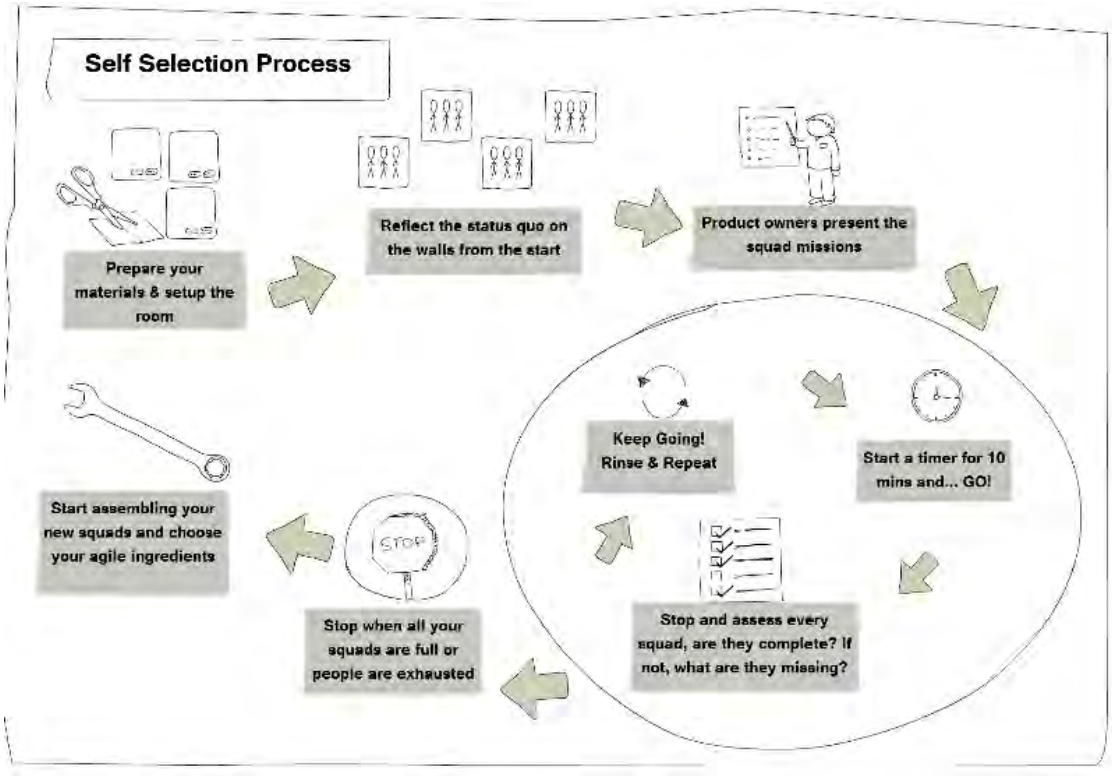
Start your first self-selection round. This is where people walk around talking to product owners and thinking about which team they want to join. When they choose a team, they blu-tac their photo into that team diagram. Be strict on your timing. We recommend a 10-minute timebox because that's enough time for people to have the conversations they need to have and also for them to overcome any nerves about moving or selecting a team. When the time is up, be very strict about stopping for your first 'checkpoint' (you may need a whistle!)

Checkpoint

It is vital to check in and publicise each team's current status after each round. To do this we use a checkpoint. At the end of each timebox, everyone stops and you use your checklists and other visual indicators to show how many full teams you have at this point. If this is your first checkpoint then don't expect to have solved the problem straightaway. The hope at these checkpoints is that one group will say they are missing two developers, for example, and another group will demonstrate an over-supply of developers, thereby ensuring those two groups can talk during the next round.

One by one a self-appointed spokesman from each team should announce:

- Whether the team is full
- What gaps they have
- Any problems or blocks they have encountered



Rinse and Repeat

The 10-minute timeboxes, each followed by a checkpoint, should then be repeated indefinitely until all the problems are solved, or the same problems are being repeated and people appear to be stuck. If the problem is solved then, congratulations, you can send everyone home! If the problem isn't solved it is time to change gears and tweak the format.

Tackle the outstanding problems

You may want to send people who are part of new, fully formed teams home at this point and you could change the room format to bring people closer together. If you still have problems that aren't being solved by more rounds of self-selection then you may have to dive into them head first. The detail will vary according to the specific problem being solved and the people involved. For us, the process involved tackling one obvious problem or bottleneck at a time and engaging the whole group to solve the problem.

For example we had a shortage of designers and no matter which way we cut it, we just couldn't make fully formed squads. One solution would have been to thinly spread the people we had across multiple teams, but that would have left everyone short so as a group we decided to populate as many full teams as we could and then use empty cards to represent the people we needed to hire after the event. This was great information to have: prior to the event we didn't know who we needed to hire or which teams they would join.

Go Home!

Don't stay too long, this is a surprisingly exhausting process and if you haven't solved the problem so far then it might be time to let people go away, give it more thought and come back to tackle it again the next day.

4. After the Day: Making this Real

At the end of a self-selection event you will have a lot of paper and hopefully lots of self-selected fully skilled teams. But so far this is just a lot of diagrams and you need to go about making this real. If possible, you should meet with each new team the very next day. It is vital that you build on the momentum you've created and don't let people go back to their day jobs. We found that the Lean Coffee [16] meeting format worked really well for talking to each team, allowing them to voice any concerns, and importantly to start talking about how and when to set the team in motion.

Quite often other work will need to be finished first so creating a schedule is important. It is then a case of keeping momentum going as your new teams work through their scheduled backlog and then turn their energies to new projects.

In Part 2 published in the next issue of Methods & Tools, we will talk about how to settle in your new teams, permission vs. forgiveness, and tips for getting the best out of your self-selection event.

References

1. http://www.estherderby.com/2011/02/the-0th-trap-of-teams.html
2. https://www.rallydev.com/finally-get-real-data-about-benefits-adopting-agile?nid=6201
3. https://en.wikipedia.org/wiki/Tuckman's_stages_of_group_development
4. http://www.estherderby.com/2011/11/miss-the-start-miss-the-end.html -sthash.65FkEH9y.dpuf
5. https://en.wikipedia.org/wiki/Frederick_Winslow_Taylor
6. http://www.danpink.com/books/drive/
7. https://checkside.wordpress.com/2012/01/20/motivation-revamped-a-summary-of-daniel-h-pinks-new-theory-of-what-motivates-us/
8. http://www.amazon.com/Lancaster-Second-World-Greatest-Bomber-ebook/dp/B002VCR07G/
9. http://www.amazon.com/Lancaster-Second-World-Greatest-Bomber-ebook/dp/B002VCR07G/
10. http://lunatractor.com/2014/01/12/self-selecting-teams-tales-from-ww2-lancaster-bomber-crews/
11. http://www.atlassian.com/
12. https://www.atlassian.com/company/about/shipit
13. http://chiefexecutive.net/ideo-ceo-tim-brown-t-shaped-stars-the-backbone-of-ideoae™s-collaborative-culture
14. Video: http://nomad8.com/my-portfolio-kanban-talk-on-infoq/
15. Paper: http://nomad8.com/portfolio-kanban-seeing-the-bigger-picture/
16. http://leancoffee.org/

Collaborative development of domain-specific languages, models and generators

Juha-Pekka Tolvanen, jpt [at] metacase.com, @mccjpt
MetaCase, http://www.metacase.com

Almost all software development activities require collaboration, and developing domain-specific languages is no exception. Language users provide feedback as the language is developed, and also different parts of the language can be developed in parallel: for example, one developer can focus on the abstract syntax, another on the notation, a third on code generators, and a fourth on integration with the development process. This collaboration becomes even more relevant when a number of integrated domain-specific languages are developed. In this article we share our experiences on how teams can collaboratively develop and use domain-specific modeling languages, and what benefits this collaboration provides.

1. Domain-specific languages and collaboration

Domain-Specific Modeling (DSM) has become popular in recent years. This is no surprise given the reported benefits of significantly improved productivity and quality [Sprinkle et al. 2009]. Working on the higher level of abstraction offered by a language, with automatic transformations producing the lower level "implementation", has been a recipe for success for decades. Not all modeling languages, however, lend themselves to automatic transformation. Languages that do not focus on a specific problem domain - e.g. general purpose languages like UML and SysML - cannot raise the level of abstraction up to the problem domain. Nor can they guarantee that the models created are complete and correct to enable code generation. These general purpose modeling languages are typically used only for sketch models, which are thrown away afterwards [Collins-Cope 2014, Petre 2014]. In contrast, a DSM language focuses on a narrow area of interest and enables executable specifications [Sprinkle et al. 2009]. Code generators provide automation by reading the models created with the language to produce various kinds of artifacts like code, configuration, test data and documentation. This automation is possible because both the language and generators need to fit the requirements of only a single domain, often inside just one company.

Creating domain-specific languages calls for collaboration. First, it is common to distinguish language creation and language use. Second, and related to language creation, the abstract syntax of a language, its rules, notation, generators and tooling is not always created by a single person. It is therefore natural that the work can be shared and the development team can collaborate. Third, while a domain-specific language focuses on a small area of interest, a single language is not always enough. Applications are large, they have connections to other systems, include various sub-domains and different developers and tasks require different views. This again calls for collaboration to provide several distinct yet integrated languages. Finally, tools play an important role and since some tools require considerable more effort to provide modeling language support than others (for a review see [El Kouhen et al. 2012]) it is natural that the work is shared.

2. Example language for collaborative development and use

In this article, we focus on collaborative language development and use. We describe the different ways to collaborate and the benefits they provide. To make things concrete we use an example case from developing heating systems. Here two integrated domain-specific modeling languages are used: one specifies the structure of the system such as various instruments connected via pipes, and the other describes behavior of an instrument (Figure 1).

The implementation of these languages is documented in detail and available for modifications [MetaCase A 2014]. Regardless of the tooling applied, the language development practices and need for collaboration are naturally universal.

