METHODS & TOOLS

.

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

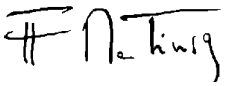
Spring 2015 (Volume 23 - number 1) www.methodsandtools.com

From Software Delivery to Software Creativity

This editorial was inspired by a quote from Mary and Tom Poppendieck book "Lean Mindset". They wrote "What's next is to stop thinking about software development as a delivery process and to start thinking of it as a problem-solving process, a creative process."

In many large companies, software development has often been traditionally considered as a "production unit" that will translate business requirements into code. There is an important amount of research and practice around the concept of "software factory", especially in the industrial domain. In recent years, there has been an increased number of situations where the software has become the product or the main way that customer will use for their shopping experience. When you use an hotel booking app or a social networking web site, you are looking for an hotel room or the capability to connect with other people, but one of the key factor to choose between competing options will be the ease of interaction provided by a mobile app or a web site. Creativity has always been present in the software development world. In a pre-Internet world however, the costs to distribute and market physically packaged software was an important barrier to put new products in front of the potential customers. Today with app stores and the acceptance of relying on hosted subscription based solutions, the relationship between the software creator and his customer has become more direct. Easier distribution doesn't necessarily translate into more successes, but it certainly provides more variety and competition in software markets. It seems to me that I discover a new agile project management solution every month.

If this improved context for software creativity has led to the multiplication of software-focused startups, the situation is different in the corporate world. The transition to accepting software developers as partners and co-creators can be slow as they are still considered as suppliers and costs that should be minimized. As in Agile initiatives, the abandon of a command & control vision and the nurturing of a trust culture are not obvious to achieve in traditional management structures.



Inside

Impact-Driven Scrum Delivery

Ingrid Domingues, @ingriddomingues
InUse, http://www.inuse.se/

Impact-Driven Scrum Delivery brings together Scrum's capacity to deliver working software, with the ability of Impact Management to define actionable metrics, and to evaluate outcomes well before the production of working software. The idea of "outcome over output" [1], lately emphasised in the Agile community [2], can now be realised in all types of projects – not only those where it is feasible to do the measuring by releasing the proposed new solution to a small percentage of real-time users. Finally, Impact-Driven Scrum Delivery solves "the Product Owner's dilemma" [3], and makes the management ideal of "pivot or persevere" [4] an inherent capability of the process.

Impact-Driven Scrum Delivery starts with the insight that, from a business perspective, design matters a lot. This doesn't mean that a bad business idea, or badly-understood user needs, could be covered up with design, but rather that a good business idea and understanding of user needs will suffer from *bad* design. I mean "design" as in "designing the digital solution", which involves everything from visuals, interaction, content, response time, metadata, performance, to maintainability and all back-end capabilities.

First, Impact-Driven Scrum Delivery addresses solutions with user interfaces. Then we state that – in order for the business to gain what is expected from the investment in this digital solution – the user must succeed. Therefore, the delivery process starts and ends with the user's activities. Understanding needs and behaviours, defining actionable metrics [5], designing user interfaces that support and satisfy users and measuring the activities that indicate business success: these are the essence of this process.

Impact Management is a framework that helps managers to keep focus on impact from idea to continuous improvement. Impact Management originated from a paper [6], *From Business to Buttons*, which defined the concept of Impact Mapping: the ability to define expected impact for business and users in a model, in such a way that it can be evaluated and measured at any specified time. Impact Mapping evolved into the framework of Impact Management [7], which gives business and project managers the ability to maintain and act upon an impact focus in their software development program or project.

The Impact Map describes the business and user values that a new product or service is expected to generate. It has a straightforward structure, similar to Simon Sinek's "golden circle" [8], describing

- *why* this solution is a good investment for the business
- *how* people are going to gain from it
- *what* the solution should encompass in order to promote user satisfaction and business prosperity.

Impact Mapping and Management caught on fast and are today widely used within software development. Impact Mapping crossed over into Agile software delivery at the start of the decade. [9] An Impact Map can be used to formulate an idea without doing a lot of analysis. But if the Impact Map is to be used for Impact Management it needs to be grounded in analysis that can explain the causality between user needs and business impact. Such Impact Maps are furnished with clear description and priority of user needs and user behaviours, as well as actionable descriptions of business and user needs, which can be employed for testing solutions at any specified time. The need for established user studies and analysis of business opportunities increases in proportion to complexity, such as projects with many stakeholders, more severe risks of being wrong and less freedom for continuous testing [10].

When Impact Management meets Scrum, Impact-Driven Scrum Delivery is born. This solves the Product Owner's dilemma, enhances the communication from business to the team, and makes teams deliver outcome over output. Impact-Driven Scrum Delivery provides a strong framework for continuous communication between two perspectives: business and delivery. The business side is responsible for the outcome, ensuring we build the right thing, whereas the delivery side is responsible for building the solution in the right way. Scrum provides a perfect communication platform where these two perspectives can meet. Impact Management gives a clear purpose and means for measuring success. Impact-Driven Scrum Delivery marries the two, and helps both perspectives perform better, with autonomy, mastery and purpose [11].

Impact-Driven Scrum Delivery solves the Product Owner's dilemma. Product Owners understand that design matters and that user interface design is crucial in order to meet expected business benefits from the digital investment. But very few Product Owners have the skills necessary for user interface design, neither on a concept level nor on a detail level. At the same time, the Product Owner has the responsibility to deliver value to business with the product that is built and shipped. Usually there is also more than one business stakeholder for any product.

This means that the Product Owner has to spend time (sometimes a lot of time) to explain the solution, negotiate scope, capabilities and outcome with other stakeholders within the company.

Add to this the team's need to get instant answers to detail questions. Thus, the Product Owner's dilemma is that he or she lacks the design skills needed to really provide the support the team needs. Product Owner needs to spend his or her time in communicating with the other business stakeholders to ensure the right business decisions are made, and to prepare all involved parties for action when needed.

Impact-Driven Scrum Delivery strengthens the business side of a Scrum project with a User Experience (UX) Lead. A UX Lead is an experienced UX designer, someone who has worked with designing different products for some years, and has therefore built a genuine understanding of the way design can strengthen business. He or she has a broad set of tools for understanding user needs, testing outcome, visualising flows and customer experiences, defining MVPs [12] and doing user testing. A UX Lead also has a profound understanding of how different user interface designs affect data and coding, how business rules must be defined to boost user experience, and so forth.

Perspective	**BUSINESS**	**DELIVERY**
Responsible	Product Owner	Scrum Master
Supporting	UX Lead	Team
Focus	Building the right thing	Building in the right way
Action	Thinking far ahead	Producing executable code

The UX Lead is the Product Owner proxy, giving the power to the Product Owner to fulfil his or her business responsibility, and leaving the UX Lead to manage the day-to-day Product Owner work. UX Lead and Product Owner start their work with defining expected business impact, understanding user needs, and designing and testing – from a business perspective – the most important flows.

This starts before the Scrum process, and by terming it "sprint minus 1" we remind people that it is a necessary antecedent to "sprint 0". Exploring business opportunities and needs, understanding user behaviours and defining actionable metrics can be swift, but more usually it takes time: not only making the discoveries, but also understanding the business implications of all the opportunities revealed, and agreeing which alternatives to move on with. This is, of course, more difficult in more consensus-focused cultures, and easier in cultures where decision-making is simple.

During "sprint minus 1" the following happens:

- The UX Lead works with the Product Owner to define expected impact, by analysing user needs and business opportunities, and boils these down to actionable user goals and business impact. This results in an Impact Map where there is a causal relation between user activities/success and expected business impact. In charting this causality the Impact Map also defines what the user needs in order to feel satisfied, touching on the capabilities or functions that will satisfy their needs, and ways of evaluating their satisfaction.
- The UX Lead also works in a structured "design thinking" way to find patterns for interaction, form and content that have been verified in a working prototype. "Design thinking" in this sense means work structured with divergence and convergence [13]. If possible, simple user tests are performed, and the goal is to verify assumptions about how the expected impact for user and business will arise. The design process strengthens the quality of the Impact Map, detailing how to measure success. Having a solid foundation for user interface architecture also gives excellent ground for detailing user stories with acceptance criteria, such as user interface details including function, form and content
- When the project is a matter of enhancing something that already exists in working systems architecture, the prototype will be treated either as an MVP in "sprint minus 1" or it will be the first shippable increment
- In projects with very limited budget the prototype can be very rough. The point is still to explore the best way to make users succeed with the most important user flows, which could sometimes be tested with simple PowerPoint prototypes
- Equipped with such a well-grounded Impact Map, the Product Owner, UX Lead and a systems architect create the first suggestion for roadmap and product backlog. The stories in the backlog will be tagged with user, and user goals, among more standard labels such as epic, increment etc.

Here is where the so-called "sprint 0" starts, where the Scrum team and process are set. When using Impact-Driven Scrum Delivery, some amendments will need to be agreed upon:

- The UX Lead is responsible for any user-interface design-related questions. He or she will be available to the team all the time, and will respond instantly
- The UX Lead has direct and daily contact with the Product Owner in order to discuss any business-related issue. The Product Owner has full responsibility for business impact and anything that can affect the business outcome. The Product Owner therefore owns the product backlog, and is the only role authorised to perform prioritisation. Usually he or she delegates the operational maintenance of the backlog to the UX Lead
- The UX Lead will prepare user stories well ahead in the backlog, to be discussed at backlog refinement meetings, held between sprints. The stories emerge from the Impact Map, refined and described with a user-interface design and user acceptance criteria. In the discussion, the team will add insights of opportunities and obstacles. Sometimes one story needs to be discussed in more than one backlog refinement meeting, in order to make user needs, good user interface and service design meet what is a good way of coding and developing the solution. The stories need care and consideration from the team, so that all can agree that the suggested solution is feasible and practical to build and maintain.
- The UX Lead is thus responsible for ensuring that all stories that reach sprint planning are well-worded and prepared. The UX Lead is responsible for refining the story when it is discussed in the sprint planning meeting and checking that acceptance criteria also meet the needs of the user interface. The UX Lead is responsible for including a visual description of any design details not laid out in the prototype

- In the "definition of done", an interaction review is added for every story with a user interface. This means that the UX Lead should be consulted before marking the story as "done". This will radically reduce the UX guilt for the team
- The UX Lead is responsible for evaluating, and validating, the extent to which shippable increments meet user needs, and create expected impact. This can be accomplished in different ways: sometimes it is appropriate to do testing in sprints, but more often user testing is treated as a separate swimlane. It is a good idea to integrate user testing into the team work in an orderly way, so that team members have time to participate in some test occasions.

Our experience of this way of working is extremely promising. It gives the team the potential to deliver at its best. The team knows that its priorities are clear. They are given full support in any design issue and they know that shippable increments have undergone user testing. This results in high-quality solutions, it solves the Product Owner's dilemma, and it delivers software that provides the expected impact for business and users.

