METHODS & TOOLS

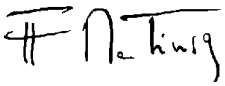
Practical knowledge for the software developer, tester and project manager **ISSN 1661-402X**

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

Money is Back for Software Tools... Or Maybe Not

The beginning of 2015 has seen a wave of new financing for software development tools companies. Mulesoft, Docker, Sauce Labs, Neo Technology, MongoDB, EnterpriseDB or CloudBees are some of the organizations that have received up to $130 million dollars to develop their activities. Even if not all companies will earn money on the long term, this is the sign that many people think that there is still money to be made in the software development tools business. The interesting thing is that all these companies are working around open source tools. Thus they are not making revenues directly with their tools, but with the sales of "enterprise editions" packages or services around the tools. This means also that the basic technology of these companies is not "protected". You could fork MongoDB on GitHub and produce your own version and extensions of this NoSQL database. So why don't people just do this? I think this is mainly because making these open source tools work at the enterprise level with a good performance is tricky. Few companies, Percona might be one of them, have the expertise to build a service and technology business around open source tools that they didn't create. This is the weak side of these tools. In many cases, the tools were initially build with limited resources and tested on small projects. You can read a lot of material on the web saying that NoSQL databases provides nice ideas but lack the knowledge and experience for a performing backend implementation. If you are not lucky to integrate performance from the beginning in the product, it is difficult to add this feature at a later stage. This is why paying enterprise features and services are the place to be. Most companies follow this road, using the same business model around open source that Red Hat introduced years ago.

Some sectors of the market like the NoSQL database are entirely based on open source projects. One reason is that traditional technology companies like Oracle, HP or IBM do not invest any more in new software development tools. There is actually an Oracle NoSQL database, but I would say that it is currently mostly flying under the web radar. The business model of these companies is based on financial performance and they will buy some emerging technology companies when they think this could contribute to improve their earnings. CA Technologies (formerly Computer Associates) is a typical (and long-standing) example of this financial strategy in technology. They just bought buy Rally Software and Grid-Tools, whose customers can now just expect to see an increase in the license prices and a decrease of any product enhancement initiative.



Inside

Software Gardening - Yet Another Crappy Analogy or a Reality?

Patroklos Papapetrou, http://softwaregarden.io, @ppapapetrou76

Introduction

The debate about how close is software development to engineering started a decade ago [1]. People started to realize that writing software has very little to learn from math and engineering [2], but we couldn't find something else to compare it with. The first article [3] I can find on the internet talking about software gardening is actually an interview with Andy Hunt and Dave Thomas, the authors of *The Pragmatic Programmer: From Journeyman to Master*, back to 2003 with. Fortunately, some years later more articles appeared discussing the same analogy and more people got exposed to the strange term of software gardener.

To be honest, I was expecting that software gardening movement would have become as popular as the agile manifesto but I was wrong. This is one of the main reason I decided to write a book about it. In this article I will tell you my short story and how / when I realized that I am a gardener and not an engineer. I will also briefly explain the analogy of software development and gardening and present how it is compared with agile development and software craftsmanship.

The story of my (developer) life

I would love to start my story by telling you, that, when I was a young and inexperienced developer, I had mentors who taught me how to write clean code, but I can't. The ugly truth is that learned the hard way how to treat code as flowers. And that is the second reason that made me write the "Art of Software Gardening" [4]. One of the core values of a software gardener is being a mentor. Sharing our experience and our acquired knowledge should be a part of our attitude. And this is what I want to teach to (junior) developers through my story and my point of view.

In the beginning of my career, back to 1997, I was a very passionate developer, like most developers are or should be when they start writing code. The tools that we were using that era - PowerBuilder, Visual Basic and Delphi - look like dinosaurs today comparing to the modern IDEs and the cool frameworks and libraries that appear almost every day. However, the principles of software development are still the same and I will dare to say that they will hardly change in the future. My only goal, when I was writing code, was to finish my assignments as soon as possible and of course without any bugs. I wanted to prove to my employer that I deserved every single drachma they were investing on me. Funny huh? We all want the same! I remember, one time, how proud I felt, when my supervisor, looked so surprised that I had completed that sexy new feature we wanted in just a couple of days.

"Did you test it?", she asked me while she was trying to make sure that I hadn't only created the user screen but all those buttons and fields were doing what they were expected.

"Of course I did!", I replied with pride. I hope you don't expect to read in this article what kind of unit testing framework I used. The first XUnit framework appeared some years later, in 2002. So, I was manually testing my code by wearing the end-user hat. Clearly, it was extremely difficult to predict all the possible ways that a user might interact with the system and my efforts were limited only to what I was understanding as a developer. This is what we called "testing", twenty years ago.

My professional life was rolling quite well. Some bugs were always expected and they were considered acceptable(!) if customers discovered them when using the system in production. We were a team of 5 developers. Most of the time we were working only on our code and I was so happy I could focus on my goal: be the fastest code monkey. One day, however, something strange happened that completely changed they way I was perceiving software development. One of the two senior software developers - oh yes, senior developers did exist that ancient period - wanted to slightly modify a feature I implemented, after a customer request. Typically, I should have done it, but I was deeply involved in another feature and she didn't want to interrupt me. Besides it was a quick improvement that shouldn't take too long. I still remember that look of her face when she asked me to explain my code. Actually this question was a trap. She was not only experienced enough to understand the code of some newbie like me, but she could also see things that I had never imagined that exist. In the beginning I felt so proud. "*I have the chance to describe my precious code to the most experienced team member*", I thought. "*This is my chance to let her know that I rock!*". Very quickly, that feeling was replaced by embarrassment, disappointment and … failure! I was listening for more than ten minutes her criticism: *"Lots of duplicated code, bad variable and function names, no clear separation of layers, large chunks of complicated code and most important. I have to re-write most of it in order to add that little feature our customer wants"*. Oh God, please help me disappear!

Suddenly, a whole new world was in front of me. Concepts that I wasn't familiar with started jumping in my head. I realized that she wasn't satisfied at all with my crappy code, although she was very gentle when she was explaining his findings to me. And that was my first experience with "clean code" but don't be fooled. I didn't changed the way I was programming from one day to another but that criticism was a shock to me. It was the first time I was feeling, development-speaking, so nude. It took me a couple of more years to start really caring about the quality of the code I was writing and even more to embrace the values and principles of software gardening.

Agile vs Gardening vs Craftsmanship

Obviously, software gardening deals with the issues discovered in the case above but it's much more than just slinging "good" code. The most important problem I was facing, had to do with the fact that I didn't really care about the code I was putting in the system. Let me remind you once again what my goal was to be the fastest code monkey. I didn't care about who would be reading it, maintain it, who would be using the screens I have created and how my code would behave when new features would be added in the system. I was just doing my - what I considered - "job", and then I was turning my back to the code forever. Like what engineers do. They build a house and then you never see them again. But we are not engineers any more. We do care about our code and we will be there to keep it clean and improve it as long our customers need and use it. Right?

I will come back to that later because I want to explain the relation of software gardening with agile practices. Many people ask me if software gardening is just another way of describing the agile movement and they try to compare those two. I want to make clear that software gardening is not competing agile values at all. It's very likely that you have already read the agile manifesto [5] and you know what it represents. The key point here is to understand that software gardening not only makes it much easier to embrace and adopt the agile practices but also takes them to another level. Why? Because agile practices are based on the way we treat software and we, the software gardeners have already changed our mindset. We already care about the code and the majority of the agile principles seem so natural to us.

You can be a brilliant software gardener even if you are developing software using Waterfall or any other monolithic process, but you work in an agile environment it's most probable that you will easier and sooner achieve perfection.

There's one more thing I want to clarify about software gardening and it has to do with "Software Craftsmanship". One might wonder what is the difference between those two definitions. Here's the definition as taken from the recent book "Software Craftsmanship" by Sandro Mancuso:" *Software craftsmanship is a long journey to mastery. It's a mindset where software developers choose to be responsible for their own careers, constantly learning new tools and techniques and bettering themselves. Software Craftsmanship is all about putting responsibility, professionalism, pragmatism and pride back into software development"*. Although I agree with most of the principles presented in that book and in general the idea of software craftsmanship, I believe that writing code is far more than a craft. A crafted software sounds to me more like an object. It might be very carefully designed, of high quality and made by the best materials but it's still an object. I prefer perceiving source code like flowers and software systems like gardens. Software systems live as long as people use them. They need to change, to evolve, to adapt to external and internal environment needs like gardens. You can't add a new level between the 23rd and the 24th floor of skyscraper unless engineering has evolved a lot the last few years. Last time I checked it was impossible. You can't just replace a pair of old or broken bridge pillars. You need to replace the whole bridge. But in software you can add a new layer or replace a module using old technology with a modern one. Because software is like gardening. By treating software as something organic, we expect change. We learn how to care for it so that when that change comes, we're ready for it.