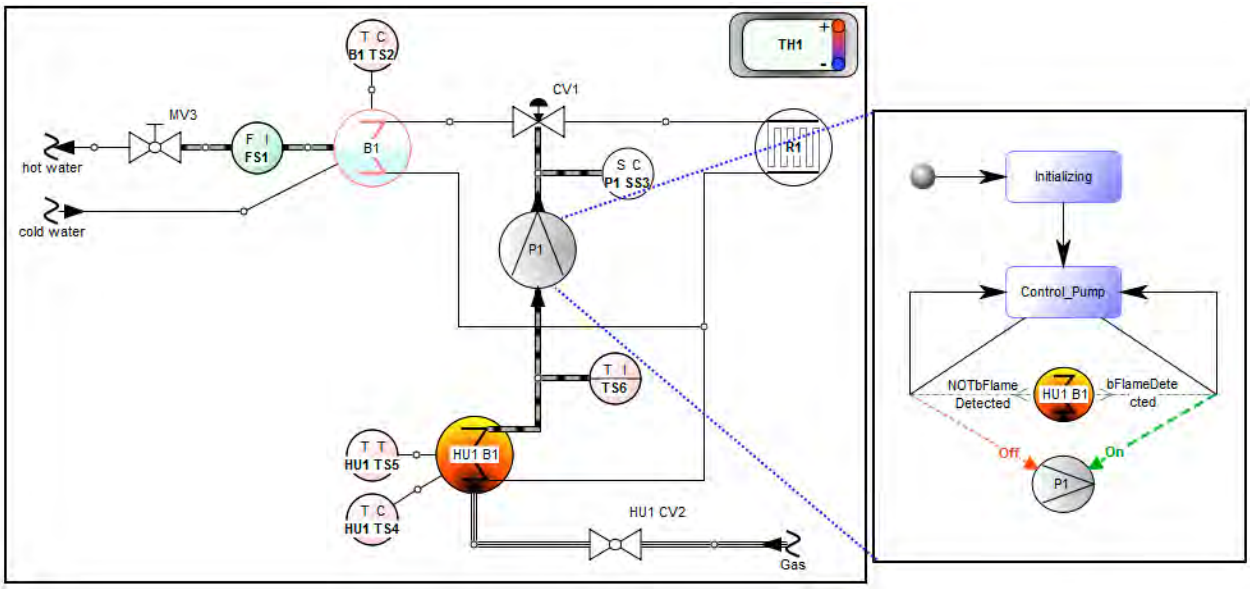


Figure 1. An example of DSM: specifying structure and behavior of a heating application

On the structural part, the diagram on the left shows valves, sensors, a burner, a pump and other instruments along with their pipe connections. All these modeling concepts are also directly the domain concepts. In other words, the language maps closely to the problem domain. The behavior of these instruments is described with another modeling language. The state machine on the right shows an example of this, defining the behavior of pump ‘P1’. In addition to states and transitions the instruments of the heating system are used as conditions and actions. For example, the behavior part of pump ‘P1’ depends on the status of burner ‘HU1 B1’: it is turned on if flame is detected from the burner.

These two DSM languages are integrated as it would not make sense to specify behavior for instruments that are not part of the system structure and vice versa. These two languages also share some of the same concepts, like the burner and pump illustrated above. The domain-specific models are not just pictures, they are formal specifications, their consistency and completeness is checked by the language definition, and most importantly they can be used to generate fully functional production code directly from the models. The same models can also be used to produce deployment and installation, test data, material calculation, documentation, etc.

3. Collaboration between language engineer and language user

The most typical form of collaboration is between language engineer and language user. In the best case, once any element of the language is defined, language users may immediately test it. An example of such tight collaboration is shown in Figure 2, where the left side illustrates language definition and the right side language use. Within the heating system the language definition on the left shows the concept ’Sensor’ and its properties along with the definition of its notation. The diagram on the right describes its use while specifying a temperature sensor ‘TS6’ that is installed in the control room to indicate the current temperature in the attached pipe.

The model showing the sensor can thus describe just those aspects of sensors that are defined in the language (aka metamodel). Also, sensors can only be used in the manner the metamodel allows, e.g. they need to be connected to a pipe but not to another sensor. The language definition may also include more complex rules, like those related to a kind of sensor: e.g. temperature sensors must have one connection to a pipe, but flow sensors must have two.

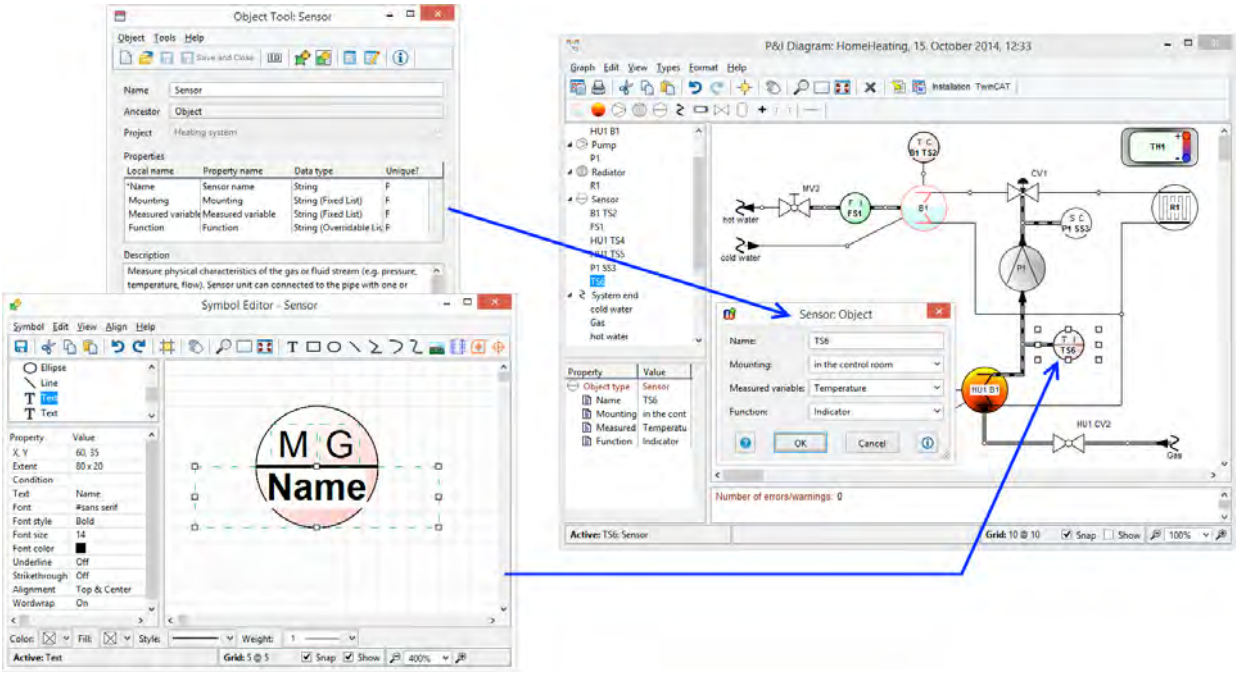


Figure 2. Collaborative language definition and language use

In this case the language development practice, and supporting tool, enables working as a pair in an agile manner. Any change in the language definition can be immediately applied by other members of the team. Such a tight collaboration between language definition and its use brings several benefits. Many of these are common for all user participatory approaches, but particularly relevant as often language engineers do not have prior experience on creating languages.

The benefits of collaboration include:

- Enable early feedback and validation of language definition. Users may immediately test the language and not only verify that the definition is correct, but also validate that the language lets users specify the kinds of things for which it is intended.
- Minimize the risk of creating the wrong language constructs and enable the language to be defined in small increments. This is particularly important if the domain is new, evolving, or the language engineers do not have prior experience of language development.
- Language adoption and acceptance improves since language users are involved early in the language definition.
- Speeds up the move to DSM, since while generators are being developed language users can already start modeling. This is particularly relevant for shorter term projects in which languages are needed quickly. If language definition takes months the projects that would need the language have already ended before the language is available.

This collaboration becomes even more relevant during the language maintenance phase when there is already a substantial amount of work done with the modeling languages. Any change in the language definition can then be immediately checked and reflected against the existing models. Also language users can see the influence of the language modification and can propose suitable policies for model updating. For example, easy parts like renaming of language concepts can be automated so the language notation and semantic rules change when the abstract syntax of the language is changed. Also changes made to the language definition can be reflected automatically to the existing models to update them accordingly. If the language update is such that it cannot automatically be reflected in the models, generators can be made to report on those parts of the model that cannot be updated automatically, and have thus been left for modelers to change.

4. Collaboration while defining the same language

A single person is not necessarily good at defining all parts of the complete modeling solution. It may also be organizationally wise to divide the work among several persons. Perhaps one of the most typical ways to divide the work is between the creation of a modeling language and related generators. Figure 3 illustrates this collaboration. On the left side of Figure 3, a language concept 'Valve action' is defined with its properties such as action (e.g. on, off) and valve position (e.g. left open, right open, both open, both closed). On the right side, the generator developer defines a code generator for actions within a state transition and uses the same language construct 'Valve action'. The properties of 'Action' and 'Valve position' are here used to produce valve-related actions.

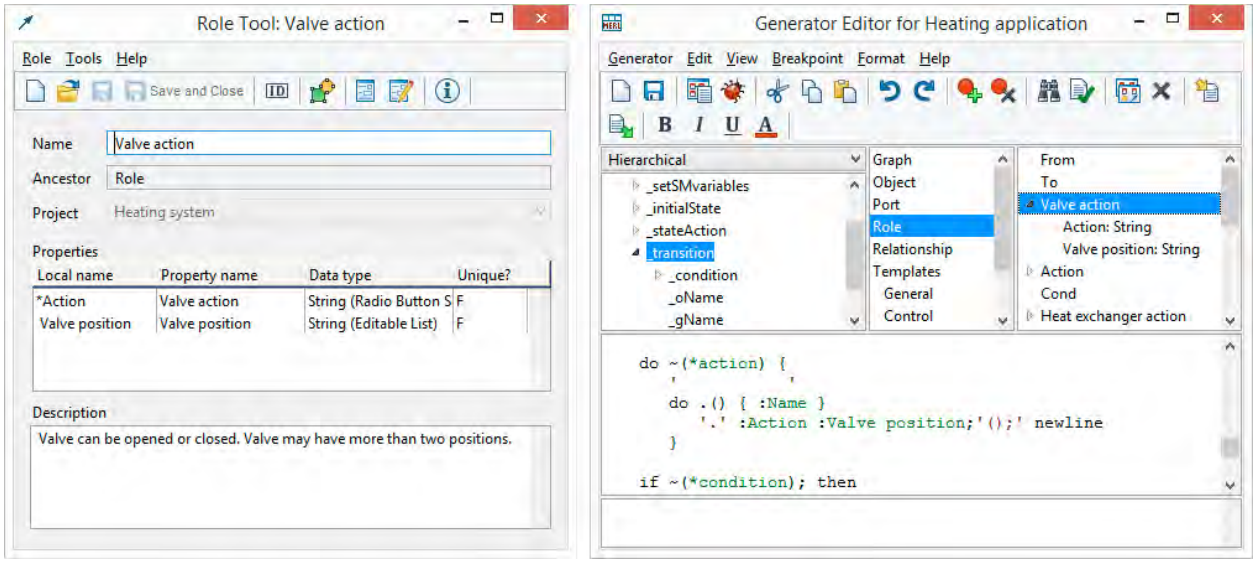


Figure 3. Collaborative development of metamodel and generator

Here the metamodel of the language and the generator are both being defined simultaneously. If the same language is used to create models for different generation needs, several generator developers can access the same metamodel definition. Perhaps the most typical order in which language definition takes place is to first focus on abstract syntax defining what kind of models can be made. This can then be extended with rules for keeping models syntactically correct, complete and consistent while generator development is started. Often in cases where the visual appearance of models is relevant, like in user interfaces, hardware or physical devices, the development of notation may also be delegated to other people than those defining the metamodel.