References

1. https://hbr.org/2012/11/its-not-just-semantics-managing-outcomes
2. Eg in Jeff Patton's latest book, *User Story Mapping*
3. Lightning talk about the Product Owner's dilemma http://www.slideshare.net/inusese/the-product-owners-dilemma-and-how-to-solve-it-2012-0417
4. Eric Reis, Lean startup http://www.entrepreneur.com/article/220302
5. http://fourhourworkweek.com/2009/05/19/vanity-metrics-vs-actionable-metrics/
6. https://www.designsociety.org/publication/29623/from_business_to_buttons
7. Published in the book *Effektstyrning av IT* (in Swedish, Liber 2004), published in English as *Effect Managing IT* (Copenhagen Business School Press, 2007, out of print)
8. Simon Sinek explains the golden circle
9. The Agile delivery aspect of Impact Mapping was described by Gojko Adzic in the 2012 book *Impact Mapping*, recently translated into Chinese and Japanese.
10. The manner in which different project conditions affect ease of impact management is further discussed in the article *Getting the Most out of Impact Mapping*
11. Dan Pink explains what people need in order to be motivated
12. Minimum Viable Product, usually defined as "that version of a new product which allows a team to collect the maximum amount of validated learning about customers with the least effort." This is one of the key principles in Lean Startup Methodology. Recommended reading, Cindy Alvarez: *Lean Customer Development*.
13. http://en.wikipedia.org/wiki/Design_thinking#Divergent_thinking_versus_convergent_thinking

Code Review: Why It Matters

Trevor Lalish-Menagh, @trevmex , http://trevmex.com

Code review is the art of studying another person's code, making suggestions and observations and learning from one another. It is a powerful tool for building knowledge of a code base, creating strong bonds in a team and improving code quality. Used wisely, code review can help make a product better and be released with fewer defects. Used poorly, in an unsafe environment, it can paralyze teams and cause strife. Whether you participate in a code review process today or you are looking at experimenting with it in your team, this article hopes to guide you to a better experience and understanding of the art of code review.

Before we dive deeply into the topic of code review, though, we first have to talk about something else: fear. Code review, the practice of allowing others to study and comment on code that you have written, exposes you, the writer of code, to extreme vulnerability. For many people, their code is their heart and soul, exposing that to others is a very personal thing to do, and can only work well when a coder is placed in a safe environment. Therefore, the first thing we must make clear when talking about code review is that effective code review, that is effective and constructive criticism of code, can only be received when the person receiving the review is made to feel secure in their position and that their work is valued to the organization. Without this cornerstone of trust, code review, and really any constructive criticism, will be met with paranoia and fear.

The source of that fear often comes from an individual thinking that they will be "found out" as a bad programmer (this type of fear is referred to as imposter syndrome [1]). Imposter syndrome is a common phenomenon among women in knowledge work, but affects many developers, especially those at higher skill levels than their peers. It is our job as people taking part in a code review to understand that this anxiety might exist in us and others, be respectful of that and comment appropriately. A code review is not a place for blaming or shaming, it is a tool we use to grow as developers, help improve quality and support one another.

Types of Code Review

There are three main types of code review we think about when we think about code review: group code review, post-commit code review and pre-commit code review. Group code review, the least effective and most intimidating of the three, is where a group of people get in a room together, the developer puts their code on a big screen and walks everyone through it. This has the double-down sides of putting the individual developer on the spot and gives others power to shame and posture over the individual. Both of these activities have the result of weakening the safety of a work environment, making trust and collaboration more difficult to achieve. For these reasons, group code review is not what I focus on when I talk about code review.

That leaves post-commit and pre-commit code review. Post-commit code review is the method of reviewing a single code change, usually in the form of a patch that has been applied to a repository of code *after* it has been checked into that repository. This has the advantage of ensuring the code gets into a deployable state (i.e. in the code repository) fast, but has the downside of potentially letting defects pass through into the repository without review. The lack of gating that post-commit review offers makes it a weaker choice than pre-commit review, but still far better than group review, since post-commit review can be accomplished in a one-on-one basis and at the reviewer's leisure.

Pre-commit code review, using specialized tools like gerrit [2] or Review Board [3], allow you to gate a code patch and ensure it passes the code review process before being added to the code repository. This approach gives you all the advantages of post-commit review as well as ensuring code is understood and approved of by more than just one person on your team before it goes out to production. I have found that once adopted, this method is the most effective and least likely to be ignored of the three.

Why Review Code?

Regardless of the method your team chooses to go about code review, there are a number of important benefits that can be gained from studying and commenting on other's code and having your own code examined. Key among these benefits is mentorship. Everyone, from the most junior to the most senior developer of your team has some bits of knowledge that others on the team do not possess. By showing others how you develop code they can learn tips and tricks that you might have learnt long ago and use without thinking. On the other side, studying the code of others, especially the code of younger developers can expose you to different ways of approaching a problem and can help you expand the way you might tackle similar problems in the future.

Another reason to do code review is to reduce what we call the bus factor [4]. The bus factor is a rather morose analogy that goes as follows: if Jim the Coder, who has been the only developer working on a particular sub-system gets hit by a bus tomorrow, how long will it take the team to get up to speed on Jim's work? If no one else has been looking at the code Jim was working on it could be a very long time until another person would be able to get up to speed on the system. In extreme cases, the system might have to be rewritten altogether. By ensuring that another person reviews all code, the process of code review helps to mitigate the concerns that arise with the bus factor.

The most compelling reason you might seek out code review, though, is to find defects as early in your development process as possible. Two sets of eyes are better than one, and with another person examining code besides the developer that wrote it, less defects could slip by before code gets out to your repository. Not every defect will be caught, nobody is perfect, after all, but it certainly can help.

How to Review Code

The mechanics of code review are quite simple:

- Write some code.
- Ask someone to look at that code.
- Incorporate their feedback into your code.
- Repeat steps 2 and 3 until both parties are satisfied.
- Submit the code to the repository.

The devil is in the details, though. Giving high quality feedback is the real art of the code review, and there are a few areas that will give you the most return on your investments. The first place you will want to focus on while reviewing code is logic statements. This might sound intuitive, but when faced with another person's code, one can quickly get overwhelmed. Concentrating on logic statements (if/else, switch/case, loops, etc.) first can help focus your review. You want to try and understand the reasoning behind the logic, and ensure that it makes sense, e.g. does an if statement have an else attached to it? Does a switch's case break properly?

What are the exit conditions of this loop?

Beyond basic logic, you can dive into cyclomatic complexity [5] issues. Cyclomatic complexity is the measurement of how complex a block of code is. Generally, if a block of code is deeply indented, meaning it has many nested if statements and loops, that is a sign that the code could be improved upon. This can often be achieved through refactoring [6], the art of redesigning code to improve understandability without mutating the outcome of the code itself. Improving understanding in a piece of code is a main goal of good code review.

To that point, another useful item to look out for in code review is proper naming. I have often heard from less experienced software engineers that the naming of variables, functions and classes are not very important. This cannot be farther from the truth, though. We write code for two audiences: a machine and a human. The machine does not care about names, in truth it ignores them almost all together. The human, on the other hand, cares deeply about names. Meaningful names turn confusing code into something that makes sense in the context of a larger system. Single letter variables and non-descriptive function names should be an artifact of a bygone era, if you see them in code you are reviewing, call it out.

When you are reviewing code, it is easy to get angry at what you see, especially if you spot errors or are reviewing code that was written in a different way than you expected. In these situations many people have a tendency to be passive aggressive. This is especially true when using a code review tool that is not in-person, and reviews are written down and reported back to the developer. It is because of this I would strongly encourage everyone that writes a review of a piece of code to reread their review and imagine themselves on the receiving end of their criticism before posting it. Reviewing code is about improving the system as a whole, it is not a tool to be used for lording one’s superiority over another.

If you are implementing code review for the first time in your team, I would encourage you to not set any artificial restrictions on who reviews code, when they review it and how it is reviewed. Many managers, when faced with implementing a code review system tend to want to add rules around how the process is done. If you can resist the urge to do this, you will find a few things will emerge from the process. The team will start to form their own rules around how and when code is reviewed. Having rules grow organically instead of being imposed, especially when attempting to invoke major culture change can give the team a sense of ownership to the process they otherwise would lack.

An interesting side effect of giving a team control over doing code reviews is that it will brew conflict, oftentimes about what might be considered trivial issues. In my experience, chief among these trivial issues that always surfaces is the fight over tabs versus spaces to indent code. This is a classic case of bikeshedding [7]. Bikeshedding is the process of obsessing over seemingly meaningless details for much too long while ignoring more pressing issues. This happens largely because of lack of familiarity in a system. When presented with an unfamiliar piece of code and asked to comment on it, if a developer cannot understand what is going on, but can see that the code has mixed tabs and spaces, they then have something concrete to comment on, however trivial. This is an important phase to recognize in the adoption of code review, and can give the team a chance to come together to discuss how to develop better code. In the tabs versus spaces situation, a team might hold a meeting to codify basic coding standards and produce an artifact to hand down to new developers, or implement a static code analysis tool [8].

Unfortunately, as is true with any process change, there will be resistors. Most resistors' issues with the process come back to fear. The number one complaint I hear when implementing code review on a team is that it will take too much time. This argument implies that un-reviewed code and reviewed code will have equal value downstream. Though if even one defect is found earlier in the process, or a chunk of code is made easier to understand the team is ultimately saving time in the long run. In two teams I implemented code review on in a large multimedia company, we found that although initial velocity dipped, it quickly recovered and then surpassed pre-review levels over the course of a year study.

There are many tools you can utilize to do code review. Gerrit is a tool that was invented by the Google Android core team to perform remote code review with git [9] repositories. Review Board is a tool that can be used with any number of version control systems and achieves a similar purpose. These tools give the developer a web interface to study and comment on code at the line, file and patch levels. They support email messaging and continuous integration options. Both gerrit and Review Board are free to use.

There is no reason to wait to perform code reviews with a tool, however. Your team can start implementing code review tomorrow by simply reviewing code in person. Once a developer is ready to commit a patch of code, they will call over another developer to walk them through their changes. Commenting is done face-to-face and changes happen right away. I have worked with a number of teams that prefer this method and do it quite effectively. Toolchains can sometimes get in the way of inherently humanistic systems, like code review.

The Side Effects of Code Review

Reviewing code has some added benefits beyond those stated above. One surprising change that often occurs is smaller code patches being submitted. This is due to the fact that people tend to not want to have to understand large and complex changes all at once. This results in larger code patches lingering in a code review queue for sometimes days at a time. Smaller code patches are easier to understand, and thus easier to review. They are also easier to revert if something is wrong with them. Smaller code patches benefit the team as a whole because of this.

Reviewing code and having your code reviewed and a safe environment also has the powerful effect of humanizing one's coworkers. When faced with a system that is not working, it is easy to blame other's work, but code review puts a face to the code itself. The act of reviewing code opens up lines of communication the previously might not have existed. Once you have worked together with another person on a line of code, it is much harder to blindly judge them for their oversights.

Over time, teams that perform effective code review also gain more trust from their peers and supervisors. This is due to many of the effects stated above: less defects, stronger lines of communication and increased velocity. It takes time to build trust, but a team that is confident enough to share its code with each other will quickly rise to that level.

I would be remiss in this article if I did not mention pair programming [10]. Pair programming is the practice of two developers sitting together and coding together. One developer writing code, and another reviewing the code as it is written. The topic of pair programming is worthy of an article in and of itself, but needless to say that if you are in a team that is making effective use of pair programming you have already achieved many of the goals of effective code review. I like to present code review as an introductory step towards pair programming. It is an easier system to implement and can achieve many of the same goals in a less disruptive manner.