The software gardener manifesto

So it's time to explain what software gardening is. Here's the definition I prefer.

Software gardening is not a practice, an attitude, a skill or a special knowledge. It's all of them plus the love you have for software development. And this love you should show it continuously, every day, in every single line of code you write.

Software Gardeners are professionals who perceive software development in a different way. Our goal is to spread the word of our core values, be mentors for the new developers and create software that it is developed at the highest quality bar.

Every software gardener believes and embraces the following Software Gardener Manifesto [6]:

- We treat software systems as gardens and code as flowers. Although we don't disagree regarding software as a "craft", we believe that software is a living and breathing "being", not just an object, created using the best materials.
- We constantly mentor young developers and we share our knowledge at every opportunity. Junior developers are like flowers that need to be irrigated to blossom. We are the water, the sun, the oil, the fertilizer for every (young) software professional.
- Software development is a lot more than slinging code. We know the practices and we apply them effectively. We make use of the most productive tools and our skill-set includes both soft and technical skills. We also understand that our overall attitude is what defines us as software gardeners.
- We care about our code and we show this care continuously, day by day, every moment in every single line of code we write.
- We are not only able to respond to change but we are prepared about the endless - internal and external - environment reform.
- We treat customers as the people who will walk in our garden will smell and enjoy our fragrant flowers. Having said that we engage them from the first day, to make sure that all their needs, requirements and expectations will be met.
- Gardening is a team sport. There's no room for lonely cowboys like Lucky Luke or epic heroes. We work as a team, we respect the team, we embrace collaboration at its highest level.

Explaining the analogy

Let me give some examples that will make the analogy even clearer. Gardeners don't just plant some flower seeds today and expect to see them blossom tomorrow. Flowers needs their time and the proper environment (soil, sun, temperature, water, fertilizer etc.) to grow. Software gardeners do the same with their code: they write some well-designed (correctly irrigated) code that will grow within the overall system. Our goals is quite simple. Deliver bug-free (no-disease), fully covered by tests (protected by future diseases) features that end-users would love to use.

Another parallelism that I love is about unwanted plants. Gardeners just uproot anything that's blocking their design or doesn't fit in the garden. Software gardeners do the same. We throw away, without any second thought, the code that's not needed (withered flowers) any more or is causing too much troubles (unwanted plants).

The last one is about environmental changes. Gardens have to cope with cold, wind, rain, snow and in some cases with extreme weather conditions. A typical hurricane can easily destroy a wonderful garden in less than two minutes. At the same time flowers and plants are threatened by warms and other bugs - oops that's the same word we use in software; I wonder why. Things are quite similar in software. We have to deal with changes in laws, with evolving customer needs, with new competitors that just launched some cool and attractive features or even with security attacks that can cause a huge catastrophe to our systems and make our customers lose confidence for ever. Internal environmental changes like bugs, people leaving the team, new frameworks and libraries and many more can easily cause the same mess.

I can go on with many similar examples that prove the analogy of gardening with software development but I prefer to keep some for my book. I want to finish this introduction to software gardening by presenting the four axis of a software gardener. We don't re-invent the wheel. We know what we need and we properly use it.

Practices: TDD, Code reviews, Automation everywhere (do you know any gardener that irrigates large gardens manually?), re-factoring.

Skills: We recognize that software gardeners equally need to have technical skills like being a clean coder, write meaningful unit testing, being polyglot (choose the right soil and fertilizer) and soft skills like being patient and self-learner, avoid multi-tasking, setting goals, being proactive and ready to adopt to changes as individual and as a team.

Tools: We pick not the “best” tools but the tools that make sense to our team, our mentality, our project our company. We recommend however at least the following set of tools to be used: issue tracking systems, advanced source control management systems, wiki, code review tools, continuous integration, continuous delivery and continuous inspection systems, virtualization and like we already mentioned: automation everywhere.

Attitude: Maybe this is what it makes that huge difference between a software developer and a software gardener. Software gardeners are committed to their profession - or should I call it passion? They share their knowledge to the community. They never keep it for themselves. By the way how many bridge or skyscraper engineering conferences do you know around the world? They love being mentors, they pursue constructive feedback and they accept criticism to be better gardeners.

To be continued...

I really hope that this article proved that software gardening is not just yet another crappy analogy but a reality in software development. The world (including us) deserves much better software and it seems that we are still missing something. Maybe viewing our industry as gardening and not like engineering or craft might be the secret ingredient to success. Who knows? I believe it and many people believe it. All we have to do is spread the word, embrace the software gardening manifesto and build better software!

If you want to embrace the software gardener's practices and attitude I kindly invite you to sign the manifesto [6] and be part of this new philosophy and practical approach of software development.

References

1. Bridges, software engineering and God http://blog.codinghorror.com/bridges-software-engineering-and-god/
2. Why building software isn’t like building bridges http://blogs.msdn.com/b/steverowe/archive/2005/02/28/why-building-software-isn-t-like-building-bridges.aspx
3. Programming is gardening, not engineering http://www.artima.com/intv/garden.html
4. The Art of Software Gardening, https://leanpub.com/art_software_gardening
5. The Agile Manifesto, http://agilemanifesto.org/
6. The Software Gardener Manifesto, http://softwaregarden.io/manifesto/

Entropy for Measuring Software Maturity

Kamiel Wanrooij, Grip QA, http://grip.qa/, @gripqa

In a software development project change is one of the only constant factors. Requirements can change, as can the technical considerations and environmental circumstances. Our jobs as software project managers and engineers is largely managing this ability to change.

As software projects grow, the ability to change often diminishes. This is in contrast to the rate of change, which generally increases through the first releases until a project enters maintenance mode and is eventually *End-Of-Life*. This difference makes software projects unpredictable and has given rise to methodologies like Agile, Scrum and Lean to streamline the rate of change. These methodologies do not, however, help increase the ability of a software project to support this rate of change. Entropy is a metric that you can use to measure a software project's ability to keep up with the rate of change.

What is entropy?

Entropy is a term from information theory that is inspired by the concept of entropy in thermodynamics. In thermodynamics, and in general, entropy is a measure of disorder in a system. It's this disorder that also interested us in software development.

Claude Shannon first defined entropy in the context of information in 1948 in his famous paper: *"A Mathematical Theory of Communication"*. Shannon defines entropy as the amount of information you need to encode any sort of message. In other words, how much unique information is present in the message. If you have a coin that always turns up 'heads', you don't need anything to record the outcome of a coin toss. A regular coin, you need one 'bit' of information to track if the coin came up heads or tails. A six-sided die: 2.6 bits (yes, in entropy, you can have fractions of bits).

This concept is often used in cryptography. The 'entropy' of a password is how many bits are required to store all possible combinations of a password. A 4-digit pincode carries less entropy than 16 alphanumeric characters with special characters mixed in. In cryptography, higher entropy means that it is harder to brute-force, since there are more possible combinations.

The same concept can be applied to changes made in a software project. If a change only affects a small part of the system, that change can be recorded with very few bits of information. If changes touch a large part of a system, you need many more bits to encode that change.

Using this logic, we can determine the impact of changes by calculating the entropy that each change carried. In a typical software project, the larger part of the system you need to modify to implement a feature or change, the harder it is to implement that change. Thus looking at the entropy of past changes tells us something about our ability to make those changes efficiently.

Coupling in software

One of the most common goals in software architecture is managing coupling. Coupling is the dependency of one part of the code to another. They are 'coupled' together, either explicitly or implicitly.

Explicit coupling happens when one part of the code directly depends on or uses the other. This is unavoidable, but should be carefully managed. A tightly coupled system can become brittle and hard to change. Most design patterns that deal with explicit coupling implement some form or part of the SOLID or DRY principles.

SOLID is an abbreviation of 5 best practices in Object Oriented software development: Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion. The impact of these best practices is beyond the scope of this article, but they're all designed to help create maintainable software architectures.

DRY stands for Don't Repeat Yourself, and is an often-heard mantra for developers. Not only does repeating yourself create additional work now, it also increases the maintenance burden later on. All repeated sections will probably require the same bug fixes and changes applied to them, if the developer remembers that the duplicate sections exist!

Implicit coupling can occur when there is no direct relationship in the code between two parts, but they are conceptually or otherwise linked together. This is usually harder to detect since it requires knowledge of how different components interact to see if changes in one also affect another.