The benefits of having several people involved in creating the modeling solution are:

- Utilize expertise from different people. While a language engineer usually focuses on the language's abstract syntax and static semantics, others, including future language users, can define the notation. This also improves acceptance of the language as concrete syntax matters - in particular when starting to work with the new language. Also different generator needs call for different kinds of expertise: while one may focus on generating code in a particular programming language, others can make generators for build scripts or integrate with existing libraries. Generators and scripts can also be implemented by other people to check models, annotate errors, provide guidance during modeling, documentation, etc.
- Language and generators can be checked early while being defined. As in any teamwork, several people see more than one, and can discuss about language definition and generators as well as test them in collaboration - even using the same jointly developed models.
- Development of languages and generators is sped up: not only because different generators can be developed by different people, but because things like notation and some of the checking rules do not need to be completely ready before making generators. This is important as generator development usually takes more time.

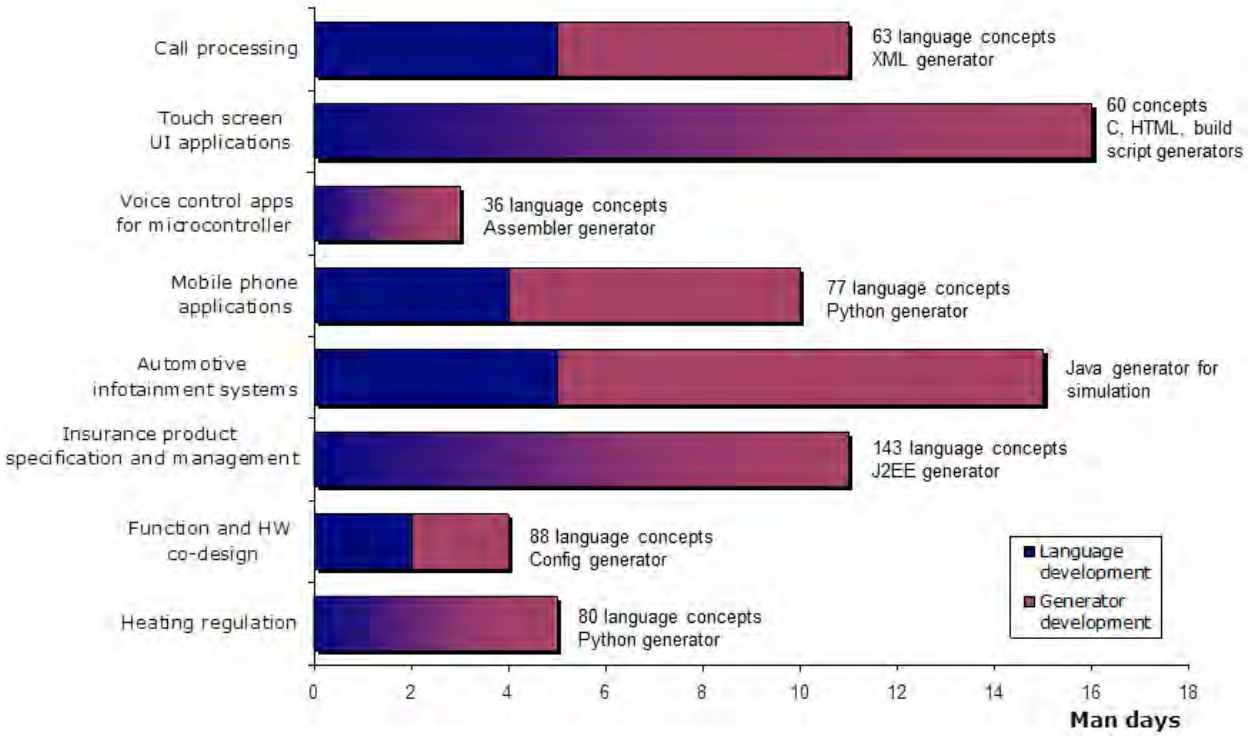


Figure 4. The effort to define a modeling language and the effort to define a generator

Figure 4 illustrates the division of effort by inspecting the time needed to develop a modeling language and time needed to develop a generator. In these eight cases where data was gathered on the effort needed, the generator development usually took more time than the definition of the language (its concepts, constraints and notation). All the above cases focused on creating a code generator for one target only. When developing several generators, e.g. for different target platforms, the effort to develop the second and subsequent generators is usually smaller. The language is then better known and the generator developer has identified good practices to access models, and may reuse parts of the existing generators. For example, at Panasonic the second generator for a different, albeit smaller, target platform took significantly less time than the first one [Safa 2007].

The effort to build generators is naturally tool dependent. If a generator is disconnected from the metamodel, needs to parse temporary models, or uses model-to-model transformations to combine different models, it takes more time to develop than if the generator development tool can access the jointly developed language (as in Figure 3 above). When different parts of the language, such as its abstract syntax, constraints, notation and generators, can be accessed and combined they can be better tested and changes made in one part be more easily traced to other parts. This will tend to lead to a better quality modeling solution.

5. Collaboration while defining several integrated languages

When several domain-specific languages are developed the number of people involved naturally grows too: different people tend to master different parts of the whole system. One language can focus on structures, another on behavior, a third on reusable parts in a library, a fourth on configuration and so on. Figure 5 illustrates the joint development of the two languages in our heating system example. The language on the left, P&I Diagram, is used to define the structure of pipes and instruments of the system. In this language, the behavior of pumps can be described with the 'Heating application' language. In other words, 'Pump' can be specified in detail with another language. This second language, based on state machines, is described in the window on the right. To support the nesting of states, each 'State' within the heating application can be specified with another submodel using the same 'Heating application' language. Integration among these languages is more detailed than shown in the figure as both languages also share the same concepts, such as some of the instruments. Examples of these two languages used in modeling were illustrated in Figure 1.

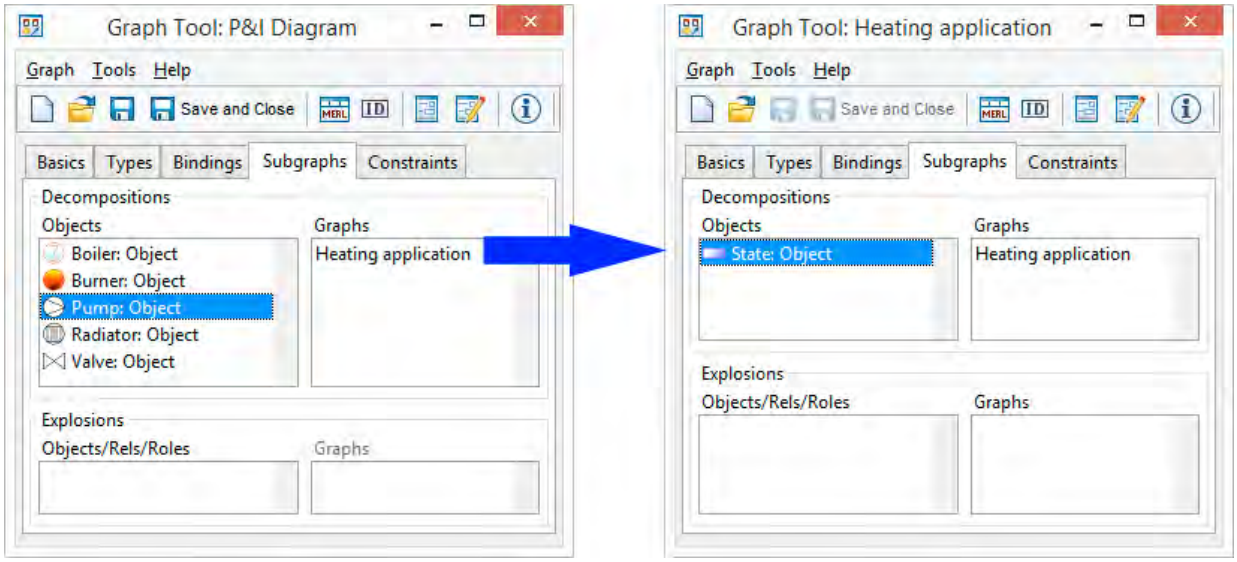


Figure 5. Integrated definition of the languages for heating system.

If several domain-specific languages are used there can also be several teams developing different languages. Based on our experience, the language development team is usually just one or two people, but the largest language development team I am aware of included over 20 people. Naturally the domain the modeling languages target influences this, e.g. if the intention is to gather and integrate knowledge from different disciplines and tasks (software, mechatronics, requirements, variability, configuration, deployment etc.).

The benefits of collaboration include:

- Languages may reuse common parts and the team can integrate languages based on a shared definition. This allows harmonizing the parts shared among the languages and better modularization.
- Integrated languages cover a richer variety of views or aspects of the system. This is important since otherwise the integration would need to be handled by defining and maintaining model-to-model transformations. Even worst, the resulting model transformations would provide a single, one-way-route only: migrating changes to models that have been subsequently edited is challenging or even impossible. Instead, integrated languages can make model integration easier. Consider a change to Pump 'P1' in Figure 1. Because the languages are integrated there is only one 'P1', and thus changing its properties in one diagram (and DSL) will change it in the other diagram too. The same applies at the language definition level: changing the definition of the 'Pump' concept used in both languages will update it in both languages.
- Shared expertise can be combined. For example in the case of automotive embedded systems, a language called EAST-ADL includes over 15 different sub-languages each covering different subdomains (architecture, safety, error modeling, requirements, variability, hardware etc.). A single person can hardly master them all along with their generators. Therefore for example the EAST-ADL implementation at [MetaCase B 2014] has been defined in collaboration among two language engineers - each focusing on different parts of the whole.
- Integrated languages support the development process. Rather than creating and editing the same kind of information in different phases by different people, integrated languages enable the same information to be shared. For example in our case of heating systems, the initial structure of pipes and instruments might be defined, followed by the specification of the behavior that references those structural elements. Inconsistencies among these two views can be reported, e.g. to check that the behavior definition does not reference instruments that are not specified in the system.