The moral of the story is that code review has the potential to make your team stronger, your code cleaner and you a better developer. If you haven't tried it out before, I encourage you to run an experiment with your team to implement code review with either a tool or by hand for four to six weeks. If you are in a team that is already implementing code review, I hope I have been able the shed some light on the practice. Together we develop better software. Happy coding!

References

1. https://en.wikipedia.org/wiki/Impostor_syndrome
2. https://code.google.com/p/gerrit/
3. https://www.reviewboard.org/
4. https://en.wikipedia.org/wiki/Bus_factor
5. https://en.wikipedia.org/wiki/Cyclomatic_complexity
6. http://refactoring.com/
7. https://en.wikipedia.org/wiki/Parkinson%27s_law_of_triviality
8. https://en.wikipedia.org/wiki/Static_program_analysis
9. http://www.git-scm.com/
10. https://en.wikipedia.org/wiki/Pair_programming

#NoEstimates - Alternative to Estimate-Driven Software Development

Henri Karhatsu, @karhatsu, http://blog.karhatsu.com

Our software development industry is estimate-driven. That can be seen in many levels. But are there really any alternatives? Is estimating necessary in software development? This article answers these questions by explaining what is #NoEstimates and by exploring some of the alternatives that we have to estimates.

Mainstream Agile

Agile has clearly become mainstream. It's difficult to find a company that does not either do or pretend to be doing Agile software development. But at the same time it seems to me that many of the companies still miss the essence of Agile. I can see this happening in at least two ways.

First, I too often see Agile teams that never have retrospectives or anything similar. The team members occasionally do talk about the problems they have in their work, but they do not actively try to track reasons for problems and to solve them. As a result, they keep having the same problems every week. Then, I hear comments like "we are having this demo because we follow Agile rules". People in Agile do things ritually because they have decided to do Agile, not necessarily understanding why that ritual might be valuable.

The lack of continuous improvement is the common factor for both issues. Stepping into the world of Agile might quickly bring some nice benefits, but the point of Agile is not just to take some Agile tools into use and to stop there. In my opinion, the most important thing in Agile software development is the idea that you are never ready, that there is always room for improvement. And this is where #NoEstimates greatly fits in.

#NoEstimates, not a method

To me #NoEstimates is an indication of continuous improvement. It is not a method or framework like Scrum, nor a 10-steps program to success. Instead it is an invitation to discuss estimates-driven software development and look for alternatives to estimates. Woody Zuill, the most important #NoEstimates advocate, has defined the term like this:

> "#NoEstimates is a hashtag for the topic of exploring alternatives to estimates for making decisions in software development. That is, ways to make decisions with 'No Estimates'."

The important thing in this definition is that people taking part of the #NoEstimates discussion are looking for alternatives to estimates. This can happen in many ways. People talk about the topic on Twitter, there are many blog posts written, many conference presentations have been held. There's even a #NoEstimates book written by Vasco Duarte [1].

There are many alternatives to estimates. A Scrum team could improve its process so that it can drop the estimation ceremony from the sprint planning meetings. A company might not be satisfied how its software development projects are started and run based on estimates. It might be an ad-hoc situation where a manager asks the developer to estimate a new feature for prioritization reasons but together they find a better way to decide about the priorities.

The common thing in all of these situations is the goal to do better software development. This can mean benefits like less waste in the process, faster feedback, improving the quality of the decisions made, getting faster to the market, etc. The purpose is not to refuse to estimate. #NoEstimates may be developer-driven but it is not a developers' conspiracy to avoid doing estimates just because doing estimates doesn't feel nice.

Maybe this misunderstanding has sometimes caused that #NoEstimates discussion has been quite turbulent, especially on Twitter. Personally I find that really sad because, like I stated above, the heart of #NoEstimates is continuous improvement. Nevertheless, it might be worth to point out also that #NoEstimates doesn't mean that everybody has to stop estimating in every possible context. If estimation brings value for you in your context, just keep doing it. However, don't expect that all the other people are in a similar situation.

Gateway drug

But if the goal is to do better software development, aren't there usually more important and acute problems than estimates? I absolutely think so, but for some reason estimating seems to be a very powerful tool for revealing these problems.

When someone tells me that estimates are needed in their company, I start to ask questions. Why do you have to rely on estimates? What is the reason that prevents you from stopping estimating? What would have to happen so that estimates wouldn't be so relevant information anymore?

These questions may not always be easy to answer but they usually reveal organizational issues. Typical examples are harmful project-thinking, missing trust between people or companies, lack of true collaboration, inefficient budgeting mechanisms, lack of visibility, slow communication systems, teams' inability to produce quality software in small pieces, lack of continuous deployment possibilities, harmful corporate policies, etc.

Solving these problems may be really hard but if you start working towards that, you may eventually be in a situation where you don't need estimates, at least not that much anymore. If you get there, getting rid of estimates has been just a nice-to-have level improvement compared to anything else you have achieved in your organization.

What is relevant here is that usually estimates, and especially their misuse, are just symptoms of some real issues that may not be so visible. #NoEstimates does not aim to solve the symptom because even though that would be easy - just stop estimating - it would probably be also dangerous. Instead, the purpose is to find and eliminate the root causes, to change the system conditions you are living in. So #NoEstimates can be, like my friend said some time ago, your gateway drug to systems thinking.

Teams that do not estimate

Even though I said that stopping estimating might be dangerous, this is not true for all teams. There are teams that estimate just because it is an organizational habit creating only waste and adding no value. I was telling my own #NoEstimates experiences to one of these teams a year ago without knowing much nothing about their situation. Later I heard that the day after my presentation, they had gathered together and decided to stop estimating. This triggered a lot of positive change in the team and around it. [2] This team is just one example. The last four teams that I have worked with have not been doing estimates. This means primarily task or user story estimates during the normal work process but also some bigger features that initially do not fit into the basic process as such. If and when these teams exist, it may be worth to understand how these teams work? What are their system conditions like?

Although every team is unique, it is certainly possible to find some patterns from each or most of them. One of them is transparency. The teams that I worked in try to be as transparent as possible for the stakeholders. Our whiteboards show the task status openly for anyone who is interested, and anyone can come and ask more what we are doing or about to do.

Another important thing is reasonable task sizes. If your task sizes vary for example between 1 and 30 days, it makes sense to estimate them. But if your tasks sizes vary between 1 and 3 days, you can focus much more on the value side of the task and just build the ones that are most valuable.The key here is to have small enough tasks by splitting the big ones. There are several ways to do this. Neil Killick uses his slicing heuristics [3]. I tend to visualize the user story and find relevant pieces that become the building blocks. You can also consider asking questions like "can we make this task smaller" or "can we make this task simpler".

When you split your tasks into smaller ones, you create yourself a very valuable side effect: options. If you have one big task, you can either do it or not. If you divide instead this task into 5 smaller ones, you have options. One option is to postpone some of the tasks while you get benefits from doing the most important ones. Another option is to decide that you don't do some of the tasks at all. Companies do not always understand that the biggest and easiest savings in software development come from the features that are not developed because they are actually not needed.

One very important thing in teams that don't estimate is that they deliver instead. We talk here about mature teams that are able to produce working software in small pieces. This is probably the most valuable thing from the #NoEstimates point of view and gives the stakeholders really nice opportunities. When they prioritize tasks, they can focus on the value aspect much more than to the cost aspect.

There are better parameters

Teams that deliver working software often help their stakeholders to focus on the value over the cost (estimates). However, even when your team is not there yet, I recommend challenging the typical way of making the prioritization decisions.

Once in a while someone asks me an estimate so that she can make some kind of decision. I never give any estimate right away but instead ask more background information. I'm interested to understand better what decision will be made based on the estimate. Usually I find out that asking an estimate is just a habit and the decision is better made by using some other parameters.

Sometimes the best decision is not to implement the feature at all. Sometimes there are two options and one is much more valuable than the other. Sometimes we just don't understand the situation well enough and it's best to gather more information for example by building a prototype.

I would like to emphasize that I do not refuse to estimate. Instead I want to understand what kind of problem we are solving together and based on that I want to help finding the best tools to solve that problem. Estimating rarely is the best tool according to my experience.

Estimating and forecasting

It is possible to replace estimates with something better in many different situations but eventually you will face a case where none of the previously mentioned ways works. For example we may want to get an idea when a feature consisting ten tasks will be ready in order to plan other related activities like marketing. Luckily we do not have to rely on estimates in such a case either. Instead of estimating we rather forecast.

In his book Vasco Duarte goes nicely through the difference between estimating and forecasting [1]. Quoting The Merriam Webster dictionary he brings out that estimating is "giving a general idea about the value, size, or cost of something". Forecasting on the other hand means "calculating or predicting some future event usually as a result of analysis of available pertinent data". The important difference is that the latter one uses data when the first one does not.

We can use data also in software development and replace estimates with forecasts. Metrics I have used are lead times, weekly throughput, and takt time. Below you can see example data of a real project. The data is calculated in the Agile Data tool (http://agiledata.herokuapp.com) based on the start and end dates of the tasks that my team has finished.

Real project statistics including three different metrics (source http://agiledata.herokuapp.com*)*

Weekly throughput means the amount of tasks finished during a week. Takt time is the difference between end dates of tasks. For example if the team has finished tasks in March 2nd, 3rd, 3rd, and 6th, the takt times for those are 1, 0, and 3 days. Lead time for one task is the end date minus the start date.

Based on the data above it is really simple to calculate some forecasts. For example since the average takt time during the last three weeks has been 2.1 work days, the forecast for the next 10 tasks is 21 work days, or 4 work weeks. Picture 2 shows forecasts for the next ten tasks for the each three metrics. We could enhance these even more by taking the deviation into account and showing confidence intervals but let's skip that for the moment.

Forecasts for the next 10 tasks using the statistics in Picture 1 (source http://agiledata.herokuapp.com*)*

If you have been doing Scrum, you probably have used story point estimates and calculated the team velocity. It might look similar to this, but notice that there is a big difference in collecting story points data and collecting the metrics mentioned above. Story points are estimates meaning that when you measure story points velocity, you effectively measure estimates. But when you measure for instance weekly throughput, you measure actual progress.

Challenge of new projects

The forecasting methods shown above are examples of reference class forecasting [4]. Its purpose is to avoid some issues that are related to traditional estimating. We can use reference class forecasting here because we have a valid reference class. The data has been collected by the team X who use technologies T in the customer domain D. The forecasts are applicable because the team X continues working in the same customer domain D using the same technologies T.

But what if we are about to start a new project? There will be a team that has never worked together. There are some well-known technologies but also some new ones. And these people have never worked for this customer before. We could surely take some numbers from previous projects but applying them would be quite questionable.

Things are getting more difficult here, but at the same time also more interesting. We are phasing the famous project dilemma. Why are we asked to make estimates when we know the least of the project? Certainly customers want to know how long their project will take and how much will it cost but there is just one problem. They cannot know. Vendors and customers can only create an illusion that they know these things beforehand.

But what if we could change the system conditions also here? Maybe we should rethink how projects are started and run?