Measure entropy in software

Measuring entropy can quickly turn into a very technical discussion. For the examples in this article we're using a very simple implementation of entropy that still caries a lot of value.

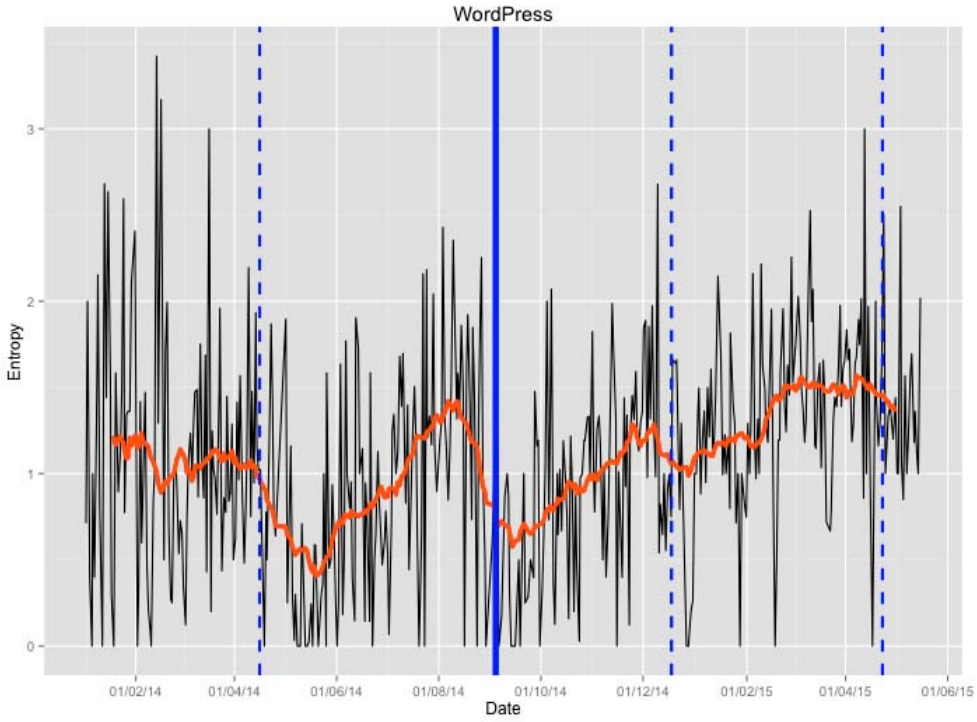
I'll define entropy as the amount of data required to count the number of files changed with each commit in the source control system. This definition is based of the assumption that each file is a logical, consistent unit that belongs together. Multiple changes to one file add just one bit of entropy to our commit. But as soon as changes cover multiple files, the entropy goes up in a logarithmic scale:

- 1 file: 0 bits of entropy
- 2 files: 1 bit of entropy
- 4 files: 2 bits of entropy
- 16 files: 4 bits of entropy
- 1024 files: 10 bits of entropy

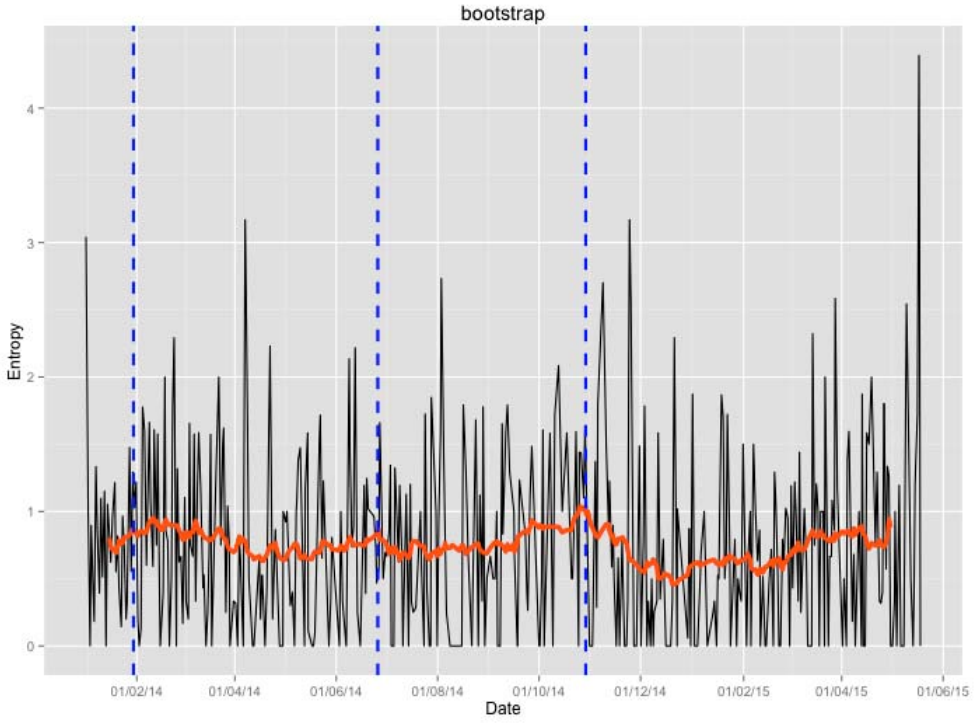
As you can see, with each bit of entropy the scope of each change doubles. This makes our definition easy to work with. It's not going to be perfect since there's obviously a lot more information in each commit than simply the number of files changed. The advantage is that it provides some valuable insights without getting too technical or too hard to compute.

Examples of entropy

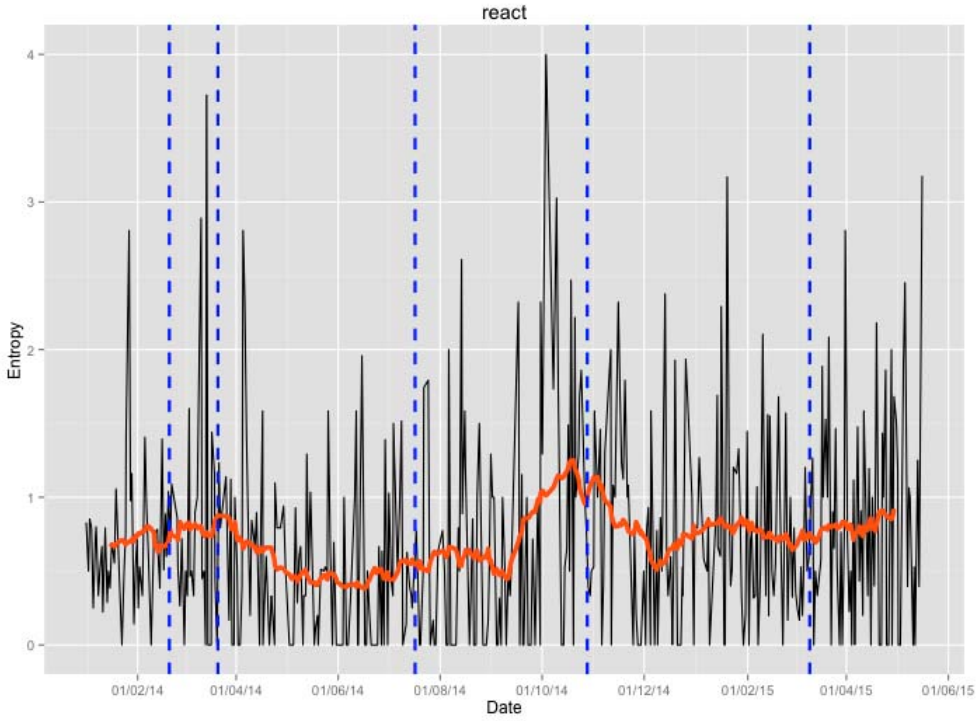
The following images contain some examples of entropy during the development process. The black line represents the daily average entropy. The red line is a 30 day rolling entropy average. The blue vertical lines represent minor releases (dotted) and major releases (solid).