Based on the language creation projects we have been involved in, the best place to start are the stable parts - those domain concepts that are well known and have clear semantics. When defining several languages, it is good to identify early those parts that are reusable or enable integration. For example, when the same domain concept is used in several languages, each providing a different view of it, it is quite common that notation and generators related to it have similarities.

Tools naturally have an impact here, as some tools permit only one user at a time whereas others enable simultaneous collaboration. Ideally, several people can define different parts of the language at the same time - and language users can test them at the same time too. Since it is hard to get the single language right in the first place, the challenge to get an integrated language correct is even harder. It is therefore important to create the language and use supporting tools that make language modifications easy.

6. Conclusions

The creation of domain-specific languages, generators and models requires collaboration, but development practices - along with tools - tend to focus on a single developer only. This causes problems when gathering feedback from users, utilizing the expertise spread throughout the team, using several related languages, or reusing concepts defined as part of another language.

We described some typical approaches used to create modeling solutions, and the benefits collaborative development offers. Collaboration between the language developer and language users enables agile definition, in which a language can be defined in small parts and tested immediately by language users in concrete development situations. Tight collaboration enables a fast feedback loop, leading to better quality languages and user acceptance. Collaboration within the language development team allows the load to be shared and utilizes expertise from all members of the team. This not only leads to better defined DSLs but also to faster deployment. Finally, while DSM focuses on a particular small area of interest, a single language is not always enough. Applications are large, they have various aspects, different developers have different views, and no single modeling language can specify it all. In such cases collaborative development is the only realistic option: language definitions, generators and notation can be reused, languages can be integrated to support the development process, and different expertise can be combined.

7. Acknowledgements

I would like to thank Saïd Assar, Benoît Combemale, Steven Kelly and Janne Luoma for providing feedback to early version of this work.

8. References

El Kouhen, A., Dumoulin, C., Gerard, S., Boulet, P., Evaluation of Modeling Tools Adaptation, <hal-00706701v2> 2012, http://hal.archives-ouvertes.fr/docs/00/70/68/41/PDF/Evaluation_of_Modeling_Tools_Adaptation.pdf

Collins-Cope, M., Interview with Grady Booch, 2014, http://www.infoq.com/articles/booch-cope-interview

MetaCase A, Heating System Example, 2014, http://www.metacase.com/support/51/manuals/Heating%20System%20Example.pdf

MetaCase B, EAST-ADL Tutorial, 2014 http://www.metacase.com/papers/MetaEditPlus_Tutorial_for_EAST-ADL.pdf

Petre, M., "No shit" or "Oh, shit!": responses to observations on the use of UML in professional practice, Journal of Software and Systems Modeling, Vol. 13, 4, 2014.

Safa, L., The Making Of User-Interface Designer, A Proprietary DSM Tool, 7th OOPSLA Workshop on Domain-Specific Modeling, TR-38, University of Jyväskylä, 2007.

Sprinkle, J., Mernik, M., Tolvanen, J-P., Spinellis, D., What Kinds of Nails Need a Domain-Specific Hammer?, IEEE Software, July/Aug, 2009.

TDD with Mock Objects: Design Principles and Emergent Properties

Luca Minudel, luca.minudel [at] thoughtworks.com, @lukadotnet,
http://blogs.ugidotnet.org/luKa/

A team began to write code much easier to read, change and extend after adopting the practice of TDD with Mock Objects. And later the team developed the understanding of the design principles with the ability to put them into practice in the code written everyday.

This observation originated the intriguing conjecture that TDD with Mock Objects led that team to write code compliant with S.O.L.I.D. design principles and partially with the Law of Demeter as an emergent property. This originated the second intriguing conjecture that these tangible improvements of the code-base led that team to deeply understand the design principles and their practical applications as a result of a process of coevolution.

This is an exploratory observational study with the goal of understanding the phenomenon observed, identifying relevant variables, turning conjectures into a verifiable hypothesis whose general validity can be comprehensively investigated with a rigorous research and controlled experiments. This study recognizes the language ambiguities about TDD and the differences between person to person and team to team in the actual practice of TDD that have relevant consequences on the outcome. It recognizes also that, when talking about engineering practices intended for people in professional software production, people and context are relevant variables that matter too.

Test-driven development (TDD) is the technique that relies on very short development cycles, every cycle starts writing a failing automated test case and finish with the refactoring of the code [1]. TDD with Mock Objects emphasizes the behavior verification and clarifies the interactions between classes [8], [3] and [4]. Law of Demeter (LoD) is a design principle that promotes loose coupling between objects, encapsulation and helps to assign responsibilities to the right object [7]. S.O.L.I.D. are 5 object-oriented principles of class design to write code that is easy to reuse, change, evolve without adding bugs [9]. Emergent property is a novel and coherent structure that arise during the process of self-organization in a complex system [15]. Coevolution is a process where two interdependent systems change together in mutual adaptation [16] [17][18].

1. Introduction

The software development team of a leading F1 Racing Team was implementing software for the Formula One Racing Championship. The team was working with a large and complex code-base, with high pressure to deliver as much new features as possible and in very short deadlines.

The team was trained on Object Orientation with the goal of writing code that was easier to understand, change and evolve without adding new bugs. After the training, style of code written by the team did not change significantly. A year later the team was trained on the job using TDD with Mock Objects.

This article is a report of qualitative observations of the team and the code-base and a report of a qualitative experiment made outside the team with small code exercises. Both team members and developers that voluntarily participated to the experiment are uncontrolled groups acting in an uncontrolled environment.

2. Observations

2.1 Initial training on Object Orientation

During 2006 team members, divided in two groups and in two different moments, had intermediate training on Object Orientation.

2.2 After the training on Object Orientation

After this training some team members more experienced with the code-base and the application domains proposed some improvements to the design at the level of namespaces and assemblies (intended as the fundamental unit of deployment, versioning and reuse of compiled code like an EXE or DLL file) and a top-down approach to implement these changes. These ideas were not implemented and so improvements in the quality of the code produced day by day have not been noticed during 2006.

During 2006 the team was also practicing unit testing, with tens of thousands unit tests running on the automatic build server. The majority of the unit tests were actually more integration tests than real unit tests. Most tests involved external systems such as a database and involved different objects and layers at the same time. The tests suites were slow and some of them brittle. The code-base overall was hard to test. An effort was made using advanced features of commercial mocking tools to mock static classes, concrete classes, classes provided by external libraries and classes instantiated directly inside the class under tests.

2.3 The training on TDD with Mock Objects

In the beginning of 2007, two groups of software developers attended an internal hands-on training on TDD with Mock Objects. At the beginning of a Sprint one group of team members went into a meeting room. They brought with them one PC, one keyboard, one projector with the screen and the user stories selected for that Sprint. Team members contributed to the Sprint goal with their knowledge of the code-base, the application domain and the technology stack in use.

Two software engineers extremely experienced in the practice of TDD with Mock Objects joined the group and contributed to the Sprint goal with their knowledge of TDD and refactoring on large complex and legacy code-base. They showed how to implement the user stories guided by TDD and mocks, in quick (5-15 minutes) red-green-refactor cycles, constantly discussing together and rotating pairs at the keyboard. At the end of the week the user stories were implemented and accepted by the end users and released into production.

The week after the team and the two software engineers went back to the office and completed another Sprint. This time team members were working in pair as usual at their workstations and rotating pairs with the two software engineers.

The same experience was repeated with another group and after that the two software engineers joined the team full time. Team members from both groups immediately appreciated that the production code and the unit tests written during the training session were better than before.

We learned:

- how to use a refactoring tool to extract interfaces, break dependencies [2] and how to inject dependencies into parameterized constructors and in methods arguments,
- how to replace static variables and singletons with more testable code,
- how to wrap third-party libraries,
- how to use a mocking tool to mock dependencies and declare and verify expectations,
- how to test non-trivial objects in isolation,
- how to quickly navigate in the IDE between interfaces and the classes and tests,
- about the practice of avoiding getters and instead using Smart Handlers that are Visitor-like objects [6].

However this practice was not followed.

2.4 After the training on TDD with Mock Objects

Team members, after the training and after continuing to practice TDD with mocks, discussed the effects of this new practice on the code.

For example there were discussions about the parametric constructors used only by the unit tests; discussions about the large use of Interfaces (as intended in Java, C# or like Abstract classes in C++ with only pure virtual functions or like protocols in Smalltalk) defined to enable the mocking of objects; discussions about the larger number of small classes each one with a narrow responsibility; discussions about the use of default constructors or factories or Dependency Injection frameworks; discussions about wrapper created to break dependencies to external libraries and external systems; discussions about the increased use of containment over inheritance; discussions about avoiding the use of static classes and singleton; discussions about the change of the point of view when writing tests with expectations on how objects interacts; discussions about where to use of strict mocks and where instead to use stubs.

The practice of TDD with mocks significantly changed our production code and our test code. We observed and recognized that the result was better code easier to understand, change and evolve. Then we tried to understand which changes were causing the improvements, which changes were just side effects needed by unit tests and which changes were caused by our inexperience with TDD and mocks.

2.5 One year after the training on TDD with Mock Objects

Between 2007 and 2008 TDD with mocks become an established practice for the team. A group of team members were constantly discussing and striving to improve our practice of TDD with mocks, another group were keen on practicing correctly and systematically TDD with mocks and on adopting new improvements proposed by the first group, and finally another group were less interested about the practice still were supportive in maintaining existing tests and in practicing TDD with mocks when pairing with a team member experienced in that technique.

The difficulties with slow and fragile tests suits observed before the training were solved in the new and changed code and in unit tests written after the training. In addition the code written was easier to understand, change and evolve then the code written before, without TDD and mocks.

A group of team members striving to understand the relation between the changes caused to the code-base by the practice of TDD with mocks begun to study S.O.L.I.D. design principles and the Law of Demeter, discussed the relation between the practice of TDD and the adherence to design principles and reached a deeper practical understanding of the design principles and were able to further improve the code produced day by day intentionally removing more violations of the design principles then before.

2.6 Documentation of the experience

Between 2009 and 2011 this experience has been documented, reviewed by some team members involved, discussed and compared with other experiences, i.e. [21], to search for similarities and differences.

3. Hypothesis

As a result of these observations we were intrigued by the conjecture that code developed with TDD and mock objects tends to conform to some degree to the S.O.L.I.D. principles and to the Law of Demeter as an emergent property.

By emergent property we understood this to mean that the tendency to the conformance is obtained without an explicit policy to do so, without training the team on the design principles or without requiring the team to produce code that conforms to the principles. This means that an improved conformance is obtained as a positive unanticipated consequence of applying the practice of TDD with Mocks Objects [15].