Project thinking

For some reasons, companies have a tendency to do things big. They say they are doing Agile software development, but at the same time they can need up to one year to start a new project. The budgeting process is usually really heavy and managers do their best to get the money for their projects. When the money has been granted, it is perfectly OK to use all of it or even go a bit over the budget. The process obviously includes vendor selection that can take even months. The vendor candidates naturally have to guess (estimate) the project size and duration when they know the least of the project. While all of this takes place, the company's CEO emphasizes in his presentation how short time to market is important factor for the company.

What if the companies would aim to true agility, also outside their Scrum teams? If the project should last 6 months, the company could first hire the vendor for a one-month project. The purpose of that would be to invest in knowledge. As a side product, the team would build working software that could be used to evaluate the concept and that could be used as a basis for the future development. After one month the company would have options. First of all, is this worth to continue or should we just accept that it was a bad idea and try something else? If it still seems to be a good idea, the team and stakeholders have probably learned a lot compared to what they knew month earlier. They may have even collected some statistics that can be used for forecasting the future progress.

Selecting a good vendor is obviously important here. If you let your sourcing department do it, it will probably sub-optimize by reaching their own goals: as much with as low price as possible. But if you do smart software development, you consider the vendor quality as a primary factor instead. If you are not familiar with the vendors, ask them how they develop software and how they would start solving your problem. Collaborate already before the project starts. Also prefer vendors who are fine with short-term notice. Of course the price matters as well, but only after the other attributes. Having price as you main criteria could be really expensive for you.

I'm aware that for some companies it is really difficult to work so that anything would happen in one month. But then on the other hand, this just means that we are seeing some organizational issues that might be worth solving.

We are uncovering better ways

The main goal of #NoEstimates is not to get rid of estimating. The main goal is to challenge the current estimate-driven software development and find better alternatives to estimates. The heart of #NoEstimates can be described with one sentence. It's the first sentence of Agile Manifesto:

> "We are uncovering better ways of developing software by doing it and helping others do it."

During my #NoEstimates journey I have found many better ways and I believe that I will find many more. I welcome you to come along.

References

1. http://www.noestimatesbook.com
2. http://visible-quality.blogspot.fi/2014/11/power-of-stories-and-how-one-hour.html
3. http://neilkillick.com/2014/07/16/my-slicing-heuristic-concept-explained/
4. http://en.wikipedia.org/wiki/Reference_class_forecasting

Self-Selecting Teams Part 2 - Keeping the Momentum

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

In this second and final part of our article on self-selecting teams, you will learn how to get your self-selection event off the ground. We will also provide tips on how to get the most out of it and keep the momentum going afterwards

In the first part of this article [1], we wrote about what self-selecting teams are, the problems they can solve, and how to prepare for and run a self-selection event. We included a preparation checklist:

- Readiness check
- Run a trial
- Define the teams
- Logistics of who, when, where
- Communication
- Rules and constraints
- Prepare materials

You might want to read the first part and to review these concepts before reading this part.

5. Permission versus forgiveness

Once you've decided you want to go with self-selecting teams, you're going to have to convince others in the organisation that it's a good idea. It's unlikely, however, that you're going to get complete buy-in for self-selection at the outset. If you know you can, great. But depending on where you are in the management hierarchy you may find that seeking forgiveness after the fact, rather than asking for permission beforehand, is the best way forward.

We know from experience that people will likely feel scared or challenged by a concept like this in the beginning, so if you start by asking for permission and don't get it, then your project is over before it's begun. We didn't have full buy-in at our Kiwi eCommerce provider in the beginning, but at least it was an organisation that operated on trust where people were inclined to give you enough rope to try something new.

We felt we needed to just go ahead and do it rather than try to persuade everyone at first. But while we didn't ask for permission, nor did we act like cowboys. We moved slowly and kept everyone in the loop along the way. Our aim was full transparency from start to finish. In fact, we probably erred on the side of over-communicating, but at least it was communicating what was actually going to happen rather than asking if it would be a good idea.

We worked a lot on risk and were very honest about the fact that we didn't know for sure whether this would work. What we did know, however, was that the worst-case scenario wasn't so bad. Ultimately the worst thing that could happen would be one day's lost productivity (which we put a dollar value on) if it didn't work, and then we could easily move back to managerial selection.

And that's what we talked about, the small size of the risk, rather than letting people imagine awful scenarios of developers fighting each other in the corridors or a complete breakdown in our current structures. We were also in a position of strength having already run a trial event, which meant we knew what was to come and could manage expectations accordingly.

6. 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've seen a pattern emerge when people first hear about self-selection. 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 had a lot of questions, which was only fair: we were not only asking them to choose who they worked with but in most cases to adopt a new way of working (Scrum, Kanban or whatever combination of Agile ingredients the new group would decide).

We were bombarded with emails, meeting invites and questions from all angles. In the early stages we fielded a lot of 'what if' questions. 'What if someone gets picked on during the event? 'What if no one wants to join a particular team?' We were fairly confident from our trial that these things wouldn't happen but we were also able to answer that we would be facilitating the session (so no bullying) and if no one wanted to join one of our teams that would actually be useful information that we would take away and have a look at.

As time went on, and people moved through 'stages of acceptance', we started to get requests for rules and trade-offs. A bit like 'Okay, I'll try your self-selection process but first you have to do something for me'. Some people requested rules for the event that would make them feel more comfortable. An example was a request for every team to have both a junior and a senior developer.

We politely resisted these requests, though, because each one could have caused a ripple effect that made the problem more difficult to solve. We were also confident that the teams themselves would know best what level of seniority would work for them. We felt this was a critical point in the process and if we'd given in and added rules on request we wouldn't have had such a successful day.

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

7. The dynamics of self-selection events

We have facilitated four self-selection sessions in various locations and have talked to several companies who have followed our principles and process for self-selection. We have observed a number of patterns that emerge almost every time.

7.1. It takes three rounds

Every time we facilitated a self-selection session it took people three rounds of 10-minute iterations to get the hang of things and to arrive at a good solution.

Usually round one is not very successful. People manage to create very few fully formed teams and most of the teams are either wildly over- or under-subscribed. For example, one of our first-round teams consisted of 7 developers and no one else.

Round two is where things start to improve and this round usually results in more fully formed teams. There are still some over- and some under-subscribed teams though.

In round three magic happens. People understand the process, they start talking and teams begin to lobby for members. Negotiations and swaps happen and some Product Owners or team members re-read the team pitches to attract potential members.

7.2. It’s all about relationships

It’s interesting to observe what people base their selection on. The most important appears to be personal relations – who people *want* to work with and who they *don't want* to work with. That said, it can be hard for people to admit this up front. In a survey we conducted after our largest self-selection event, most people said they had based their choices exclusively on doing what was best for the company and what type of work they wanted to do.

But that ran contrary to our observations: during the day most of the conversations we overheard were about who wanted to work with whom and we noticed that many people were only “available" in twos or threes. Often when one person moved, other people moved along with them. Also, people who had not been able to make it for the day were allowed to nominate proxies to make a selection on their behalf. The most frequent instruction to proxies was to “just make sure I’m in your team”.

Sometimes people don’t want to work with each other. And that’s okay. People know whether or not they’re going to gel in a team with a particular person and if not it makes sense that they would choose not to work with them. During our biggest self-selection day we observed two people in particular who seemed to have taken a dislike to each other. When one of them moved their photo to a team the other one was in, the first one would move their photo to a different team. This happened several times and whenever those two coincidentally ended up in the same team one of them would move again.

And this wasn’t a problem at all. There were no dramas and the people who were affected made a choice that was good for them and their teams. Had they been selected to work together, as they easily could have by managers focusing only on compatible skills, both they and their teams would have suffered.

Sometimes it was hard for people to leave an existing team. As one participant put it: "*I didn't like the sick feeling I felt moving out of my existing team – didn't like 'hurting feelings' (for want of a better phrase).*"

Interestingly, people seemed to believe that they were *supposed* to choose without taking personality into account. This is probably because they felt choosing people's personalities over their skills would have resembled a popularity contest rather than an optimum team selection exercise. However, we noticed that people seemed to select who to work with based on similar values rather than who is popular or who they like going for a beer with.

7.3. Most people like the process and the results

After each self-selection session we survey people about their experiences and expectations. We ask what they think of the process and whether they like the results.

Every single time our results have showed that by far most people like the team they end up being part of. Surprisingly, most people tell us that they now work with the team they expected to work with, so people seem to go into the day with some expectations about the outcome already. After the day, almost everyone is in favour of self-selection as the best way to design teams. Even people who initially fear and doubt the process come away with a positive attitude.

At our eCommerce business it was interesting to notice that people in their 30s and 40s, who had been in the workforce for longer than others, had much more awareness of what a privilege the opportunity to self-select was than recent graduates. People in their first jobs probably assume this is normal and that's not a bad thing: hopefully, with their expectations and experience, it soon will be.

There will always be people who don't like the idea of self-selection and we have never forced anyone to participate in any of the sessions. We have always given people the opportunity to opt out by leaving their photo in a "Need a team" area where other people can pick and assign them from. The opportunity to opt out is important, not only for people who don't want to take this kind of responsibility but also for those who are very new to the business. Picking a team in your first week on a new job is a lot to ask.

Here are some of our favourite quotes from our surveys:

"It gave great insight into what other teams and parts of the business do."

"People had equal opportunity to do what they really want to do / think they would be good at."

"It's good that people could decide for themselves, and trust was given to do this well."

"The set-up and agenda were explained well and it was an easy-to-follow process."

"Freedom! Fascinating to see how it all worked out. Excellent result, and nice to know we were able to achieve it without having to get nasty or dictatorial."

"It was a good way to bring issues to the fore quickly and show them visually. I kind of liked going round and seeing whether squads were formed or not."

"I did not observe tension or conflict, which was a great thing. Proved that this was a valuable exercise."

"It did give people the chance to choose where they wanted to go but everyone was also thinking about what would work best for the company."

"People self-organising, surreal."

"Got to chat with people I'd only seen around the office but not really met yet – only been with the company for 4 months. Also the fact that we actually get to do this at all. Would never even have been considered in my last company."

7.4. It really worked for the company

One of the greatest wins was that people proved that they really can be trusted to solve complex problems in a way that's best for the organisation. People in teams who had the privilege to decide for themselves who they wanted to work with weren't the only winners. As an organisation we learned a lot about what people actually wanted to do and some of this information will be very useful for future hiring. For example, after the first selection round almost all developers had moved themselves into teams focusing on back-end heavy projects, leaving the more front-end and design-centric teams.

For management it was really good feedback to have some areas of the company highlighted that are less appealing – potentially because of the particular match of people and interests or because of the way those areas were managed. It provided a great opportunity to dig deeper and to ask the right questions. Also, the gaps we were left with could drive our future recruitment, which was an important takeout from the day.

Self-selected squads have been more stable than the teams we selected as managers prior to this. They also appear to be more productive and we have seen fewer personality clashes and petty disagreements. Overall, people are happier, teams are more stable and our delivery is better and faster than it was before. Our internal measures tell us that we have a much greater output and higher quality.

8. After the event: the beginning not the end

A successful self-selection day is just the beginning. We came out of our largest self-selection exercise with 11 pieces of paper of photos and names. While they were a perfect starting point they really were just 11 blueprints for teams. Next we needed to create the teams in the real world.

8.1 Expectations

Coming out of the day people have a lot of questions about what's going to happen next, when they can start working in their new teams, and whether this is actually for real or if there will be some last-minute management decree that changes some or all of the team designs.