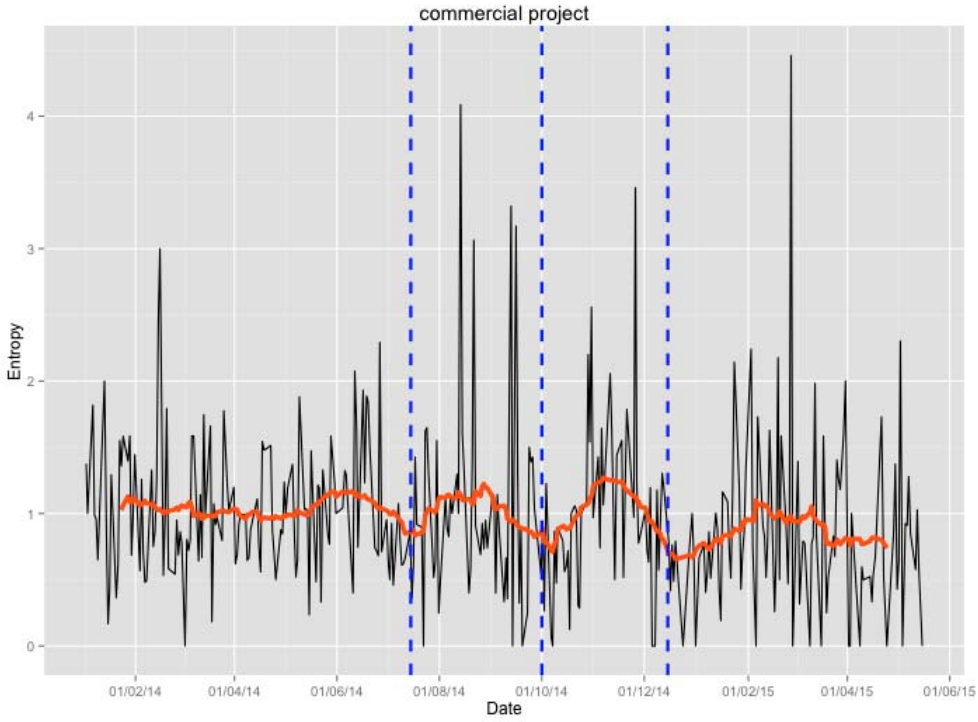
This is an entropy analysis of the WordPress repository over the past 18 months. There are a number of peaks and valleys that coincide with the past releases. Most releases happen after a peak in entropy has passed and entropy levels have started to go back to normal levels. The average entropy slowly slides upward.



We see a totally different picture looking at the Bootstrap repository. The overall entropy is quite low with an average of less than 1 bit and has very small peaks. This indicates that over the past 18 months, changes could be made very consistently. The only peak above 1 bit was quickly followed by a maintenance release that reduced entropy again.



React shows just one significant peak in the same period, with some minor fluctuations. The overall entropy is well below 1 bit, indicative of a mature codebase with the right degree of decoupling.



This is a commercial project that I worked on. Entropy was stable but begins to fluctuate slightly as the technical limits of the system draw closer. Their current goal is to stabilize at a lower level than at the start of the measurements to facilitate future change.

This is our internal product development. We had been doing greenfield development for almost a year, only bringing down entropy right before the internal test releases and our public beta release. The current entropy is still above 1 bit, in contrast to some of the more mature projects that are well below 1 bit of entropy on average.

Causes of entropy

As seen above there is a lot of overlap between the effects of coupling and the way we measure entropy. Both deal with effects spanning across multiple areas of the codebase, and indeed coupling is a major contributor of entropy.

Some of the most obvious ways coupling can introduce entropy occur when there is a high degree of structural coupling. Changing a class name or adding parameters to a method often requires all places where this is referenced to be updated as well. The heavier one file relies on these details in other files or modules, the higher the effect is on entropy.

More generally, most violations of the DRY principle can introduce higher levels of entropy. As soon as any representation, functional or technical, is duplicated across the system, it requires more effort to change that.

More subtle ways that can still have an impact in entropy are functional or implicit in nature. When two objects share some mutable state for instance, they are effectively tied together by that state. Modifying one can introduce changes in behavior in the other, requiring a second change to offset that unwanted behavior.

Apart from coupling a more desirable method that introduces a lot of entropy is a large system refactor. When removing coupling between modules initially you'll also touch upon all affected modules. Highly coherent modules, which are modules that are very tightly related by design, can also introduce more entropy.

Other ways that increase entropy, either desirable or unwanted, are writing automated tests along with the implementation; unfocussed, broad functional changes tied together in a single change; initial setup of systems or libraries.

Effects of entropy

Entropy itself is not necessarily a bad thing, but it is an indication of stability and maturity of the development process. Since coupling is a major contributor to entropy, tracking entropy is a good way to see which areas of your system are explicitly or implicitly tightly coupled. This might not be easy to do just by a static analysis of the codebase itself.

Periods of high entropy changes indicate areas that are likely to be dependent on each other. Changes to one part might introduce bugs or rework on the related parts. As the scope of changes increases, the risks and testing efforts also increase.

Even after entropy levels decrease, those high-entropy changes introduced are still present in the systems. This could affect future changes in a negative way, so a lower entropy level does not necessarily mean that it won't increase again in the future. Whenever the rate of change is low this is not so bad, but as the rate of change increases those areas will again introduce higher entropy levels.

Since entropy is both a measure of the rate of change and the breadth of those changes, it's very relevant when scheduling releases. During periods of high entropy it is highly inadvisable to release since changes still have high impact and risk. When entropy decreases, this indicates that changes are more local and thus easier to implement and test. These are the changes that can be done safely closer to a release.

Preventing high entropy

There are many ways that entropy can be prevented using software design principles such as DRY and SOLID and creating abstractions with the correct level of detail. It also helps to define your systems with proper consistency boundaries that match your problem domain.

It's always a good practice to be aware of these principles and to incorporate them into your coding habits. Team members should encourage each other to frequently revisit these principles and exchange ideas on how to improve their craft.

Not all of these things can be designed up front. While it certainly helps to understand the domain before designing technical systems, it's not always clear up front how to structure the technical systems. Overdesigning software can certainly lead to entropy later on in the development process.

As a system matures it's impossible to prevent areas that are tightly coupled together. Keeping track of this by measuring entropy helps to identify these areas before the impact of them grows so large that it introduces major risks to the development process.

Conclusion

While high entropy is not a problem in itself, it is a common result of software projects that struggle with efficiently responding to changes. A lack of some development best practices such as SOLID and DRY design frequently causes increased levels of entropy. A non-technical contributor to entropy can be an unfocussed functional specification process.

None of these are guaranteed to introduce entropy, and not all entropy is an indication of these problems. Some entropy can even be caused by very legitimate and valuable changes, but because entropy is closely related to a number of best practices, it is a valuable metric to track as part of release management.

The READ, RATT, AART Requirements Methodology

Fred Henry Williams, Williams Technical, www.williamstechnical.com

We need a better method for defining system and user requirements that includes the best of Waterfall and Agile. Using the format READ, RATT, AART for clear user and environmental specifications helps you organize, track, prioritize, develop and test systems with user and business value. Requirement specifications, test cases, user stories improve with clear writing. Combine Waterfall and Agile strengths for better requirements that provide a clear view of what users want, where, and when. This introduction to the method shows you how to organize, track, develop and deliver quality systems using READ, RATT, AART.

Waterfall and Agile aren't so starkly separated. Most organizations retain a mix of methods. Here's how it can work better, taking advantage of Agile and Waterfall strengths, and blending them together to define realistic and testable functionality.

To combine the best of Waterfall and Agile, use the acronyms READ, RATT, AART.

READing your users allows you to describe what they want, who they are, and how they'll know they got what they wanted. Share in a Waterfall style shared reference document the Results, Environment, Acceptance criteria, and Demographics for each actor in the system.

RATT states the Result required, the Actors, Test of usefulness, and Test of business value. This is a clear problem statement, with developer and QA tests.

AART is a clear statement in the simple present tense. Name the Actors, an Action, and Results. Add at least one Test of how the actors define success.

Why did we get into conflict between Waterfall and Agile in the first place?

Putting a waterfall in the background of your agile stories…

Early methods for developing requirements drew from the experience of construction and military planners. Their contribution to the field is generally known as Waterfall. This is a top down method where all requirements are defined up front, and cascade down through the organization. Typically this resulted in hundreds of pages of details which were followed by the developers, tested by QA, and finally validated through user acceptance.

The waterfall method works... but it relies on an assumption of omniscience. Somehow business analysts, architects, programmers, and project sponsors are presumed to actually know the details of the implementation at the onset of the project. Few changes are allowed, except under heavily controlled conditions. Because validation and acceptance occurs at the end of the project, which might be years after the drafting and approval of requirements, new approaches and adaptations resulting from the experience of creating the system, are either glossed over or rejected.

Organizations such as the IEEE promoted this methodology through publications, training and certifications. CMMI, PMP, Prince2, Six-Sigma, ITIL and other flavors of project management and control continue to dominate the professional qualifications of the programming world. (PMBOK)

There is an excellent reason why this method still dominates the world of systems development.

Narratives mislead, but are nonetheless essential to understanding. Stories seem to be how most normal humans organize information about the world, automatically sorting an array of objective facts into hero's quests, tales of transformation, monsters slain, mountains climbed. In this process subjective facts and assumptions, which may not be true, are invariably inserted.