The number of violations of the design principles can be measured every time a class is changed observing the code-base. Therefore the positive trend of this number of violations after the adoption of TDD with mocks can be verified. This is a hypothesis that can be verified with observations and also with code metrics.

Team members learned by observing positive effects of the changes in the code induced by the practice of TDD and mocks and this led to a deep practical understanding of the design principles and team members were able to further improve the code produced day by day intentionally removing more violations of the design principles. As a result of these observations we were intrigued by the conjecture that the tangible improvements of the code-base produced by the practice of TDD with mocks led the team to deeply understand the design principles and their practical applications as a result of a process of coevolution.

By coevolution we meant that the better understanding of the design principles and their practical applications in the code written is obtained as a result of the process of a mutual adaptation of the code-base and the team, where the positive change of the code-base is initiated by the adoption of the practice of TDD with mocks and the change in the team follows with a mutual adaptation process [16] [17][18].

The practical ability to avoid and remove violations of the design principles, even the ones that are not related to the adoption of TDD with mocks can be easily measured in the code-base and tested with exercises, before the adoption of TDD with mocks and after the adoption of the practice. The hypothesis of improvements of the practical ability to remove more violations can be verified with observations of the code-base and also with exercises.

4. Evaluation of the hypothesis

In order to evaluate the hypothesis, in addition to the evidence that the code was easier to understand, change and evolve, we evaluated the conformance of the code to the design principles by observation, sampling the code we were changing. We found that the code produced was more adherent to the S.O.L.I.D. design principles then before. And we found the code was only partially adherent to the Law of Demeter and this was compatible with what is reported in [8]. Indeed while every access to objects getter was usually wrapped to avoid "train-wreck", this had not removed all the violations of the Law as instead the delegation of behavior does.

Some team members discussed in retrospective about this experience and reported that they noticed that code developed with TDD using Mock Objects was easier to understand and change, they observed in the code characteristics that made it easier to read and evolve, they learned from these observations and they adapted they coding style to further pursue these useful characteristics. Some software engineers perceived commonalities in the source codes that were easier to understand and change and autonomously and voluntarily began to study the design principles and apply them in an aware and intentional way.

Then they found that what they reported was explainable with a well-known secondary effect of the emergent properties called coevolution.

5. Evaluation of the hypothesis in other teams and contexts

Discussing the observations of the experience of the team with other teams and experts helped to identify common misunderstanding and hidden assumptions that so need to be explicitly stated and described as preconditions in order to verify the hypothesis in different teams and in different contexts.

Also factors as people, the requirements, the technology used and environment where the development happen must be taken into account as possible relevant variables [19] in order to verify the hypothesis in different teams and in different contexts.

5.1 TDD with Mock Objects defined

TDD is generically described with the red-green-refactor cycle, how every phase is actually executed can substantially change from team to team, from programmer to programmer. The style of TDD with mocks referred here is the one originated in 1999 in the London-based software architecture group and then experimented and evolved in the Connextra team and later also in the London Extreme Tuesday Club (XTC).

It is the one described in the paper presented at the XP2000 conference [8] and the one presented at the OOPSLA 2004 conference [3] and is the unit testing approach described and explained in great detail in the GOOS book [4].

5.2 Properly trained developers

While here it is made the hypothesis about learning and developing a deep understanding of the design principles through a process of coevolution, the ability to practice effectively TDD with mocks is a given precondition. There is no claim here that the practice of TDD with mocks can survive inadequately trained developers.

5.3 The people and the environment

Since no one can be forced to learn a new technique, it is relevant that the people in the team have a purpose to learn TDD with mocks. In the team we had the tests suites that were slow and some of them brittle and we were striving to solve those issues. The environment was also a relevant variable. There was a high pressure to deliver new functionalities, a volume ten times bigger than the actual capacity of the team. Because of this, only the top priority functionalities were implemented and so they were used immediately after released. Because the short deadlines, we had to implement the features incrementally and so after the first release of a new feature the team usually had to reuse, change and extend the code just created or changed in the previous Sprint to extend the feature. And the deadlines were often of 1 or 2 weeks and less often of 3 weeks.

The relevant variables are:

- early feedback from the users: immediately after every feature is released, defects and bugs are reported;
- early feedback form the code: immediately after every feature is released, its code is often reused and changed and extended and this make it tangible how easy the code just written is easy to change and extend;
- very frequent releases: the feedback loops are really short and so the actions and the outcomes are under the same learning horizon enabling the team to learn from the experience.

5.4 No centralized point of control

The code-base was large including a large number of different integrated applications. And distinct autonomous interdependent departments were driving the evolution of the applications. The lack of a central point of control for the evolution of the system makes it clear that a centralized policy to evolve the design of the code could not be effective [20]. This encouraged the team to investigate other ways as emergent design driven by TDD and mocks.

6. The experiment

To better understand the phenomenon in general, between 2009 and 2011 an experiment was made: some developers outside the team voluntarily accepted to solve small coding exercises and answering to a survey. The exercises consisted in refactoring some code that had various violations of the S.O.L.I.D. principles and the LoD, with the goal of making the code testable and write the unit tests.

The survey's questions were about the proficiency of the developer in TDD with mocks, in TDD in general and in S.O.L.I.D. design principles. The solution of the exercises were compared with the level of proficiency declared in the survey and a conversation with the developer followed to clarify possible doubts. This experiment was conducted with an uncontrolled group and in an uncontrolled environment. The results were qualitatively measured.

The results of the experiment suggest that developers not proficient in TDD with mock, especially the ones that wrote integration or acceptance tests more then real unit tests, removed fewer violations of the design principles. Even the ones that claimed to be proficient in the S.O.L.I.D. design principles. Those developers proficient in TDD but not in TDD with mocks, that wrote real unit tests, removed more violations of the previous group. The group of developers proficient in TDD with mocks removed the major number of violations. Some of the

violations were not removed by any of the participants to the experiment.

7. Evaluation of the results

When discussing the conjecture that originated this study with other experienced software engineers and TDDers, a comment was that the skills and expertise required to design properly an application are vast and cannot be replaced just by applying TDD.

The preparation for this experiment made it very clear that the kind of the design improvement discussed here is the one that relate to the design of the classes, the distribution of responsibilities among different classes and how objects collaborate to each other sending messages at run-tine, and this is consistent with earlier research results [22]. The design at a more coarse grained level that focus on the organization of namespaces, components and sub-systems, domain models and on the definition of a compact and expressive language to implement features in that domain, is outside the scope of the hypothesis of this study, indeed is more related with Acceptance-TDD.

The results of the experiment showed that many developers proficient in the S.O.L.I.D. design principles and very capable of arguing and explaining the principles removed fewer violations of the ones practicing TDD with mocks.

A possible explanation is that the design principles are not specific to a language, a technology stack and a domain, so they are described in general and abstract terms. And the connection between the general abstract description and how to apply them in the code is not given. Because of this, the help TDD with mocks gives to remove violations and write code adherent to the principles make a huge difference.
This huge difference is evident and tangible and helps developers to make the connection between the general and abstract definitions and the practical applications in the code.

8. Analysis of the relation between TDD with Mock Objects, S.O.L.I.D. code and the Law of Demeter

This part analyzes how the team practiced TDD with Mock Objects and how this promotes the conformance to the design principle. TDD with Mock Objects defines [3] [4] [8] ways to write testable code, below these are labeled as Practice. For example it tells to pass dependencies in through the constructor. TDD with Mock Objects describes also a set of test code smells in the unit tests code that are related to possible problems in the design of the production code. They are labeled below as Smell. For example one smell is a bloated constructor. A list of possible solutions is suggested for every test smell

A practice explicitly describes what to do, while a smell requires to the developer a judgment based on knowledge and experience. Indeed a test code smell is a hint that something might be wrong somewhere in the code under test. It is not a certainty. It is up to the developer to check out the design of the code under test, and, based on his/her knowledge and experience, to decide whether the code actually need fixing, whether can be tolerated or whether is just OK as is [13].

8.1 Open-Closed Principle and Dependency Inversion Principle

The Open-Closed Principle (OCP) states that classes and methods should be open for extensions and strategically closed for modification: so that the behavior of a class can be changed and extended adding new code instead of changing existing code and many dependent classes.

The Dependency Inversion Principle (DIP) states that both low level classes (e.g. representing the persistence details or intra-systems communication details) and high level classes (e.g. representing application domain concepts or business transactions) should both depend on abstractions (e.g. interfaces): high level classes should not depend on low level classes. This improves the re-usability of classes and enables the evolution of the existing code with small local changes.

8.1.1 Practice

When writing a unit test with TDD using Mock Objects, a parameterized constructor is added to the class in order to inject all the dependencies, directly or through a factory that can return more than one instance of a dependency and permits to instantiate a dependency later in time. Look [3] at paragraph 4.9.

```
public class MonitoringSystemAlarm
{
public MonitoringSystemAlarm()
: this(new TirePressureSensor(), 17, 21) {}

public MonitoringSystemAlarm(
ISensor sensor,
double lowPressureTreshold,
double highPressureTreshold)
{
// ...
}
```

The point here is that all the dependencies implement their own interface and the interface type is used for the parameters in the constructor. The same holds true for dependencies that are passed as arguments of a method of the class. All this makes it possible to pass a mock object everywhere a real object is expected. This is not a work-around for a limitation of the mocking tool that cannot mock a concrete class, instead this is the deliberate way that TDD with Mock Objects adopts to break dependencies between classes, to make relationship explicit, to promote the coding of classes that are easy to reuse and that can be changed without provoking an unpredictable cascade of many changes. This is how TDD with Mock Objects helps to write classes that adhere to the DIP. Look [4] at chapter 20, paragraph "Mocking Concrete Classes".

8.1.2 Practice

Since with the practice of TDD with Mock Objects almost all the dependencies of a class are interfaces, all these dependencies give the possibility to create new implementations that extend the possible use of the class behavior. E.g. a logger class could log on different implementations of IAppender interface: file, console or db; a deposit class could work with different implementations of IOnlinePaymentsMethod: PayPal or Credit cards.

The interfaces and implementations are separate so it is possible to completely substitute anything at any point by providing another implementation of the interface. Moreover the use of interfaces prevents the use of public member variables (aka class fields), and singleton and static variables are discouraged because they are not unit test friendly and mock friendly. This help to write classes that adhere to the OCP.

8.1.3 Practice

The frequent refactoring during the red-green-refactor cycles of TDD with Mock Objects helps to remove conditionals (i.e. if and switch statements) and also the conditionals that check for object type (e.g. through C++ Run-Time Type Information or through Java and .NET Reflection). This too helps to write classes that adhere to the OCP.