For us the first thing was to follow up with each team (11 within a week) to discuss ideas, concerns and to find a start date. We ran a meeting with each of the teams using the Lean Coffee [2] format where we not only managed expectations but also handed over some responsibility to the teams to make it happen.

Three main themes emerged during the Lean Coffees:

- When can we start and what happens to our current projects?
- Are we adequately resourced?
- How will seating work?

It's a great sign of people's engagement when they are concerned about their current work and don't want to abandon existing projects. However, all those ongoing projects had created a web of dependencies that we needed to break. New hires were also an issue: some teams were waiting on a new designer being recruited, for example, so they couldn't start until later.

In general, we agreed on the guiding principle that no time is ever a good time so, if in doubt, go for sooner rather than later.

8.2 Creating teams

As we were a pretty big organisation with many ongoing projects and we were growing rapidly, it took us almost three months to finish the process of transitioning existing projects to the new teams and disbanding the legacy structures. Among our considerations were the projects, people leaving, a month of Christmas holidays, and new recruits, and we were also very aware of a danger that management could stop us if they felt the process was starting to cost too much in time and disruption.

Looking back it was fascinating to see how those three months were filled with little bursts of new teams starting when a new person arrived and freed up a group of people. It was a bit like when the perfect shape comes down in Tetris and we could fill 3/4 lines at once.

During the transition it was helpful to make the timeline visible to everyone in the organisation. We displayed a big wall calendar with planned start times for teams and checked off teams as they launched. We believe that the transparency was an important part of keeping things going and not losing momentum. If there's one thing we would add next time it would be regular follow-up meetings every other week or so with the not-yet formed teams to keep their focus and buy-in.

One of the main concerns people had was whether their teams were adequately resourced. We found that people often fell back into emphasising roles over skills: while we had made sure that everyone bought into the idea that a team needed a certain number and spread of skills (depending on the team's work) and that these skills weren't necessarily the same as roles, some people still felt insecure about whether they as a team had enough of the requisite skills.

Fundamentally, this was a sign that some teams didn't feel confident yet that they were up to the task and people understandably felt nervous. This is a natural part of any team formation and while we weren't too concerned we reassured teams that we trusted their self-selection and that if, after 3 months, they found they needed another or more of a particular skill, they would be allowed to hire into the team.

8.3 Kicking off teams

With things falling into place and new teams starting in rapid succession, we needed to make sure that the new teams were off to a good start. After all, 30% of a team's chance for success depends on how they kick off. [3]

During self-selection we proved that people can be trusted to solve complex problems, know what's best for them and the company and, in general, be treated as the adults they are. In the spirit of letting people be in control of their way of working and finding a way that works for them, we don't mandate whether teams run Scrum, Kanban or any magic mix of Agile ingredients (or even Waterfall, for that matter, but no team has ever chosen that).

We believe it's important to maintain the spirit of self-organisation and to make sure teams understand that the trust and control they were given when designing their own teams wasn't a one-off. It was ongoing. Truly high-performing teams need to be in control of the way they work. When new teams started we guided them through a process of selecting Agile and Lean practices to help them come up with a process that works for them. [4]

8.4 Supporting teams

After the initial kick-off, ongoing support is important and we offered support, training and coaching to all our teams. We were constrained by the fact that we had only two full-time Agile coaches, so we worked on growing more internal coaches. We set up a structure which enabled people to help and support each other as much as possible. We had an Agile Coaching Guild and offered things like book clubs and suggested reading while also trying to pair people up so they always knew where they could go for support.

9. Go and do it!

Self-selection honours the principle of trusting people to be responsible adults who can solve complex problems and organise in a way that's best for the organisation and themselves. We believe that companies get the best results when people can choose what they work on and who they work with.

Now it's your turn to go and do it, to try self-selection and test your own hypotheses around whether this could work at your organisation.

You can use our two articles as your cheat sheet but if that isn't enough then the following resources could also be helpful:

- Use our Team Self-Selection kit, where you will find all the parts of the toolkit you will need: http://nomad8.com/team-self-selection-kit/
- Read our blog article of when we trialled the process originally: http://nomad8.com/the-self-organising-organisation/
- Read our blog article from when we ran the biggest self-selection event that we believe has ever taken place: http://nomad8.com/total-squadification-large-scale-self-organisation/

Bear in mind that it will never be a good time to do this, you will never fully clear the decks and get all your projects to a perfect state of completion. So go ahead and jump right in!

References

1. http://www.methodsandtools.com/archive/selfselectingteams.php
2. http://leancoffee.org/
3. http://www.estherderby.com/2011/11/miss-the-start-miss-the-end.html - sthash.65FkEH9y.dpuf
4. http://nomad8.com/how-we-kick-off-new-squads/

Laws for Software Development Teams

Allan Kelly, http://allankelly.net/

When discussing and organizing software development teams, there are some principles, sometimes called laws, which teams need to be aware of. These laws may not change a decision you are about to make today but they should inform your about thinking and organizing your teams.

Brook's Law

Perhaps the best known of all the laws is Brook's Law. No discussion of software teams can go very far before Brooks' Law is mentioned:

> "Adding manpower to a late software project makes it later." (Brooks 1975)

Brooks' can be generalised as:

> "Adding people to software development slows it down"

Countless development teams have proved Brooks Law since he first wrote about it. Indeed, Brooks Law - together with Conway's Law - form the bedrock on which much software team thinking need to be based.

When a new team member joins a software development effort they need to learn about what is going on, how the technologies are bring used, they system design and numerous other things. The team slows because existing members must take time to brief the new recruit and "bring them up to speed" - in other words, teach them how the team works and what they need to know, "knowledge transfer." This process is sometimes called "on boarding."

It is not just in the first week that new recruits need help. Some authors (e.g. Coplien and Harrison 2004) suggest it can take up to a year before new recruits are a net productivity gain. Personally I wouldn't put the figure so high but it depends on many factors. It is reasonably safe to assume that few new employees do not require some assistance during their first three months.

In fact, the team slow down may well occurs long before a new recruit is added to the teams. New recruits don't just appear. Managers must request more "resources" - perhaps they need to engage in lobbying of their own managers. Sometimes job specifications must be written, checked, issued to human resources, sent to recruitment agents, the whole process must be managed and then....

Resumes and CVs arrive. These must be read, considered, rejections issued (one hopes), candidates called in for interview, and second interview, packages negotiated and job offers made.All before someone gets to cut a line of code. Even if a personnel or human resources department manages much of the process team leaders and members will be distracted. The time they have for actual development work will be reduced.

Brooks' Law does not imply that teams should not expand, that would be unrealistic and unsustainable. But it does mean that expanding a team is seldom a quick fix and if teams want to grow they must use some of their productivity capacity to grow their productive capacity.

In Joy, Inc. Richard Sheridan makes a bold claim his company has broken the law:

> "I'm pleased to report that Brooks' Law can be broken. ... Our entire process is focused on breaking this law. Pairing, switching the pairs, automated unit testing, code stewardship, non-hero-based hiring, constant conversation, open work environment and visible artefacts all topple Brook's assertion with each." (Sheridan 2013)

Reading Sheridan's description I believe he is right. Whether all these practices are required I don't know. Maybe a team could get by without or another. However I suspect this list is actually shorter than it should be.In the book Sheridan describes a software development environment very different from the one most developers and managers find themselves in. His company, Menlo Inc, goes to great lengths to build, share and strengthen their culture and community. Until more companies embrace this approach and there are more examples to study, it is difficult to say if Sheridan's example can be copied, I hope so.

Right now I believe each of the practices Sheridan describes is worth adopting in its own right. Combined they are even better, and if they break Brook's Law even better. But I also know that just about every company I visit, and particularly large companies, can find a reason why they cannot adopt one or more of these practices. I guess that means that these companies will be constrained by Brooks' Law.

Conway's Law

> "Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations" (Conway 1968)

Another interpretation of Conway's Law would be:

> "Ask four developers to build a graphical interface together and there will be four ways of performing any action: mouse, menu, keyboard shortcut and macro."

The organization and structure of companies and teams plays a major role in determining the software architecture adopted by developers. In time, the architecture comes to impose structure on the organization that maintains and uses the software.

For example, suppose a Government decides to create a new social security system. "Obviously" this will be a major undertaking, it will require a database, some kind of interface and lots of "business logic" in-between. Obviously therefore it requires these database, interface and business logic developers, and since there are many of these folks testers and requirements specialists too.

Suddenly the roles, software and process architecture are visible. Any chance of developing a smaller system is lost. And since all these people are going to be expensive management must be added, requirements set and so on.

Come back in ten years time and the organization maintaining the system will now impose the same architecture on the organization. Reverse Conway's Law can now be observed:

> Organizations which maintain systems ... are constrained to communication structures which are copies of the system.

Now there must be database specialists, business logic specialists, etc. Moving away from such an organization structure is impossible.

Conway's Law tell us that where there are organizational barriers there will be software interface barriers - layers, or APIs, or modules, or some such. This effect can be beneficial - it support modular software designs and application programming interfaces - and it can be detrimental, creating barriers which are obstacles rather than assets.

Conway's Law must be considered when designing teams, organizations and systems. Attempting to break Conway's Law - consciously or in ignorance - will generate forces that have the potential to destroy systems and organizations.

Like cutting wood along the grain it is better to consciously respect and work with Conway's Law than attempt to break it or cut across the grain. This is the key part of Xanpan and informs much of this book.

Dunbar's number: Natural breakpoints

> "Extrapolating from the relationship for monkeys and apes gives a group size of about 150 - the limit on the number of social relationships that humans can have, a figured now graced with the title Dunbar's Number." (Dunbar 2010)

One frequent and reoccurring question asked about software teams is simply: "How big should a team be?" The work of anthropologist Robin Dunbar and his eponymous number, 150, provides some interesting insights when attempting to answer this question.

Dunbar presents a convincing case that 150 is the upper limit for organizational units of people. He also shows that this number reappears in military formations from Roman times onwards, in Neolithic villages, in Amish communities and in modern research groupings. Above 150 community is less cohesive, more behaviour control sets in and hierarchies are needed.

His research and analysis highlights several significant group sizings. Dunbar's Number might be better called "Dunbar's Numbers." There appear to be different groups nested inside other groups, the smaller groups are tighter, and these groups seem to nest by a factor of three.

Thus, 3 to 5 people seems to be most people's innermost group of friends, the next ring of friends is about 10 strong making taking the total to 13 to 15 people. Next 30 to 50, the typical military fighting platoon, and then 150 - the smallest independent unit in a military company and the point at which businesses start to create separate groupings.

Dunbar also suggests there is a grouping at 500 and 1,500, and that Plato suggested the ideal size for democracy was 5,300. Military unit sizes are an interesting parallel:

Organizing unit	Size
Fireteam	four or fewer soldiers
Section/Squad	eight to 12 soldiers, several fire teams
Platoon	15 to 30 soldiers, two sections
Company	80-250 soldiers, several platoons
Battalion	300 to 800 soldiers

Source: Wikipedia, English edition

This list could continue, and of course there are variations between countries and even between different wings within one military. Broadly speaking these unit sizes follow Dunbar's findings.