To avoid such errors engineers, mathematicians, and scientists developed evidence based and logically testable methods to remove narrative. This necessary step, however, replaces narrative with pure description. If modern technological systems had no effects on the lives of humans, there would be no need to change this. Omniscient description would be enough to communicate all that must be said of the world.

Yet the failures of this approach, especially as projects grow in size and complexity, have frequently embarrassed the advocates of the waterfall methodology. Over budget, late, and cancelled projects, often costing hundreds of millions of dollars and wasting untold working hours, appear in the news regularly. (Chaos report)

In response to this dissatisfying and occasionally disastrous approach, leaders of the software development industry have attempted to modify or replace the waterfall methodology. Among the approaches we have seen Extreme Programming, Test Driven Development, and Agile.

These newer methodologies reject top-down approaches, acknowledging that there is a cone of uncertainty that can only be clarified through the process of actual developing the system. In short, narrative matters. Objective facts change over the course of development. Reasoning that it is better to recognize this fact rather than cover it up, Agile rejects documentation-first approaches, favoring instead an iterative and adaptive method of delivering functionality, reviewing feedback, and changing approaches in response to user reactions. In a widening spiral, functionality is delivered according to what users value and what organizations can deliver quickly.

Empathy is the catalyst for valuable Agile requirements. Understanding and feeling for your real world actors allows you to create something they buy and use. There's usually a minimum of two actors, the user and the buyer. They want different things, and both must be satisfied with the results you offer.

The READ, RATT, AART methodology provides a practical method to improve empathy in Agile user stories, while also taking advantage of the Waterfall method of listing non-negotiable requirements.

There's no need to create elaborate background stories for your actors. Avoid the mistake of giving irrelevant details. You want to provide just enough information about their roles, experiences and expectations so they're realistic and useful.

Describe their environment in Waterfall format, with lists and tables. It's useful to list the real-world constraints that are non-negotiable. These parameters include where they work, the technology they'll use, devices and locations, legal requirements, and so on. It's a waterfall type list, with the most important criteria listed first.

Then turn your attention to their goals. What results do they want? What is acceptable? How can they know they got what they expected?

This approach also has its critics. Agile assumes that the end-user is the primary customer for a system, and that all functional requirements can be negotiated and trade-offs defined with heavy user involvement in the definition of the system. Narratives from the user perspective, heroic tales in the form of user stories, are presumed to be the only required view of the system.

In fact, few end users have the skills, time, or interest in becoming development partners. So programmers end up making guesses about what is required. The stories aren't told from the first person perspective, but a 2nd or 3rd person view. Essential description of time and place, environmental factors and constraints, get left out.

Examples of Agile methodology failure include systems where regulatory and legal requirements are defined through user stories, rather than explicitly stated through simple description. If you are developing an accounting system for a government agency, you cannot expect the end users to have sufficient knowledge of the underlying practices and expected levels of performance. Capturing this information in a user story is unrealistic, leading to worse problems than simply stating the requirements in formal Waterfall language.

So any narrative alone, embedding human assumptions of heroes' journeys, and leaving out description of the landscape where the story is set, must fail to capture all the requirements. It's like a movie where you only see actors performing against a green screen. We must include the Waterfall in the background, explaining the time, place, and constraints of what happens to the actors in the Agile story.

Those who actually make systems, well-schooled in logic and the world of inconvenient facts, know this already. We have never accepted a waterfall-only or agile-only view of our work.

The result is that few (if any) organizations are only "Agile". Most seem to have instinctively created a hybrid of the methodologies, what is jokingly refered to as "Watergile". Sometimes this is superficially a renaming of waterfall artifacts with agile terminology...a true waste of time and effort which delivers nothing of value while creating confusing and chaos. Other times old waterfall requirements are forced and twisted into the agile methodology, and what was once a clear requirement becomes vague.

Clearly we all need to recognize that all methodologies have strengths and weaknesses. To strive to have perfect knowledge before a project even starts is ridiculous. And to be so adaptive and iterative that few predictions of cost, schedule, or even functionality can be reliably delivered goes against general realities of the business world.

After decades of attempting to define system and user requirements, using various waterfall and agile methodologies, may I suggest an open compromise which blends the advantages of both methodologies? With our methodology we recognize the primacy of user requirements and the advantages of stating these in narrative stories. We also recognize the real-world situation where non-negotiable requirements can never be ignored and must be incorporated into the requirements specification in a way that the developers can meet both user goals and environmental realities.

Most task tracking tools provide shared reference pages. Whether as a wiki or a document, use these to describe who uses your system, and why they want to use it.

If it makes sense you can create a Universal Environment. Like a novel, describe the non-changeable system in terms of constraints. All actors in the system inherit universal environmental constraints. Exceptions are specified within an actor's description.

Actor descriptions, in READ format:

- Results
- Environment
- Acceptance
- Demographics

Actors have background and environments that allow or require individual results.

Use these pages as references for all Agile user stories. These are the Waterfall requirements for testing. All user stories inherit Univeral requirements. All user strories specify one or more actors, and describe what the actors want, and how they can get it.

Describe environments of your stories with Waterfall

Let's look again at how our brains structure stories. At the beginning, and continuing throughout a well-written story, the author adds and expands on details of the setting. This descriptive prose is presented either from the viewpoint of the protagonist, or by the omniscient narrator. The best stories often include both views.

Within any setting, a farm house, a magic dragon's lair, a space ship hurtling through uncharted galaxies - the story must include actors doing something interesting. That’s where user stories are essential. But without the setting, the stories lack context.

So when writing requirements specifications, include both waterfall style explanations of the hard-facts which will not change, the time and place, physical characteristics, assumptions required to impose order on the system in the real world.

There are two types of description, 1st and 3rd person. So while defining the system, you can organize descriptions according to universal (3rd person) and variable (1st person) descriptions of environment.

Universal descriptive requirements cannot be broken by any actor within the system. These include system constraints and underlying architecture, integration with other systems, and limitations of what current tools can provide.

Individual descriptive requirements change according to the specific user. Mary at the front desk does not have the same level of access to a system as the CTO. A remote user cannot have the same performance expectation as the system administrator. Each actor's perspective affects their experience of the system, and how they use the system to perform their story.

Each requirement specification references the universal environmental constraints. (QA validates this in scripted testing.). The specification references the named actors description. These actors perform actions within this environment. Use READ to format these actor descriptions.

Each actor has required results, motivations to perform the actions. Actors get results from their actions. The story specifies which actors are doing what, and the actor's background and circumstances show where, how, and when they perform their part. Use RATT for describing functionality. These can be sorted, with actors added and removed. For developers, these are functional requirements. For product owners, these are business cases for user value, and profit. If the team cannot articulate the Results, Actors, and Tests of value and profit, it's an early and clear warning that you must further define the requirement.

Using the structure of READ, RATT, AART to structure and validate user and system requirements.

Be the author of your requirements specifications by doing what authors must. Describe the setting and introduce the actors. You don't put all the details in the first chapter, but build and modify as you write. So in the first iterations of your specification, you put down the big picture.

Where you state your universal environment is up to you. It can be a section of a document, a wiki page, a set of metadata. What matters is that these statements rarely change, and are true for all actors wihin the system.

But don't impose too many of these universal requirements at the beginning. Instead, quickly move the narrative of your system forward by introducing the characters and what they want.

If your universal setting is a magical kingdom, when you introduce the actors the first thing you'll do is state whether they are kings, or wizards, knights or peasants. Is the knight rich or poor? Does the wizard work in a cave or in the castle? Is the peasant struggling with a mud hut, or a happy villager following the ancient traditions? Does the peasant's story lead to marrying into the royal family, and moving to the castle?

Users do not often care about the underlying process the system uses to get the results they want, any more than most drivers care to know the details of how their car's fuel injection system operates, or a fantasy narrative explains how the magic wands works. Our hero only wants to win the road race, or incant the right spell to transform a frog into a prince

So we start first by defining both the desires of the end user and the realities of the environment in which they will operate the new system. The results we deliver will continually evolve and expand over the life-cycle of the project, just like a novel develops and expands as you read, but the environmental constraints change slowly if at all. Most novels don't shift in the middle from slice-of-life to space opera, even though good novels do shift the experience of the characters, resulting in changes of their local environment and achievement of important goals. So it makes sense to differentiate between these two types of inputs into the requirement specification. What remains the same, what changes within the perspective of the actor, and what does the actor do to earn that shift of perspective

READ your users

Define your users first within the parameters of **READ:** Results; Environment; Acceptance; and Demographics. Each actor who uses the system, including non-human actors like servers, APIs, and regulators can be defined according to READ. System outcomes can be sorted acording to named specific actors. You can also sort specifications according to shared environment factors, or the expected results. By shifting the perspective between these views, you gain clarity into overlapping requirements and constraints.