8.1.4 Where TDD with Mock Objects doesn't help in the matter of OCP and DIP

A way to adhere to the OCP not directly enforced by TDD with Mock Objects: the use of the template method design pattern, call-back functions, events (publisher-subscribers design pattern) and policies as sorting criteria delegated to other classes.

A way to adhere to the DIP not directly enforced by TDD with Mock Objects: the use of the template method design pattern to encode a high level algorithm implementation in an abstract base class and have details implemented in derived classes. Thus, the class containing the details depends upon the class containing the abstraction. The same result can be obtained with the builder design pattern.

8.2 Single Responsibility Principle and the Interface Segregation Principle

The Single Responsibility Principle (SRP) states that there should never be more that one reason for a class to change: a class should have one and only one responsibility. The Interface Segregation Principle (ISP) states that clients should not be forced to depend upon interfaces that they don't use: fat interfaces should be avoided, while interfaces that serve only one scope should be preferred.

8.2.1 Smell

Writing a unit test with TDD using Mock Objects can lead the class under test having a bloated constructor: a constructor that has a long list of arguments used to inject dependencies. This is the smell that the class has too many responsibilities and one suggested refactoring is to break up the class into more classes each one with a single responsibility. Another suggested refactoring for this smell is to package a group of dependencies into a new class that contains them and deals with the related responsibility. Fore more details look [3] at paragraph 4.8 and [4] at chapter 20 the paragraph "Bloated Constructor".

8.2.2 Smell

A unit test with a lot of expectations is a smell that the class under test has more than one responsibility and the suggested refactoring is to extract into a new class a group of those collaborations declared in the expectations. [3] at paragraph 4.7 and paragraph 5.4 and [4] at chapter 20, paragraph "Too Many Expectations".

8.2.3 Smell

When a group of test cases uses the same group of member variables (aka class fields) of the text fixture class, this too is a smell that those test cases deal with a distinct responsibility and the suggested refactoring is to extract from the class under test the responsibility into a new class. For more details look [14].

8.2.4 Smell

When writing a unit test with TDD using Mock Objects it can happens to mock one method call of a dependency (e.g. set an expectation) and at the same time to stub another method call on the same dependency (e.g. set the return value for the method that could be invoked zero, one or many times).

```
[Test]
public void Send_Diagnostic_String_&_Receive_Status()
{
var mockTelem=mocks.StrictMock<ITelemetryClient>();
mockTelem.Stub(m => m.Connect());
mockTelem.Stub(m => m.OnlineStatus).Return(true);
mockTelem.Expect(m => m.Send(DiagnosticMessage));
//...
}
```

This is the smell that the dependency might have 2 distinct responsibilities. The suggested refactoring is to split the two responsibilities into two different classes.

8.2.5 About those smells

In all those cases, after breaking up the class, the result is new classes that adhere to the SRP. The class interface too is split into distinct interfaces that will adhere to the ISP [4] chapter 20, paragraph "Mocking Concrete Classes". The interfaces obtained with this process often mimic the implicit public interface of their class, so as a result you see pairs of things, like ITelemetryClient and TelemetryClient.

8.2.6 Practice

Another way to put too many responsibilities in a class is the abuse of inheritance. TDD with Mock Objects encourages the use of composition over inheritance and this prevents the abuse of inheritance and also the violation of the SRP caused by the abuse of the inheritance. For an example look at [3] paragraphs 2.1, 3.3.1 and 3.7.

8.2.7 Where TDD with Mock Objects doesn't help in the matter of ISP

A way to adhere to the ISP not directly enforced by TDD with Mock Objects: even when an interface mimic the implicit public interface of a class that already has a single responsibility, sometimes there can be chances to further break up the interface into distinct interfaces aimed at different clients, with the goal of eliminating an inadvertent coupling between clients and between DLLs. This decreases the number of dependencies and the number of recompiles needed after a change. And the result is a better conformance with the ISP.

8.3 Liskov Substitution Principle

The Liskov Substitution Principle (LSP) states that methods that use pointers or references to a base class must be able to use instances of derived classes without knowing it: all the derived classes must honor the contract defined by the base class.

8.3.1 Practice

A method implementation that checks for an object type of the actual argument (e.g. through C++ Run-Time Type Information or through Java and .NET Reflection) violates the LSP as well as the OCP. With the practice of TDD with Mock Objects, the bar become red when changing the method parameter type from the base class type to the interface type in order to mock the argument. The LSP violation is surfaced by the failing test, and to get a green bar the violation must be removed.

8.3.2 Practice

TDD with Mock Objects and TDD in general change the design of base and derived classes from a process of invention into a process of discovery: first commonalities among different classes are found and then are extracted in a common base class. The commonalities are found after the test is green (red-green) and the duplication is removed refactoring the code (green-refactoring). This prevents many violations of the LSP that can happen when a base class is designed upfront or when classes are derived upfront. Furthermore, TDD with Mock Objects promotes the use of composition over inheritance. For more details and examples look [3] at paragraph 2.1, 3.3.1 and 3.7. This avoids many violations of the LSP too.

8.3.3 Practice

Furthermore, a derived class that overrides a virtual method violates the LSP when it replaces the precondition of the base class method with a stronger one and when it replaces the post condition with a weaker one. This violation can be detected executing the unit tests of the base class also against the derived class. This holds true for TDD and for unit testing in general.

8.3.4 Where TDD with Mock Objects doesn't help in the matter of LSP

All the previous practices prevent or avoid violations of the LSP. Adherence to the LSP is easier to verify in the context of its clients using the base class and the derived classes. The LSP makes clear that in OOD the ISA relationship pertains to extrinsic public behaviors that clients depend upon. The main focus when writing a unit test with TDD using Mock Objects is on the behavior on the Design by Contract, in this case the behavior of the method that is overwritten in the derived class. Look [3] at paragraph 2.1. When there is a violation of the LSP it can be highlighted by some unit tests e.g. when the expectations on the same interface methods on two different tests are inconsistent. It is up to the programmer to notice the inconsistency and finding how to fix the LSP violation. It is also up to the programmer to spot the refused bequest smell and to fix it when appropriate.

8.4 Law of Demeter

The Law of Demeter (LOD) states that methods of an object should avoid invoking methods of an object returned by another object method, the motto of LoD is "Only talk to your friends" and the goal is to promote loose coupling.

8.4.1 Practice

Avoid the use of getters; replace them with Smart Handlers that are Visitor-like objects [6] that are passed to the object without getters. With this practice code tends to conform to the LoD just like when applying the Tell, Don’t Ask principle. For an example look [8] at paragraph 4.3.

8.4.2 Smell

A single modification in the code that requires changes to expectations in two different tests is a smell that design is breaking the Law Of Demeter. This is true especially when the initial modification in the code involves getters. The suggested refactoring is to replace getters, with Smart Handlers. For an example look [8] at paragraph 4.3.

8.4.3 Smell

Also a unit test with a lot of expectations with mocks that return other mocks is a smell that the class under test has a responsibility that belongs to another object and the suggested refactoring is to apply the heuristic "Tell, Don't Ask". Fore mode details look [3] at paragraph 1.2, and [4] at chapter 2 paragraph "Tell, don't ask" and at chapter 20 paragraph "What the Tests Will Tell Us (If We’re Listening)" and also [5].

9. Findings

The results of observations, experiment and analysis are compatible with the two initial conjectures and lead to identify the preconditions, the relevant variables and the hypothesis that can be tested. The precondition is that the developers must be properly trained in the practice of TDD with Mock Objects and able to apply it properly as described in [3][4][8].

Relevant variables of the environment, within the software team operates, are:

- early feedback from the users about defects and bugs in the new releases
- early feedback form the code: features are developed and release incrementally so the code just released is immediately reused and changed and extended;
- very frequent releases: every week or two in order to have very frequent feedback that enable learning from the practice

Another relevant variable is the pressure and the will to release working and valuable software as fast as possible, and the presence of mentors for the practice of TDD with Mock Objects to support safe experimentations and improvements.

The first hypothesis: number of design principles violations in the code-base decrease faster when classes are changed by a team properly practicing TDD with mocks. The second hypothesis: after practicing TDD with mocks, also the number of violations of the design principles not directly related to the adoption of TDD with mocks (for example the ones describe in the paragraph 8.1.4) decrease progressively more.

The result should be different from team members that don’t practice TDD or practice TDD improperly writing using tests that are more like integration tests. In that case the number of violations in the code-base is not expected to decrease as much as for the team doing TDD with mocks.

10. Discussion

The analysis documented here about the relation between the practice of TDD with mocks and design principles is useful to evaluate the conjecture that improved conformance to the design principles is an emergent property. Indeed the Practices as described in the analysis show that some of the violations of the principles are removed as direct consequence of those practices. This cause-effect relationship does not indicate an emergent behavior even if this is a positive unanticipated consequence. So we should name this a weak emergence.

At the same time the Smells described in the analysis don't have a direct relation with removing violations of a design principle, it is the result of a judgment based on knowledge and experience of the developer that is developed practicing TDD with mocks. We can call this proper emergence. The coevolution used to explain the process of learning the design principles and their practical applications when practicing TDD with mocks as well as the emergence used to explain the improved conformance they both arise during the process of self-organization in a complex system. And since team members are human beings, it is a socially complex system [10][19].

Joseph Pelrine is one of Europe's leading experts on Agile software development. He has worked as assistant to Kent Beck in developing eXtreme Programming, is an accredited practitioner for the Cognitive Edge Network, and his work focus is on the field of social complexity science and its application to Agile processes. He suggested the use of the ABIDE model (Attractors, Barriers, Identity, Dissent/diversity and Environment) developed by Dave Snowden at the Cynefin Center for Organisational Complexity and now at Cognitive Edge [11] to search for relevant parameters of the socially complex system. In particular he suggested that the two software engineers extremely experienced in the practice of TDD with Mock Objects that trained the team and then joined the team acted as Attractors in the process of self-organization of the socially complex system.

Following the ABIDE model, the practices or TDD with mocks acted as Barriers in the self-organization. While the frequent feedback from users and the code that define a structure of the interaction between team members and the users and the code contributed to define the Environment where the self-organization had place. This is consistent with research results about iteration and learning [23].

The conjecture reported here, that the process of learning is emergent phenomenon, has been studied before also by Dr. Sugata Mitra. Dr. Sugata Mitra, Education scientist, professor of Educational Technology at New Castle University UK and Chief Scientist of NIIT since 1999 with his 'Hole In The Wall' experiments is testing his speculations about education as a self-organizing system where learning is an emergent phenomenon [12]. Sphere College in Phoenixville Pennsylvania, and Khabele School in Austin Texas have an educational philosophy that incorporates elements of self-organization and emergent education.