Miller's Magic seven

In Agile, especially in Scrum circles, a team size of seven (plus or minus two) has become accepted wisdom. However this heuristic has is little more than that, a heuristic. I have seen little or no evidence to suggest five, six, seven, eight or nine is the best answer.

Those who state "Seven plus or minus two" often refer to George Miller's famous paper "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information" (Miller 1956). However I suspect that many, if not the vast majority, of those who cite this paper have never read it.

The paper Miller considers the arguments for seven being a significant number in terms of brain processing capacity - the "chunks" of information the brain can work with. However in the end he concludes that while seven does reoccur again and again there is insufficient evidence to be conclusive:

> "For the present I propose to withhold judgement. Perhaps there is something deep and profound behind all these sevens, something just calling out for us to discover it. But I suspect that it is only a pernicious, Pythagorean coincidence." (Miller 1956)

The paper might have been better titled: "The Magical Number Seven, Plus or Minus Two?"

In his closing words Miller also says "I feel that my story here must stop just as it begins to get really interesting." Indeed, Miller's paper is over 50 years old, psychologists and information theory have moved on.

Admittedly the five to nine person range should give the team a good chance of managing variability. At the lower end it would justify a tester and a requirements specialist and at the upper end could still work with only one of each. So based on my own arguments five to nine makes sense.

But I am prepared to accept larger teams, I believe there are circumstances where this is justified - which I will elaborate on later in this book.

Scrum Team sizing

So how did Miller's paper on individual information processing come to get applied to software team size? The link seems to have be some Scrum texts: The Scrum Primer states "The team in Scrum is seven plus or minus two people" (Deemer et al. 2008). "While the 2011 Scrum Guide states: "more than nine members requires too much coordination. Large Development Teams generate too much complexity" (Sutherland and Schwaber 2013).

To complicate matters the Product Owner and Scrum Master may not be included in this count. The Scrum Guide says:

> "The Product Owner and Scrum Master roles are not included in this count unless they are also executing the work of the Sprint Backlog." (Sutherland and Schwaber 2013)

While the Scrum Primer implies that the Product Owner is outside the team. In short, different writers make different recommendations at different times so who are actually team members - and who is "just involved" - is unclear.

It is a little unfair to point the finger at Scrum. As already noted, teams in range of the four to eight people are seen elsewhere. Miller's paper seems to have provided an easy rationale for enshrining team sizes of seven plus or minus two. Experience also shows there is a limit, however the limit might be a little larger than Scrum suggests.

Parkinson's Law and Hofstadter's Law

> "Work expands so as to fill the time available for its completion"
> Parkinson's Law, Wikipedia
>
> "It always takes longer than you expect, even when you take into account Hofstadter's Law." (Hofstadter 1980)

I am sure that if most readers cast their minds back a few years they will recall being at school, college or university. And I am sure most readers will have at some point been set "course work" or "project work." That is work, an essay, a coding task, or some other assignment, which has to be completed by a certain date.

When I deliver training courses I usually ask the class: "Do you remember your college work? When did you do it?" I feel confident that like those on in my training classes most (honest) readers will admit to doing course work a few days before the deadline. And a few, very honest people, will admit to completing it the night before.

But very few people miss the deadline.

Once, during my master degree, I began a piece of course work very early. I "completed" it very early, but I then used the remaining time to revisit the work, again, and again, and again. To edit it. To improve it.

Psychologists who study these things show that humans are very bad at estimating how long a task will take but very good at working to deadline (e.g. Buehler, Griffin and Peetz 2010a). (My Xanpan book contains more discussion of this topic.)

When more time is available work expands, when more people are available work expands too.

During the late 1990s I worked at Reuters on a project to connect to the Liffe futures exchange. At first the deadline was very tight and it was hard to see how it could be met. In an effort to ensure the deadline a second developer was hired. But then it transpired that this deadline wasn't particularly important, a second, later, deadline was far more important.

The second deadline was easy to make, even with one developer let alone two. As a result far more software was developed to meet it. The system under development was allowed to expand to use all the time and resources available.

Software development is haunted by Parkinson's and Hofstadter's Laws. Asked to estimate how long something will take will inevitably results in too little time, but given plenty of time and work expands.

One research study (Buehler, Griffin and Peetz 2010b) observed that optimism, about how long a task will take to perform, might cause someone to start a task earlier than someone who provided a pessimistic (longer) estimate. But the total time taken by the optimist to perform the task would actually be longer the time taken by the pessimist. Deadline may well be more important than estimates in determining completion times - (see Ariely and Wertenbroch 2002)

Gall's Law - plus Parnas and Alexander

Less well known than the laws above but very important for software development is Gall's Law:

> "A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over with a working simple system." (Gall 1986) via Wikipedia

Gall's Law echoes the words of David Parnas:

> "As a rule, software systems do not work well until they have been used, and have failed repeatedly, in real applications."

Parnas and Gall are emphasising different aspects of the same thing. Something architect Christopher Alexander calls "organic growth." The fact that all three have identified the same axiom in different settings can only lend weight to validity.

In software development a technique called "walking skeleton" advises teams to produce a simple, basic, working piece of code which just pushes all the right (high risk) parts of a system - a skeleton which just about walks. After creating this, the team adds the flesh - layer on functionality - onto something seen to work.

This principle can be applied to the teams as well as the software:

> "A complex team that works is invariably found to have evolved from a simple team that worked. A team designed from scratch never works and cannot be patched up to make it work. You have to start over with a working simple team"

Since a more complex team equates to a larger team this law starts to hint at how large teams can be created, and how Agile can be scaled, or rather, grown.

This obviously parallels Conway's Law: if a team set out to build a walking skeleton then the skeleton needs to be build by a skeleton team. To build it with a bigger, more complex team would be to build more than a minimal skeleton.

When teams start big Conway's Law implies that the architecture will be big and complex, Gall's Law tells implies that such a system will be unlikely to work and in time the team will need to start over with something smaller.

Kelly's Laws

I would like to add my own two laws to this canon. Laws which I coined some years ago. Although scientifically untested I have found to be highly useful in navigating the issue of team size:

> Kelly's First Law of software: Software scope will always increase in proportion to resources
>
> Kelly's Second Law of software: Inside every large development effort there is a small one struggling to get out

The first of these laws follows from Parkinson's Law while the second seems to be a consequence of interplay between Parkinson's Law and Conway's Law. Once a project gets big the work expands, there is still a small project in there somewhere!

If a software team is bigger than absolutely necessary it will come up with more work, bigger solutions, advanced architectures, that justify the team size. It is always easier to add someone to a team than to remove him or her unwillingly.

By keeping the team small, at least initially, create the opportunity to find a small solution. Starting with a big team will guarantee a big solution.

Conclusion

The list is not an exhaustive discussion of "laws" around teams, I'm sure behavioural psychologists could add some more - and perhaps find fault with some that I discussed.

Individually these laws provide heuristics for organizing and managing software teams. More importantly the interplay of these laws can be quite profound.

Given Dunbar's number(s) there are limits on team size and effectiveness, considered with Conway's Law there is a potential limit on system size. The only way around this is to decompose a large system into multiple smaller systems. At first glance this run against Gall's Law but this is not necessarily so provided those systems can be sufficiently separated.

But teams are not suddenly born fully formed and effective. Conway's Law working with Gall's Law again implies they must be grown. Brook's Law implies that teams cannot be grown too fast and Parkinson's Law means that over big teams will make their own work.

Kelly's second law hints at the solution: avoid big, aim to stay small.

One may find these laws inconvenient, one may choose to attack the validity of these laws. Certainly these laws sit badly with the approach taken in many commercial environments. Rather than attack the laws and rather than seek to break the laws, I find a better approach is to accept them and work with them. Finding a way to work with these laws can be commercially uncomfortable in the short run but in the longer term is usually more successful.

Source

This article is an excerpt from Allan's new book "Xanpan book 2: the means of production" which will be available later in 2015. You can register your interest at https://leanpub.com/xanpan2 where you can also find "Xanpan: Team Centric Agile Software Development."

References

Ariely, D., and K. Wertenbroch. 2002. "Procrastination, deadlines, and performance: self-control by precommitment." Psychological Science 13
(3).

Brooks, F. 1975. The mythical man month: essays on software engineering. Addison-Wesley.

Buehler, R., D. Griffin, and J. Peetz. 2010a. "The Planning Fallacy: Cognitive, Motivational, and Social Origins." Advances in Experimental Social Psychology 43: 1-62.

---. 2010b. "Finishing on time: When do predictions influence completion times?" Organizational Behavior and Human Decision Processes (111).

Conway, M. E. 1968. "How do committees invent?" Datamation (April 1968). http://www.melconway.com/research/committees.html

Coplien, J. O., and N. B. Harrison. 2004. Organizational Patterns of Agile Software Development. Upper Saddle River, NJ: Pearson Prentice
Hall.

Deemer, P., G. Benefield, C. Larman, and B. Vodde. 2008. "Scrum Primer." http://www.scrumalliance.org/resources/339

Dunbar, R. 2010. How many friends does one person need?. London: Faber and Faber.

Gall, J. 1986. Systemantics: The underground text of systems lore?: how systems really work and especially how they fail. 2nd ed.. General Systemantics Press.

Hofstadter, Douglas R. 1980. Godel Escher Bach: An eternal golden braid. Harmondsworth: Penguin Books.

Miller, G. A. 1956. "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information." The Psychological Review 63: 81-97. http://www.well.com/user/smalin/miller.html

Sheridan, R. 2013. Joy, Inc. Penguin.

Sutherland, J., and K. Schwaber. 2013. "The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game." http://www.scrum.org/Scrum-Guides

Kanboard - Open Source Kanban Board

Frédéric Guillot, http://fredericguillot.com/

Kanban came out of lean manufacturing techniques made by Toyota that used it to manage their workloads. The same principle can be applied to software development by matching the amount of work in progress to the team's capacity. This method gives a visual workflow, clear focus and incremental changes over the time. Kanboard is a free and open source software that put in practice the Kanban method. You can use it to manage any activity of your company: Lean business management, software development, web-marketing operations, recruiting process, sales pipeline, etc.

Web Site: http://kanboard.net/
Version tested: 1.0.11
System requirements: Web server with PHP
License & Pricing: Free and open source (AGPL)
Support: Bug tracker https://github.com/fguillot/kanboard/issues

Presentation

Kanboard is a web application that implements the Kanban methodology. The concept behind Kanban is pretty simple:

- Visualize your workflow
- Limit your work in progress

The goal is to work efficiently. You have a clear overview of your project with the board where each column represents a step in your workflow. The task limit avoids working on too many tasks in the same time and keeps you focused to get the job done. The Kanban methodology has fewer constraints than other Agile project management approaches like Scrum. It is more flexible and focuses only on the essentials.