Name your users. Define user demographics and environment, then determine what results they desire along with what makes that result "alright" or acceptable. If the user is another system, use the same format and name the system. In practice, these Actor descriptions appear to be persona descriptions. You can use real people, archetypes, personas, or even systems that act in your environment.

Whether you include a photo is up to you. The level of detail is up to you, and should grow and be trimmed over time. Define the four READ elements.

Results: Begin by stating the results each end user desires. Limit this to the most important outcomes the users expect. Trying to guess all required results up-front is not recommended. Better to define what matters most, and deliver that first, allowing your stories to grow over time. Use the present tense, active voice, and always start with the name of your user. For the most specific users, create lightweigh user personas in accordance with the principles defined by Alan Cooper. (Inmates Running the Asylum)

- **John** sees daily reports showing how many people have visited his web site.
- **Mary** sends weekly messages to her grandchildren.
- **The system** sends notifications of system performance parameters to **Richard** via text message.
- The **file database** saves all changes to files.
- **Chrome** browser displays the content of web pages.

Environment: For each user of the system, list the physical and technical evironment where the user will receive the results of your work. What is their operating system, bandwidth, regulatory and legal matrix in which they must use the system? This is unlikely to evolve rapidly, so there's no reason to put this information into the form of user stories. Rather you can create a list, table, or collection of links to outside definitions of the constraints the user experiences. This information resides in your user descriptions.

- Win7, broadband connection, FireFox browser.
- Open office plan. Noisy. Frequent interruptions.
- FDA regulatory oversight. Sarbanes-Oxley financial reporting.

Acceptance: Acceptance criteria of the users can be defined in qualitative terms. It's not required to declare what the perfect result will be. Rather, define what is "good enough" or meeting their expectations of how a system should work. As the peasant character in our story may be satisfied tilling his fields, as he progresses through the narrative of marrying into the royal family he'll become more demanding. This can be stated in terms of comparison (result delivered at least as fast as our competitor's product) or absolute numbers (result delivered within .003 seconds). As these expectations change, you can update these parameters. What you want to define is the condition that would prompt the acceptor of the work say the feature is "alright". Performance beyond the "alright" condition is likely to be wasted effort, over-kill that won't be appreciated.

- Faster than our previous system.
- Includes profit-loss statement daily.
- Does not prevent me from using another system simultaneously.
- Works on existing hardware.
- Costs less than 20 dollars per user per month.

Demographics: For each user of the system, define their demographic information. This does NOT require excessive information. If you are creating Personas to stand in for actual users, it's certainly not helpful to developers to find extraneous details irrelevant to the project and the results desired. It is enough to state the user's age, profession, education, and experience with

similar systems. It's not helpful to include the user's imagined hobbies, family details, and the names of their pets. Only demographic information relevant to their role is required here.

- John Harris: 35, systems admin, MSDN certified, full time employee.
- Mary Smith: 67, retired English teacher, uses public library computers.
- Richard Hall: 43, project manager, 18 direct reports, Prince2 certified.

Kings in a magical realm have unique demographics. So do wizards, thieves, knights and knaves. When you tell a story, it's important to show that the characters all have a back story that explains their motivations. Demographics provide this information in shorthand, helping the developers understand the who, why, when, where of the actors using the system.

If this information is not included, or is false, the system is highly unlike to produce the expected results.

Describe proposed functionality as RATTs

All characters in a novel have a motivation. They want to find the grail, rescue the victim, climb the mountain. Without this motivation, the character has no reason to participate in the story. If there's no rat to eat, the cat will only sleep. To make it easy to remember that these motivations, or problems, are the reason people will use your system, it's instructive to think of functionality as rats.

After you have defined your system users with READ, describing the users, their setting, and their motivations, you can describe and analyses proposed system functionality using the acronym RATT: Result, Actor, Test of user value, Test of business value.

Result. For each result you listed in the user descriptions, create an instance of proposed functionality. Be specific enough to easily create a test, from the user's perspective, where they can know whether they have received the result desired. (If you cannot easily create such a test, you need to find a better understanding of the requirement before proceeding.)

Actor. Name the end user. Link to the profile that contains that user's READ information so that all environmental and acceptance information for that particular user is part of the functional description as well as the result. This is how waterfall constraints (performance, reliability, speed, legal and regulatory needs) blend with agile user stories. The environment of the individual users are a key part of every user story, and cannot be violated.

Test of user value. State how the user knows they got what they wanted. State how the user values this result. Try to include monetary, competitive, perceptual, emotional, and financial measurements to define the value the the user places on this feature. High values increase the priority of the feature.

Test of business value. How will adding this function to the system benefit the business. State this value in terms of money, time saved, capabilities expanded, or strategic goals met. If you cannot easily state at least one benefit to the business for developing a feature, question whether the result is worthwhile developing or should be abandoned as unrealistic or unprofitable.

Note: No proposed functionality should be defined any further unless both tests of user value and business value are defined and agreed to by the customers, development team, and business representatives.

This is your functional description. What do specific users get. When do they get it? Why do they want it? Why can we give it to them?

Some functional RATTs

Result: Barrack reads intercepted communications.
Actors: Barrack Obama, (President of the United States, the White House)
Test1: Barrack can discover suspects.
Test2: OurCompany can sell this service to the NSA for Barrack's use, at 2.5M per month.

Result: Alice sends a message to Bob that Charles cannot read.
Actors: Alice, Bob, and Charles (Encryption Hobbyists, Coffee Shops and Data Center)
Test1: Charles cannot read intercepted communications.
Test2: Alice and Bob will pay $5 per month for this service.

Result: Peter creates and prints a TPS Report.
Actors: Peter (IT worker, Office Cubicle)
Test1: Peter delivers TPS Report to Lumburgh, including cover sheet.
Test2: Our IT support division is not outsourced for cost cutting.

Write user stories as AART

The important outcome of organizing information about the users, the results they require, and tests of user and business value is that it allows you to write clear specifications that can be used by developers to deliver high quality targeted functionality to your users. QA can also use these stories to demonstrate passing or failure.

To ensure this information is clear, it's important to use simple language.

The simplest form of English is the present tense active voice. The form is Subject, Verb, Object. For example, "Fred drinks coffee", or "Alice sends a message to Bob."

In Agile user stories, you should name the actor or subject first. Then describe the action the user takes to achieve a result. Follow this with at least one test, so you know when the result is delivered.

Actors:

Name the actor who performs each action. This actor is involved in a situation with a RATT. There can be more than one actor in the scene, as when a system administrator helps a user to log on to the system, or a wizard transforms a peasant into a knight.

Action:

Name an action within the system. Sir Pendelton moves into the castle, the Active Directory authenticates and authorizes a Richard, Susan, and Barrack so they can view files. The Sir Pendelton is ushered into the royal court, or Susan prints visitor's badges with names and photos.

Results:

Describe how the actors know what has changed. The receptionist Susan (link to Actor description) confirms the name on the badge is the same as the visitor's, and knight Pendleton (link to Actor description) enters the castle to see the king waiting on his throne.

Tests:

You can include as many tests as make sense. If you find you're including more than 7 +/-2 tests for a user story, you should probably break up the AART story. At a minimum you should include a test of user value. You may also want to include a test of business value for each AARTful story.

The knight Pendleton sees the crown and scepter, proving the king is on the throne. Our receptionist Susan can see the calendar and all scheduled visitors along with their details.

AART is a continuation and re-arrangement of the RATT. It's a change of focus, when the development teams describe the Action. Large (more than one sprint) AARTful user stories are Epics with related user stories. Large tests can be promoted to stories, and small stories can be demoted to tests.

(First, Copy the Actor, Result, and Test-A, and change the order.)

Actor(s): Personas/Customers have a name. Use name to link to the persona description/customer interviews.

Action: What does the actor do with the system. Explain interaction at a high level.

Result: What does the actor get at the end. The result is a "real-world" consequence, valuable to the actor.

Test-A: Acceptance test. State the user value. How does the actor know they got that result? This is a black box acceptance test.

From RATT to AART -- Barrack's Epic

The RATT product description:

Result: Identify suspects.
Actor: ***Barrack*** (link to Barrack description and environmental constraints)
Test-A: *Barrack can read profiles of suspects* (link to suspect description list).
Test-B: Barrack pays DefenCorp $2.6M monthly for reports.

Becomes AARTful user story:

Barrack reviews selected TELCO records which identify suspects.

Test-A: Barrack sees only suspects matching selection criteria.