We present here some comments and quote form experts in TDD and in TDD with mocks that are relevant to this study. A relevant quote from Steve Freeman: *No technique can survive inadequately trained developers*. A relevant quote from Nat Pryce: *TDD does not drive towards good design, it drives away from a bad design. If you know what good design is, the result is a better design*. A relevant quote from Kent Back: *TDD doesn't drive good design. TDD gives you immediate feedback about what is likely to be bad design*. A relevant quote from Michael Feathers: *writing tests is another way to look the code and locally understand it and reuse it, and that is the same goal of good OO design. This is the reason of the deep synergy between testability and good design.*

11. Threats to validity

Since this is an observational study based on observations in an uncontrolled experiment, it is not free from overt biases as in the sampling of code observed and in the judgment of the code observed in regard to adherence to design principles. There is also the possibility of hidden biases as the lot of tacit knowledge of good design by the observed team. Since observations have been documented in retrospective, they potentially suffer from the Texas sharpshooter fallacy.

12. Conclusions

The observations, the analysis of the relation between TDD with mocks and the design principles and the qualitative experiment are compatible with the conjecture that the practice of TDD with Mocks Objects led the team to write code more conformant to the S.O.L.I.D. design principles and partially to the Law of Demeter. They are compatible also with the conjecture that the practice of TDD with Mocks Objects led the team to learn and develop a deep understanding of the design principles and their practical applications. And finally they are compatible with the conjecture that conformance to the design principle is an emergent property and learning design principles is a process of coevolution.

The qualitative experiment and the analysis of the relation between TDD with mocks permitted to roughly quantify the expected improvement of conformance to design principles due to the practice of TDD with mocks. Information and understanding developed with this study permitted to identify preconditions and relevant variables and to turn the conjectures into hypothesis that can be tested in a subsequent empirical software engineering research.

13. Acknowledgments

Thanks to Paolo Polce and Gerardo Bascianelli that joined the team and shared their knowledge and deep experience on TDD with Mock Objects. Thanks to Antonio Carpentieri and Riccardo Marotti and all the dev team members of the F1 Racing Team for their curiosity to explore new ways of writing code, for their courage to give up old skills for new ones, for their trust and respect that permitted us to engage in discussions, open disagreement and coding experiments and come out with new useful understanding and insights. Thanks to those members of the XPUG-IT and UGIdotNET Italian community that voluntarily participated in the experiment. Thanks to those who helped to review this paper, suggested ideas and improvements and shared and discussed their own experiences with TDD.

References

1. Beck, K. 2002. Test Driven Development: By Example, Addison Wesley
2. Feathers, M. 2004. Working Effectively with Legacy Code, Prentice-Hall
3. Freeman, S., Mackinnon, T., Pryce, N. & Walnes, J. 2004 . Mock roles, not objects. In OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, 236-246. Available also from: http://planningcards.com/site/articles/papers/
4. Freeman, S., Pryce , N. 2010. Growing Object-Oriented Software Guided by Tests, Addison-Wesley
5. Hunt, A. and Thomas, D. 1998. Tell, Don’t Ask
6. Gamma, E., Helm, R., Johnson, R. & Vlissides, J. M. 1994. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional
7. Lieberherr, K. and Holland, I. Assuring Good Style for Object-Oriented Programs IEEE Software, September 1989, 38-48.
8. Mackinnon, T., Freeman, S., Craig, P. Endo-testing: unit testing with mock objects. In Extreme Programming Examined, Addison-Wesley, Boston, MA. 2001. 287-301
9. Martin, R. C. 2002. Agile Software Development, Principles, Patterns, and Practices, Prentice-Hall

10. Arrow, H., McGrath, J. E. & Berdahl, J. L. 2000. Small Groups as Complex Systems: Formation, Coordination, Development, and Adaptation, Sage Publications
11. Cognitive Edge: http://www.cognitive-edge.com/
12. Mitra, S. Self organising systems for mass computer literacy: Findings from the 'Hole in the Wall' experiments, In International Journal of Development Issues, 4(1), pp 71-81 (2005).
13. Beck, K., Fowler, M. 1999. Refactoring: Improving the Design of Existing Code, Chapter 3 Bad smells in code, Addison-Wesley, See also http://en.wikipedia.org/wiki/Code_smell
14. Feathers, M. 2004. Working Effectively with Legacy Code, pp 251 Heuristic #4: Look for internal relationship, Prentice Hall
15. Goldstein, J. Emergence as a Construct: History and Issues, In Emergence: Complexity and Organization 1 (1): 49–72 (1999)
16. Kauffman, S. A. 1993. The origin of order: self-organization and selection in evolution, Oxford University Press
17. Kauffman, S. A. 1995. At Home in the Universe: The Search for Laws of Self-Organization and Complexity, Oxford University Press
18. Goerner, S. 1994. Chaos and the evolving ecological universe, Langhorne PA: Gordon & Breach
19. Pelrine, J. On Understanding Software Agility - A Social Complexity Point Of View, In E:CO Issue Vol. 13 Nos.1-2 2002 pp 26-37
20. Forrest S. Balthrop J. Glickman M. Ackley D. Computation in the wild. E. Jen, editor, Robust Design: A Repertoire of Biological, Ecological, and Engineering Case Studies, pages 207–230. Oxford University Press, 2004. Reprinted in K. Park and W. Willinger Eds. The Internet as a Large-Scale Complex System, pp. 227-250. Oxford University Press (2005).
21. Madeyski, L. 2010. Test-Driven Development: An Empirical Evaluation of Agile Practice, Springer.
22. Madeyski, L. 2006. The Impact of Pair Programming and Test-Driven Development on Package Dependencies in Object-Oriented Design - An Experiment. Lecture Notes in Computer Science, 4034:278-289, Springer
23. Taylor, K. , Rohrer, D. 2010. The effects of interleaved practice, http://www.researchgate.net/publication/227530785_The_effects_of_interleaved_practice/file/5046351d5cf029c602.pdf

Further reading

Refactoring legacy code driven by tests - Coding Dojo
https://github.com/lucaminudel/TDDwithMockObjectsAndDesignPrinciples/tree/master/TDDMicroExercises#readme

Emily Bache, 'The Coding Dojo Handbook', https://leanpub.com/codingdojohandbook

Paper reference: https://github.com/lucaminudel/TDDwithMockObjectsAndDesignPrinciples
This article was first presented in 2011 and is published here with permission of Luca Minudel

BDDfire: Instant Ruby Cucumber Framework

Shashikant Jagtap, @Shashikant86, http://shashikantjagtap.net/

BDDfire is a Ruby library that creates skeleton for the ruby Cucumber Behaviour Driven Development (BDD) framework. Cucumber is very popular BDD framework. Cucumber can be more awesome if we use it with the right tools. BDDfire supports Selenium, PhantonJS, Appium, Saucelabs, Browserstack, Testingbot, Relish, Cuke_Sniffer and many more open source libraries. BDDfire aims to integrate all the friends of Cucumber and make the Cucumber framework stronger.

Website: https://rubygems.org/gems/bddfire
Version: 1.6.0
System Requirement: Ruby 1.9.3 +, NodeJS
License & Pricing: Open Source, MIT License,
Support: Website

BDDfire is an open source tool built around the Cucumber BDD framework for Ruby which supports various popular open-source libraries like Capybara, Selenium-WebDriver, Poltergeist, Relish, Cuke_Sniffer, Rubocop, Appium, Saucelabs, Browserstack and many more. BDDfire will create all the directories and required files to support the latest open-source libraries in the Cucumber framework.

BDDfire Features

Using BDDfire brings many benefits:

- BDDfire will create template directories and files for the Cucumber project.
- Integration with Capybara for executing acceptance scenarios in real and headless browsers.
- Generation of template directories to support RSpec
- BDDfire supports Cuke_Sniffer and Rubocop libraries that detect smell in Cucumber.
- BDDfire supports Relish for the living documentation. Rakefile has been automatically created as a part of BDDfire
- BDDfire supports YARD documentation of the Cucumber project.
- BDDfire allow you write steps using the Page Object Pattern
- BDDfire can run scenarios in the different viewport by specifying screen resolution.

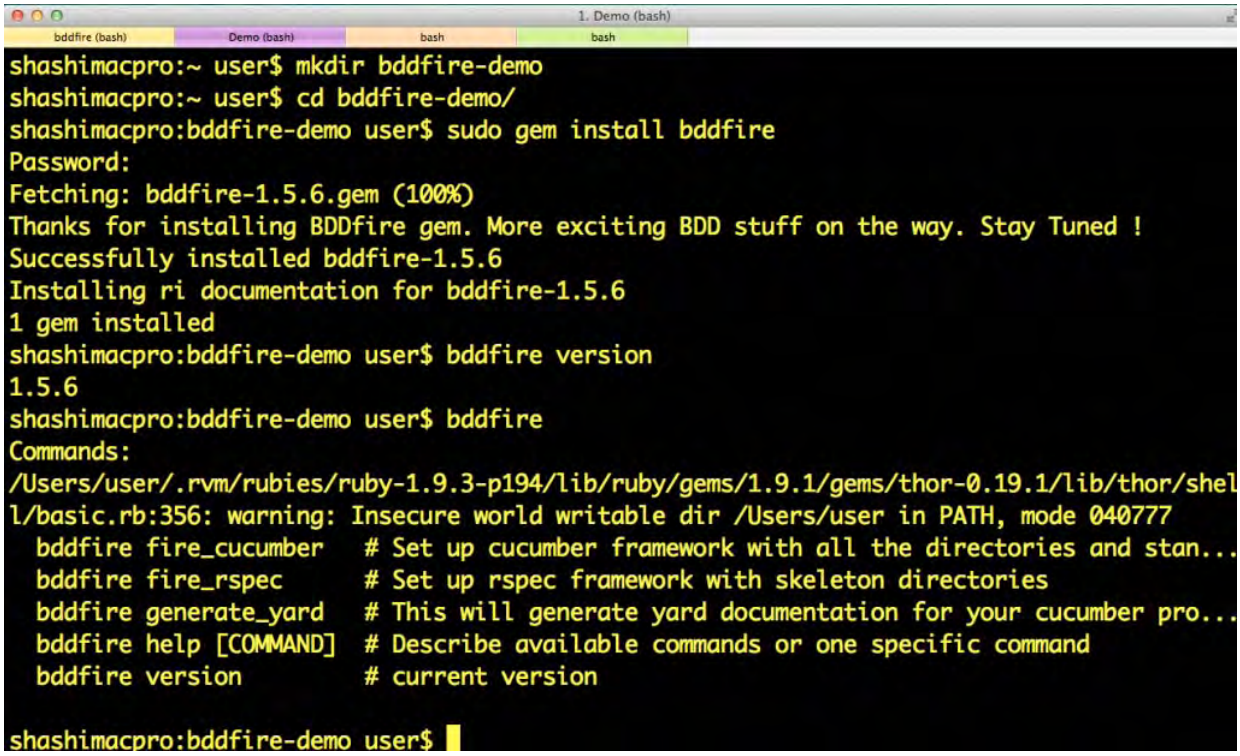
BDDfire Installation and Usage

BDDfire is a RubyGem so you can install it by using a simple command.
$ gem install bddfire

If you already have Gemfile, then just add ‘ bddfire’ in your Gemfile. Once, installed you can see all bddfire options by using command bddfire
$ bddfire

It will display the following options :
Commands:
bddfire fire_cucumber
bddfire fire_rspec
bddfire generate_yard
bddfire help [COMMAND]
bddfire version

This will look like this in the terminal .