Installation

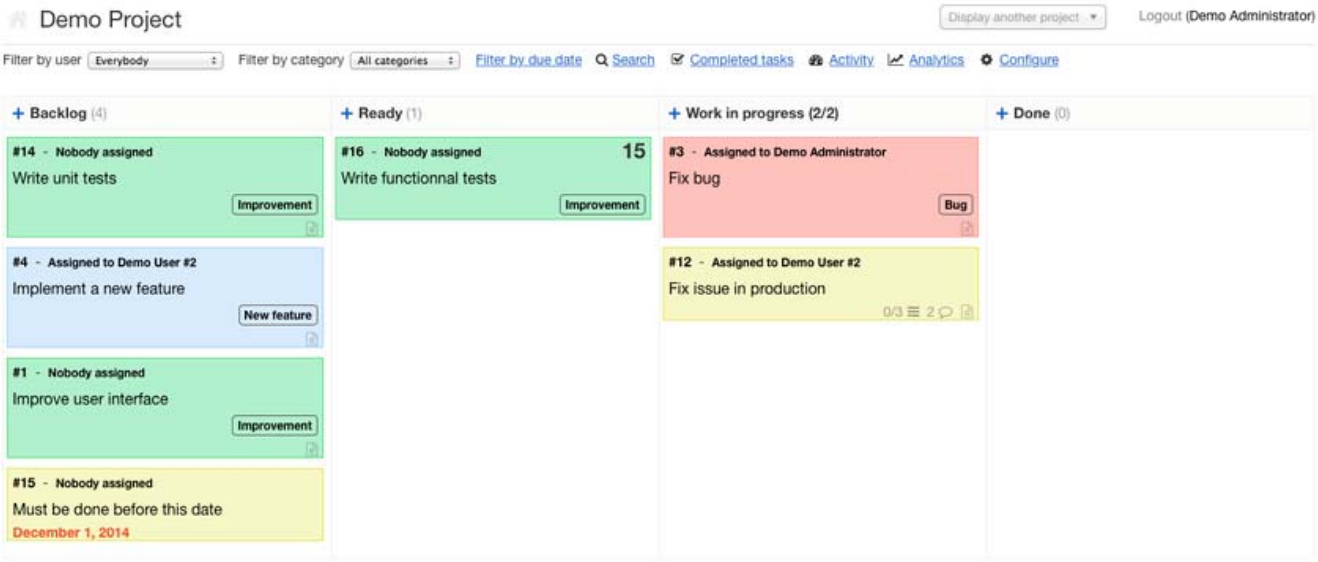
Kanboard can be installed almost anywhere: any shared-hosting provider, on a dedicated server, a virtual machine, a Raspberry Pi or your local machine.

To install Kanboard, you need to have a web server configured with PHP. The procedure is pretty straightforward:

- Download the archive
- Uncompress the files on your server
- Make the directory data writeable
- That's it

By default your data is stored in the embedded Sqlite database but you can choose to use Mysql or Postgresql.

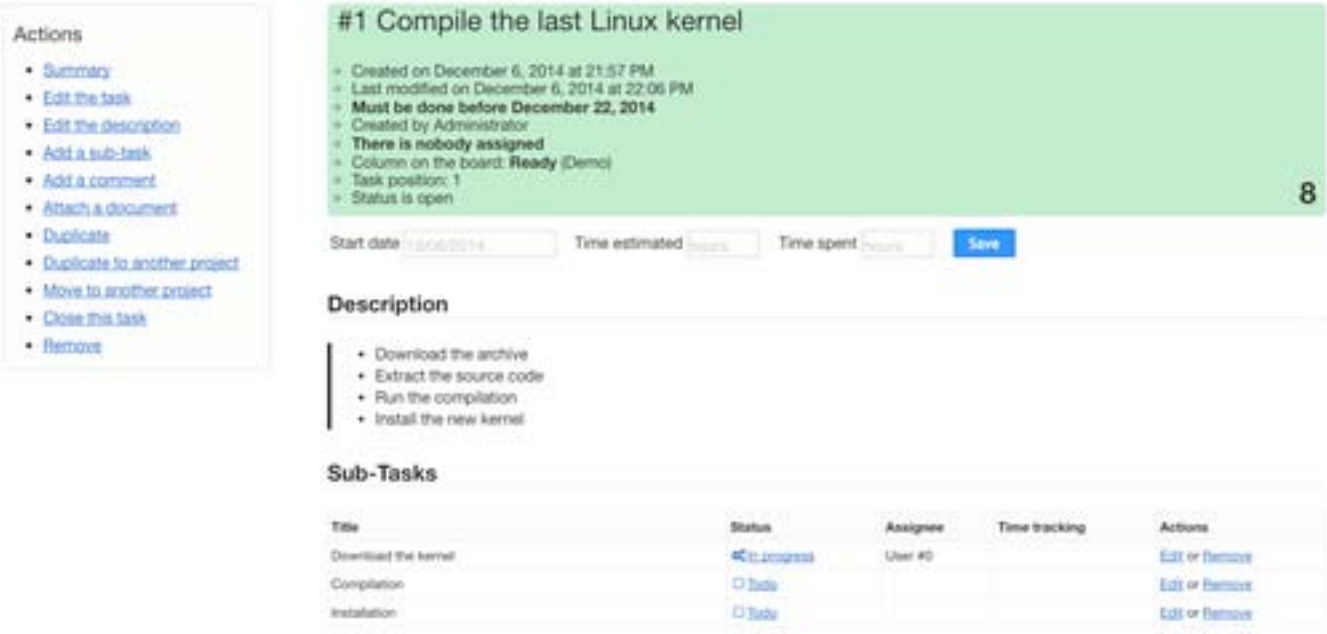
Working with the board



With the board view, you can:

- Drag and drop tasks between columns easily. Switch between your boards in one click.
- Customize the columns according to your workflow.
- Limit your work in progress to be more efficient.
- Create and edit tasks quickly.
- Filter the tasks by category, assignee or due date.

Working with tasks



Tasks have several properties:

- Title
- Description (compatible with the Markdown syntax)
- Assignee
- Category
- Column
- Color
- Complexity: used in Agile project management, the complexity or story point is a number that tells the team how hard the story is. Most of the time people use the Fibonacci series
- Original Estimate: estimation in hours to complete the tasks
- Due Date

Moreover tasks can have subtasks, comments and attachments. You can also duplicate a task in the same project or move the task into another project. Closed tasks are hidden on the board, so only active tasks are visible. You can also track your time by setting a start date and fill the time-spent field. These data are exportable in CSV format.

Working with projects

Kanboard can handle multiple projects. There are two kinds of project:

- Project with multiple users
- Private project for a single user

Private projects don't have user management. They are designed to be used by a single person. Any user can have his own personal project. Projects with multiple users are better suited for small teams. Only administrators can create them.

User permissions

Access to projects can be restricted to some users. A standard user can be promoted “project manager”. This role offers the possibility to change the configuration of the project and to access the analytic data.

Board sharing

Optionally, a board can be shared with external people, a private link allows to access the board in read-only mode from everywhere. This feature is interesting if you want to display your board on a large screen in your office, the board will be refreshed automatically. When sharing is enabled, you also have access to the RSS feed with all project activities.

Integration

Kanboard can be connected to external software. At this time, Github and Gitlab are supported by default. Mostly used by software developers, this feature can synchronize tickets and performs automatic actions based on commit messages.

Board configuration

All boards can be customized. You can change the number of columns, the title of the columns and the task limit per column. When the number of tasks goes over the limit, the background of the column becomes red. That means there are too many tasks in progress at the same time. Multitasking reduces the efficiency of the team.

Category management

Each project can have different categories, so you can group your tasks by categories. If you always create the same categories for all your projects, you can change the application settings to define your categories by default.

Swimlanes

Swimlanes are horizontal rules that separate your boards into multiple zones. It could be useful to separate software releases, to divide your tasks in different products, teams or whatever you want. All projects have a default swimlane. If there is more than one swimlane, the board will show all swimlanes. Of course, you can drag and drop tasks between swimlanes. Inactive swimlanes are not shown on the board.

Automated actions

Automatic actions are a very powerful way to minimize the user interaction. Each automatic action is defined like that:

- An event to listen
- An action linked to this event
- Optionally there is some parameters to define

Each project can have a different set of automatic actions. Here are some examples:

- When I move a task to the column "Done", automatically close this task
- When I move a task to the column "Work in progress", assign this task to the current user
- Assign automatically a color to a user or a category
- Assign a task to a specific user when the task is moved to a defined column
- Close a task when commit is received from Github

Project duplication

Projects can be duplicated in one click, so you can configure a template project with categories, users, columns and automatic actions. You can then copy this template for all your new projects.

Analytics

Kanboard is able to generate some reports:

- User repartition: Number of tasks assigned per project member
- Task distribution: Number of tasks per column
- Cumulative flow diagram: This a graph display the quantity of work for a given time interval
- Export

All tasks and subtasks can be exported as CSV files. So it is very easy to generate reports and work with any spreadsheet software.

User management

Kanboard implements a simple role management system. There are only two types of users: administrators and standard users. Administrators have access to everything and regular users cannot change the application settings. However, any simple user can be promoted project manager to be able to change the configuration of some projects. The Kanboard authentication system can work with external providers. Users can be authenticated through LDAP, ActiveDirectory, Github, Google or a reverse proxy.

Conclusion

Kanboard provides a simple way to manage projects. It is lighter and faster than traditional project management software. Kanboard focus on simplicity and efficiency. The learning curve is minimal: there is no special training needed or complex process to learn. Non-technical people can easily use Kanboard.

ConQAT – The Continuous Quality Assessment Toolkit

Benjamin Hummel, hummel [at] cqse.eu, @BenjaminHummel
CQSE GmbH, http://www.softwarequalityblog.eu/

Long-lived software systems are subject to gradual quality decay if no counter measures are taken. Continuous quality control counters this decay through an orchestrated application of constructive and analytic quality assurance techniques. The Continuous Quality Assessment Toolkit ConQAT provides the tool-support required enacting continuous quality control in practice. It supports the rapid design and configuration of quality dashboards that integrate diverse quality analysis methods and tools. The dashboards come with a set of interactive tools that support the in-depth inspection of identified quality defects and help to prevent the introduction of further deficiencies. To cope with the diversity of real-world software systems, ConQAT is not limited to the analysis of source code but takes into account various types of other development artifacts like models or textual specifications. Through its flexible architecture, ConQAT can be customized to address the quality requirements that are truly relevant for a software project. Thereby, it helps to successfully counter quality decay of software systems.

Web Site: http://www.conqat.org/
Version tested: 2015.2
License & Pricing: Apache Public License 2.0
Support: Provided by CQSE GmbH (http://www.cqse.eu/)

Design Principles of ConQAT

ConQAT was initially created by the Group for Software Quality and Maintenance at the Technical University of Munich in 2005. Starting from the early days, it was created around a set of design principles that guide its development until these days.

Quality data needs aggregation and visualization

Automatically analyzing the quality of large software systems generates an enormous amount of data. To prevent users being overwhelmed by too much data, ConQAT aggregates analysis results so that they are comprehensible and presents them in an appropriate manner. For this, ConQAT must be aware of the analyzed system's structure to produce meaningful results. As even highly aggregated results must be effectively conveyed to users, ConQAT provides powerful visualization mechanisms that go beyond tables and charts.

Dedicated views

Assessment results must be accessible to all project stakeholders, such as through a web site or a specific client. Because participants have different interests, ConQAT can provide a customized view for each stakeholder.

Trend analysis

Many quality defects are hard to identify in a single system snapshot, but one can see them by tracking changes over time. Moreover, various metrics are hard to interpret on an absolute scale and are better suited for relative measures. To foster trend identification, ConQAT can store and present historical data.