Additional tests:

Test-A: Barrack uses a secure computer (link to description) to see suspects (link).
Test-A: Deliver suspect identify reports to Barrack every 24 hours.

Test-A: FBI (link), CIA (link), and NSA (link) search and sort suspects (link).
Test-A: Sample TELCO records (link) identify suspects contactsID (link to 3rdDegree API).
Test-A: Use live TELCO metadata to sort/search and export contact and activity records.

Test-B: DefenCorp can sell same system to Vladimir, Jong Un, and Bandar?
Test-B: Barrack gives TELCOs immunity from law suits in trade for metadata?

Using AART to describe the Actor, Action, Result and Tests, we can see what matters most. Before we jump to design details.

This example is an EPIC user story. It cannot be completed within one sprint. We must add acceptance tests, then convert them into smaller user stories to clarify and constrain this Epic. The existing tests should lead to additional AARTful user stories which complete the epic.

Prioritize functionality at the RATT level. When it's time to develop, use AART to define user functionality and specific tests for acceptance.

Let's promote one of Barrack's acceptance tests into valuable AARTful user stories.

Test promoted to story: TELCO records (link) identify suspects contactsID (link to 3rdDegree data standards).

RATT:

Result: Richard searches and sorts suspects' metadata.
Actor: Richard Peters (link), NSA analyst
Test-A: Richard searches TELCO records by name, phone number, or IP address, and receives sortable suspects report.
Test-B: Essential functionality for operation to deliver to Barrack (link).
Test-B: Can Richard use existing 3rdDegree UI (link) to search/sort by suspects TELCO metadata (link)?

AARTful user stories:

* Richard searches for Osama (link) through contactsID, to see a list of suspects.
Test: Richard types contactID. Richard sees matches for Osama's contactID display. Richard uses the data for analysis, sorting, and creating reports.

* Richard analyses the list of suspects.
Test: Richard sorts the metadata to identify patterns.
Test: Richard uses graphical interface to represent patterns.

* Richard cross-references Osama with DreadPirateRoberts.
Test: Richard uses search filtered metadata to identify patterns.
Test: Richard uses search filtered metadata to visualize hypothetical patterns.
Test: Richard saves metadata and visualizations as reports for Barrack.

Tests can be moved up and down in priority, and they can be promoted to user stories.

Similarly, during sprint grooming and project planning, user stories can be demoted to tests.

Non-negotiable Waterfall type requirements can be stated as tests, and constraints are included in the Actor's Environment description. Your Universal Environment specification is tested through scripting.

Defining what is important

No one except the creator really has the goal of using technology only for itself, especially when it's technology in the form of new software. What other people want is a result that is valuable. If the result comes in an innovative form, we may consider it a world changing project. But no matter what, we want a real-world result.

Use READ, RATT, AART to define what is useful and valuable.

Development teams deserve better information in requirements, especially tests of the user and business value of proposed system features. Please do not prescribe a solution to the developers. Within the RATTs to AART process, there must be no specification of how to meet the customer's goal, only a testable description of the desired end result. Describe the magic spell, but not how the magic works!

Understanding your user's real-world goals is essential to creating requirements specifications. Reliably making something that gives someone a specific result is valuable. Delivering value lets us get paid for our efforts, as well as giving us a sense of pride and accomplishment.

READ, RATT, AART will not resolve all the problems of developing new products in the world. But openly combining both Waterfall and Agile requirements into a three level definition that provides a testable view of the world is a compromise that will reduce the documentation burden while improving developer and QA understanding of what is important for delivery.

References

SDLC high level view http://en.wikiversity.org/wiki/Technical_writing_sdlc

AART of Writing Requirements (32:11) https://vimeo.com/78057283

Original AART Proposal http://agileprague.com/pool/vzor/upload/AARTproposalDraft4.pdf

Writing AARTful Requirements (00:31) https://www.youtube.com/watch?v=3ikkOSqtKQU

How to Sabotage Projects, Part 1 (43:45) https://www.youtube.com/watch?v=Fm9_lA4bjaU

About Personas http://en.wikipedia.org/wiki/Persona_%28user_experience%29

Creating Personas http://www.usability.gov/how-to-and-tools/methods/personas.html

Lean UX in Public Sector? - Part 1: Deciding Our Way of Working

Anna Forss, Erik Nilsson, Sebastian Lejnehed, Jenny Wilander,Försäkringskassan

I guess Agile is not what you think about when you hear “public sector” or “government agency” and if you think that it’s not possible to go there, you are definitely wrong.

Försäkringskassan is a Swedish government agency, responsible for the Swedish Social Insurance. In practice, we handle 1/4 of the Swedish incomes! Our customers come from every corner of the Swedish society as even the crown princess of Sweden receives a monthly income for her daughter.

For the past four years, Försäkringskassan has gone through a major transformation program with Lean as a business strategy. In parallel, Agile has started to grow as a way of thinking within the development of IT solutions. The customer-facing e-services have been driving the Agile implementation, as we thought that we could try to take the Agile journey to the next level. With a project management method focused on long extensive pre-studies, we wanted to challenge it by testing Lean UX and conducting a pre-study that would produce user tested prototypes rather than text documents.

Lean UX is a concept developed by Jeff Gothelf, using the Lean Startup Concept as the basic principle for developing UX. Working in a consistent way with rapid experiments and real tests, Lean UX enables development teams not only to develop wonderful user experience, they also develop a way of thinking where the experience is based on actual customer insights.

The pre-study

The goal of the pre-study was to prepare Försäkringskassan for the next generation of the web site. The main purpose was to make the web site and its services available to mobile users, but we wanted at the same time take the next steps towards quicker time to market and life cycle management of both content and design components.

The team was composed of about 10 people with varying competencies. All Försäkringskassan projects are sponsored by a project owner who is responsible for outside communication and for the overall project vision. The product owner, who works more directly with the team, makes this vision concrete. The project manager plays also the role of Scrum master the design team as well as the development team. The design team consisted of three interaction designers and two art directors. The team also got a lot of input from the head of UX and visual design, a web communicator and a representative from the Communications department. This group focused on ensuring that the team followed the overall visual and communicative profile and tonality.

Methods

I'm happy to say that all team members agreed on a scientific Agile method for developing the new navigation. Since Lean is the business strategy for Försäkringskassan, we can always go back to the basic PDCA thinking. The work was conducted in three iterations. The final objective was to have a hypothesis for the navigation that was crisp enough for us to go into the implementation phase of the project. Step 1 was about identifying what was needed but even more important: what could we leave out for the implementation phase.

Key values

User input is king

The user input was crucial. We needed to work with real customers from the very start and we wanted to see them using real devices during their tests. Hand in hand with this requirement, we did not want to design anything that was not possible with responsive design so the team started designing directly with a tool that created clickable prototypes that can run on a real responsive web site.

The team decided that we needed to identify a target scenario and target group for the basic navigation of the whole site. It was not necessary to specify the navigation within services in these early stages, but if we didn't have an idea on how you move around on the site in a general way, we were sure to go in the wrong direction in the project. Since the information we have on the web site is complicated, we needed also to understand how individuals could find specific information about a specific area on the site.

One of the major challenges is that everyone in Sweden can be a customer of Försäkringskassan. This means that basically everyone is within the target group but you can't test for everyone so early in the project. We decided that since a large portion of the customers are parents or parents to be, this was a very important target group and if we could meet their needs, chances are that we could also meet the needs of the rest of the customer segments. The team therefore decided to focus on parents in these early tests.

Read up on your web statistics and use it to formulate questions, not to get arguments for your assumptions

The web statistics also became a foundation in these early stages. As we tried to answer questions directly not with guesses but actual values so that we could all see that some of the assumptions we made were right. Some assumptions could also be verified and could therefore be excluded from the actual tests.

Iterate iterate iterate

If you can hasten the user input - do it. If you can validate your assumptions quicker - do it. By shortening the feedback loop, you would ensure that you did not walk in the wrong direction too long and you would not fall too much in love with your ideas before testing them with real users.