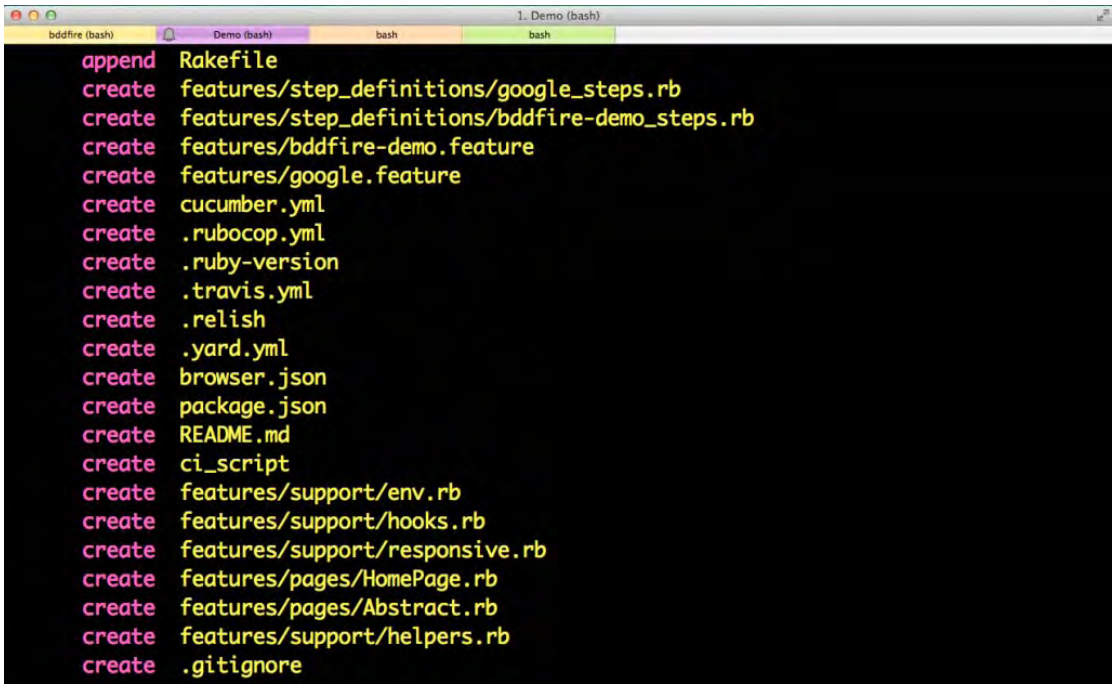


Cucumber Framework for Ruby

BDDfire can generate scaffolding for Cucumber by creating the necessary directories and files with relevant code in it. BDDfire has an option ‘fire_cucumber’ which does it all for us
$ bddfire fire_cucumber

Once you run this command, you will have the entire BDD Ruby Cucumber framework setup within a minute. This will create Gemfile with all the required RubyGems and Makefile to execute features with different profiles.

The output will look like this:



Now we have a new 'Gemfile' in our project. We need to install all the dependencies with ruby bundler:
$ bundle install

Now that we have all the skeleton directory structure for the Cucumber project, let's have a look what we got inside the directories and files.

Real and Headless Browser Support

Automated cucumber scenarios can be run with a real browser or with headless browser. BDDfire gives you the option to run them in both. Selenium WebDriver can be used to run automated Cucumber scenarios in real browsers without any additional setup. Selenium driver for Capybara is setup for you with a default Cucumber profile. In the cucumber.yml file, you can see that Selenium WebDriver is setup as default capybara driver. We can run our scenarios in the Firefox browser like this:
$ bundle exec cucumber

In a similar way, we can run the scenarios in headless browsers like PhantomJS using Poltergeist as Capybara driver. The Poltergeist driver is already setup for you in 'features/support/env.rb' file. BDDfire also created a profile to run scenarios with headless browser.
$ bundle exec cucumber -p poltergeist

You can see your scenarios executed but you won't see any browser launched, this helps to run automated Cucumber scenarios faster than with real browsers.

Responsive Test Automation

BDDfire allows automated test scenarios in the different viewport and screen resolutions. BDDfire creates a file 'features/support/responsive.rb' which is the config for the running scenarios with Selenium and Poltergeist driver.

The supported screen sizes are 320, 600, 770 and 1026, but you can change this if you need. In order to run scenarios with different screen resolution, we need to set the environment variable DEVICE.
$ DEVICE=320 bundle exec cucumber

This will run automated scenario with Selenium driver and with screen size of 320. We can execute the same with the Poltergeist driver.
$ DEVICE=320 bundle exec cucumber -p poltergeist

Mobile Test Automation and Cloud Testing

BDDfire has a built-in Appium driver setup that allows you to run your automated Cucumber scenarios in mobile devices. Appium is an open source test automation framework for the mobile applications. Using the Appium driver, you can run tests in the Android and iOS devices. In order to run them in the real devices, you need to do the initial ADB setup for Android and UUID setup for iOS. This part is covered in the Appium documentation. Android ADB provides the ADB serial number for the devices attached to the computer. You can use this serial number to run automated cucumber scenarios in the mobile devices.

You need to run npm packages to install Appium locally.
$ npm install

This will install the npm dependencies and you can launch the Appium server
$ ./node_modules/.bin/appium

Now you can execute automated Cucumber scenarios in the attached Android device
$ ADB_SERIAL = XXXXX bundle exec cucumber -p appium

In a similar way, BDDfire gives you an option to run tests in third party cloud testing services like SauceLabs, BrowserStack and TestingBot. In order to use those services, you need to create an account and get an username and assess key. You can then use that access key in ‘features/support/env.rb’ file to run the automated Cucumber scenarios. If you are running tests in the BrowserStack, you will have all the stacks defined in the 'browser.json' file. We can use any of them. We need to configure BrowserStack username and access key in the driver configuration.
$ BS_STACK=osx_firefox bundle exec cucumber -p browserstack

Using the same setup, you can run automated scenarios in the other third party testing services like SauceLabs and TestingBot.

Living Documentation

BDDfire gives you an option to use Relish as living documentation tool. Relish publishes feature files online so that the whole team can read and access them. In order to use Relish, you need to have a Relish account and API TOKEN that can be obtained by visiting 'https://www.relishapp.com/api/token' after logging in.

BDDfire creates a file ‘.relish’ where you need to paste your token. Now you can publish your feature files online by using
$ bundle exec relish push {Publisher}/{project}

Code Quality

BDDfire comes with two code quality checking tools : Rubocop to check Ruby code quality and Cuke_Sniffer to detect smells in the Cucumber features, step definitions and hooks. BDDfire creates a '.rubocop.yml' file with basic Ruby code quality rules. You can change the configuration according to your requirements. With this configuration, you can use Rubocop to detect smell in our Ruby code
$ bundle exec rubocop

In similar way, you can use Cuke_Sniffer to detect smell in the Cucumber project. Cuke_Sniffer runs through all feature files. step definitions, hooks and support code. It finds out dead steps and suggests improvement to write better feature files. You can execute Cuke_Sniffer like this:
cd features
$ bundle exec cuke_sniffer

Continuous Integration

BDDfire creates a 'ci_script' file to run automated Cucumber scenarios on continuous integration server like Hudson and Jenkins. It will clear the workspace by deleting old reports, creates YARD documentation for your project and runs Rubocop and Cuke_Sniffer before executing your scenarios. You can also change this file to suit your project need. BDDfire also creates a 'travis.yml' file to support execution on TravisCI.

Page Object Pattern

BDDfire allows writing steps using the Page Object Pattern. Page object pattern is a great way to maintain the automation framework by abstracting locators and common methods to the Page classes. BDDfire creates a directory called "features/pages" that has 'Abstract.rb' and 'HomePage.rb" files. You can create instance of these classes at any point in the steps_definitions like this :
@home_page = HomePage.new(Capybara.current_session)
@home_page.visit_home_page

This makes your step_definitions more stable. You need to change page classes if something changed and step_definitions remains unchanged.

Rspec Framework

BDDfire also creates skeleton directories for the Rspec project as well. It will create necessary directories and files with relevant code in it. Just run
$ bddfire fire_rspec

Summary

BDDfire can add value to the Cucumber BDD framework for ruby by integrating awesome open-source libraries with it. BDDfire can help companies by setting up their Cucumber framework and making it extensible.

Further reading

http://shashikantjagtap.net/category/bddfire/
GitHub: https://github.com/Shashikant86/bddfire

Mobile Dev + Test Conference, April 12–17, 2015, San Diego, CA
The inaugural Mobile Dev + Test Conference addresses mobile development for iOS and Android, mobile testing, performance, design, user experience, smart technology, and security. Hear from experts in the field about where the future of smart and mobile software is headed. With 50+ learning opportunities, including sessions about the most popular topics in smart and mobile technology, there will be lots of tips and tricks to take back to the office to better your processes and increase your value.

Register by February 13 2015 and save up to $400. Go to https://well.tc/Ycc

---

Advertising for a new Web development tool? Looking to recruit software developers? Promoting a conference or a book? Organizing software development training? This classified section is waiting for you at the price of US $ 30 each line. Reach more than 50'000 web-savvy software developers and project managers worldwide with a classified advertisement in Methods & Tools. Without counting the 1000s that download the issue each month without being registered and the 60'000 visitors/month of our web sites!

To advertise in this section or to place a page ad simply http://www.methodsandtools.com/advertise.php

---

METHODS & TOOLS is published by **Martinig & Associates,** Rue des Marronniers 25,
CH-1800 Vevey, Switzerland Tel. +41 21 922 13 00 Fax +41 21 921 23 53 www.martinig.ch
Editor: Franco Martinig ISSN 1661-402X
Free subscription on : http://www.methodsandtools.com/forms/submt.php