Everything is customizable

Quality requirements are highly project-specific due to differences in target systems, applied tools and processes, involved technologies, and project participants. Moreover, such requirements are not fixed, but rather evolve over the course of a project. Consequently, ConQAT is highly customizable to support project-specific tailoring of both the analysis process and results presentation.

Support diversity

Because diverse elements influence product quality, a quality control tool must apply to many factors and artifacts. ConQAT not only analyzes source code, but also provides measures for other artifacts, such as documentation, models, build scripts, or information stored in change management systems. Also, because stakeholders discuss quality attributes on many different levels, ConQAT facilitates all analysis types at different detail and granularity levels.

Extensibility, autonomous operation and performance

To adjust to yet unsupported artifacts or analyses, ConQAT provides an extension mechanism that lets users add further analysis or assessment modules. To obtain timely results in a cost-effective manner, ConQAT must be easy to integrate into other processes, using fully non-interactive analysis tools. Finally, quality control is particularly relevant for large-scale systems, which are difficult to assess manually. A quality control tool must be capable of analyzing large systems within a reasonable amount of time.

ConQAT Core – A flexible and extensible data-flow language

The core of ConQAT is an interpreter for a data-flow configuration language. The central element of ConQAT's configuration languages are so-called processors that are interconnected in a pipes-and-filter oriented style. These processors implement diverse functionality and work like functions that accept multiple inputs and produce a single output. This design offers a high degree of reuse and provides a powerful configuration mechanism as processors may be interconnected in numerous ways. To facilitate reuse, configuration fragments can be organized in so-called blocks, which can then be used to build more complex configurations.

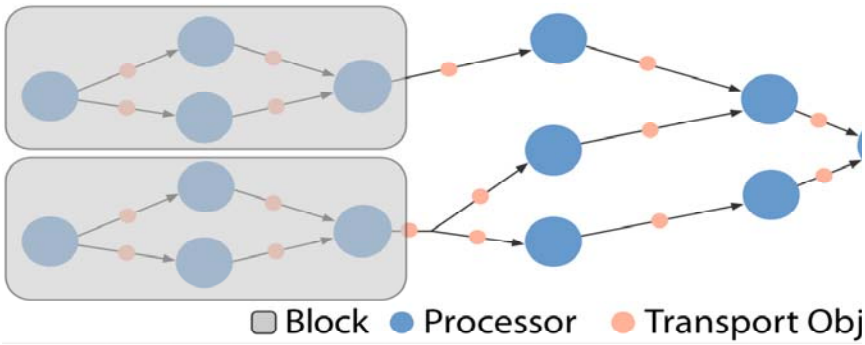


Figure 1: ConQAT configurations are constructed from processors and blocks

ConQAT provides an extension mechanism using bundles. A bundle can contribute new processors and blocks to extend the configuration language. The ConQAT distribution itself consists of a large collection of bundles, but using custom bundles, ConQAT's functionality can be extended as needed.

The ConQAT interpreter is a command line application for executing ConQAT configurations. This allows ConQAT to be easily integrated into other processes, such as a continuous build. The output of ConQAT itself is defined by its configuration and as such can be adjusted as needed. Typically, an output of HTML dashboards is used, which can be easily provided to other developers or integrated into other tools, but other output format, including CSV and XML, are possible.

Authoring Tools

To support the creation of complex analysis configurations, ConQAT comes with a set of graphical tools that simplify the creation of blocks and configurations. The Eclipse based IDE provides a library browser for locating available processors and blocks, access to the documentation of these elements, and also refactoring support, to simplify restructuring of configurations. This ConQAT workbench also allows the execution of configurations with a single click, which greatly simplifies testing and debugging of configurations.

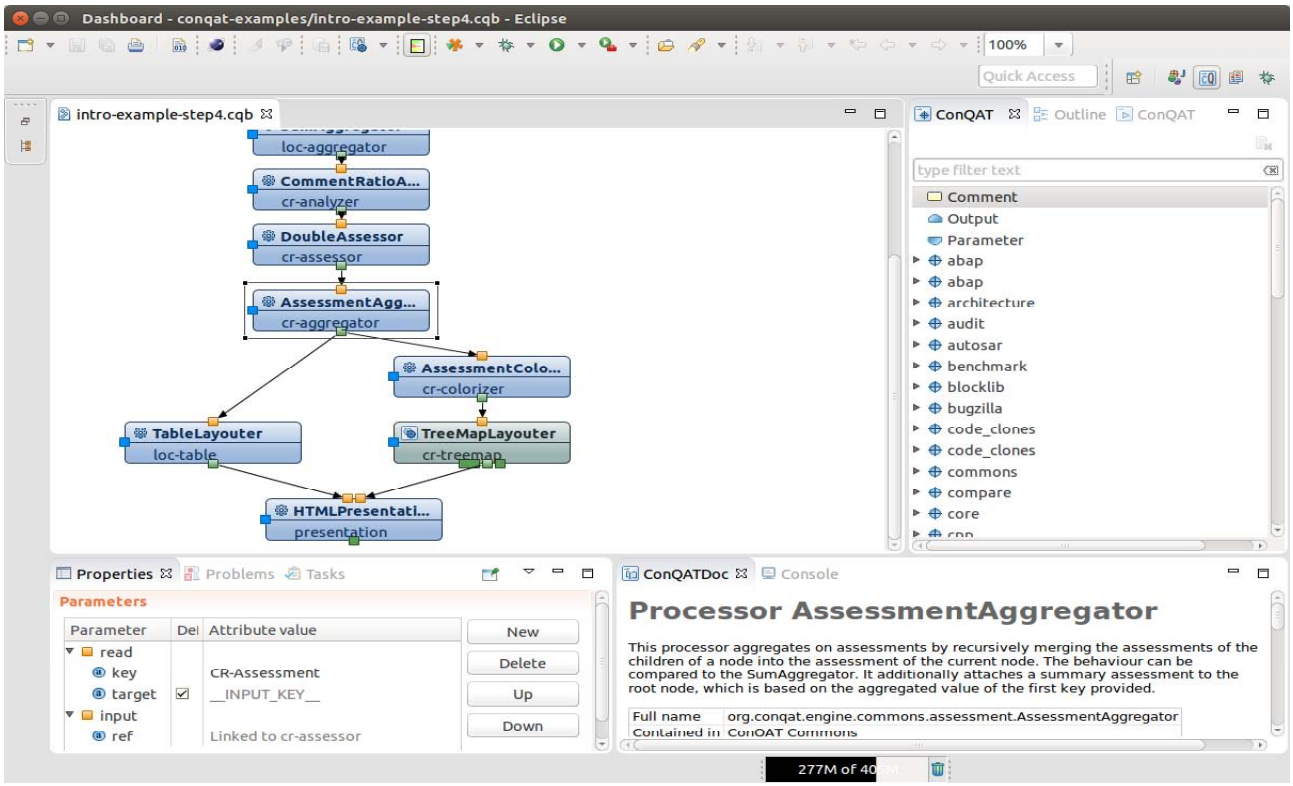


Figure 2: The ConQAT workbench is used to create analysis configurations

Analysis Support

The configuration mechanism and the authoring tools are generic and could also be used for other kinds of tools. What makes ConQAT specifically usable for software quality analysis is its huge range of existing building blocks for the creation of such analyses:

- Different input sources: Read from file system, ZIP files, or the version control system
- Different kinds of analysis targets: Analyze source code, models, or word documents
- Different programming languages: Java, C/C++, C#, ABAP, Python, PL/SQL, etc.
- Wide range of analyses: Starting from simple metrics, such as lines of code, to more advanced analyses, such as clone detection and architecture conformance analysis
- Many different ways of aggregating and filtering analysis data
- Flexible output as HTML, CSV, or XML
- Wide range of visualizations, including graphs, treemaps, charts

This set allows a very fine-grained adaptation of an analysis configuration to a specific project or analysis task. This also helps when developing a new kind of analysis, as the developer can focus on the core of the analysis and can reuse input processing, aggregation and visualization from ConQAT.

Dashboards and Inspection Tools

While each of the building blocks of ConQAT might seem very simple at a first glance, this simplicity allows them to be reused and combined in many different ways. By combining these simple parts, one can arrive at rich dashboards that display a variety of analysis results very quickly. Examples of such dashboards are provided in the ConQAT example project that is provided in the ConQAT workbench download.

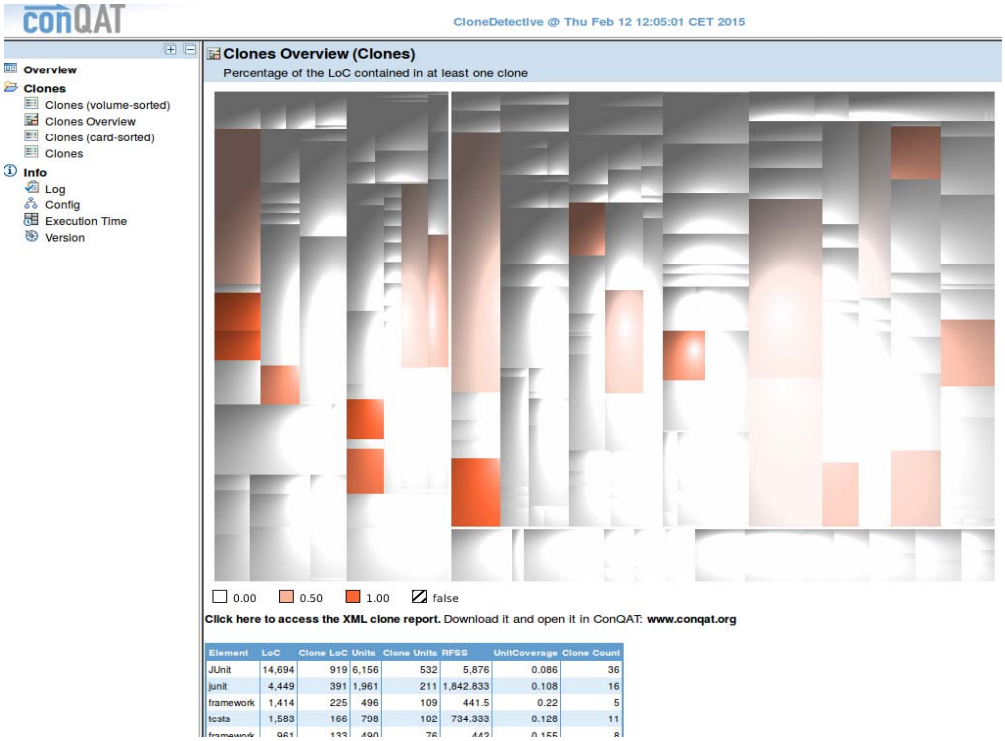


Figure 3: A simple cloning dashboard created with ConQAT

While these dashboards can provide a good overview of a system's quality, it often can be important to support a more detailed view to better understand the root cause of a quality problem. For this, the dashboards can be extended to link to the source code. Additionally, ConQAT provides graphical inspection tools that integrate with Eclipse and can be used to explore redundancy in source code (code clones) and analyze architecture violations.

Where to go from here?

This article can only provide a first idea of the capabilities of ConQAT The tutorials provided on ConQAT's web site (http://www.conqat.org/) are a good starting point to better understand ConQAT. There you can also find the reference documentation, which describes many of the existing analysis steps.

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