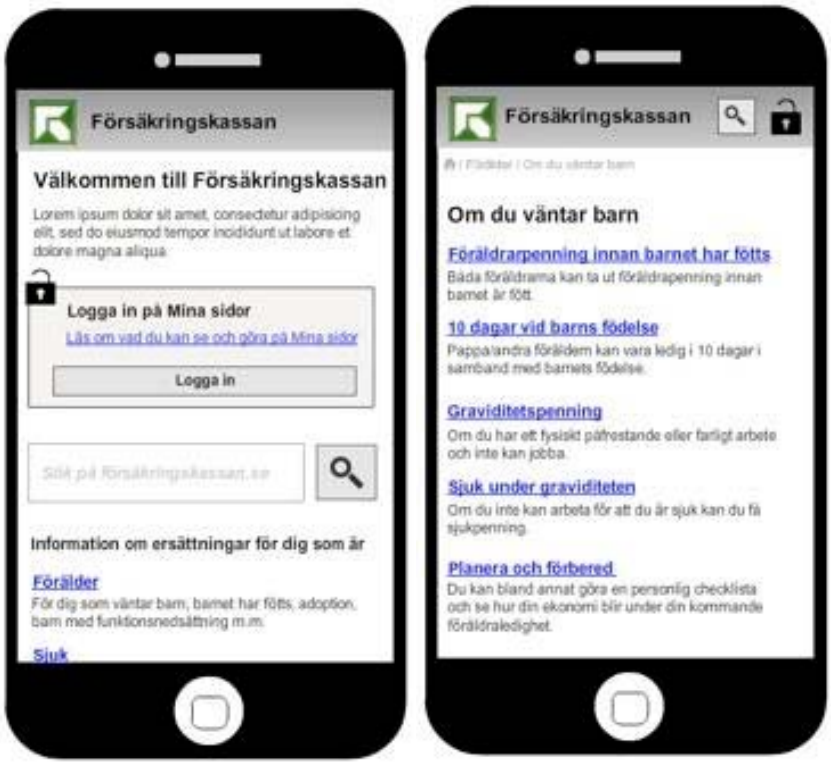
Iteration 1 - the first hypothesis

For the first sprint, the hypothesis was that at least one of the four navigation concepts we found from evaluating large sites that are often used by our customer groups (people living in Sweden) would also work for navigating the Försäkringskassan content.

Four concepts were defined:

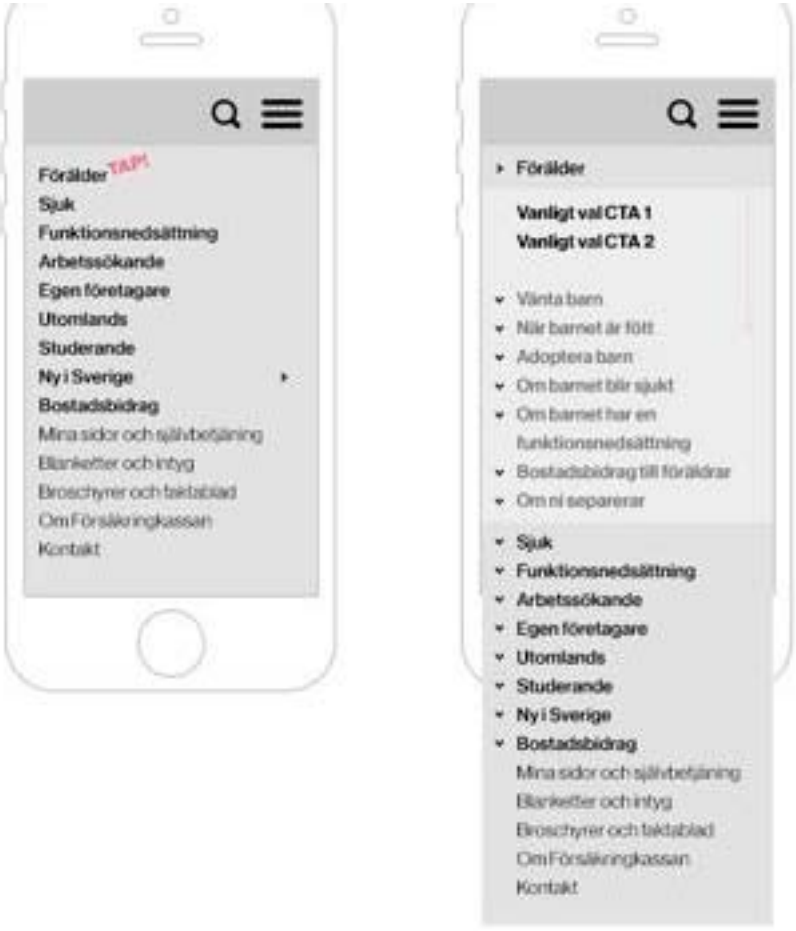
- Search based navigation
- Menu based navigation with one level
- Menu based navigation with two levels
- Menu based navigation with full depth

Below is the first concept: search based navigation. The upper section of the page shows branding but the often-found menu option is lacking. On the left side: start page. The search bar is very prominent. On the right side: A top level page.

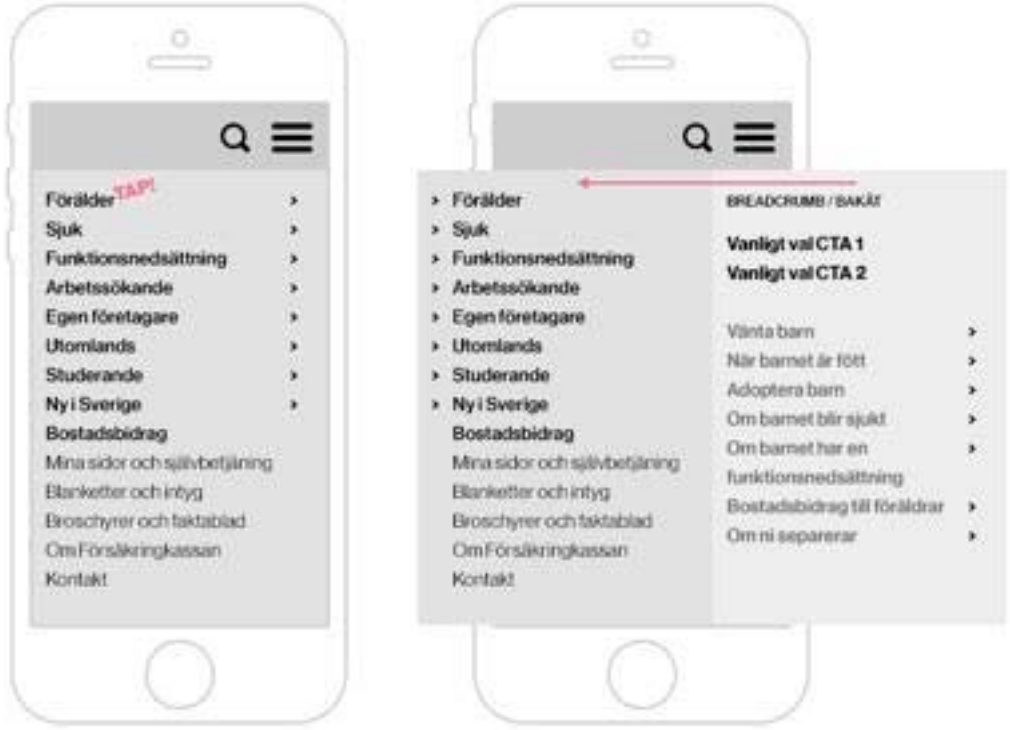


Below you can see the second concept: menu based with one level. In this case, the search icon and the menu option can be found next to the branding on the very top of the page. The search option is not prominent on the start page. On the right, the menu can be seen as it would look like when clicked. The hierarchy of the content is very deep but the menu just shows the top level. The other two concepts included lower levels of the hierarchy in the menu but the principle was the same.

In order to test the menu bar, the team also realized that there are two conventions for how to show multiple levels. One option is to expand the menu down:



Another option is to expand the menu sideways:



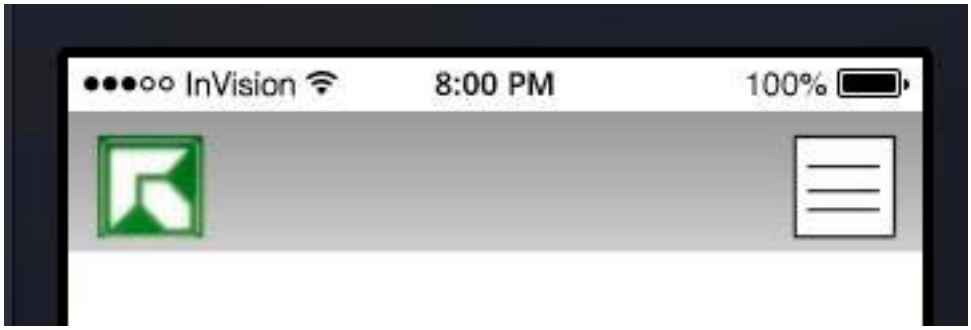
Conclusions

The most significant conclusion for this iteration was that these very early tests were really useful for the upcoming work. A lot of facts turned out to be assumptions and now the team could focus on these assumptions early in the process. The team decided to work with both the menu and the search/content concepts but narrowed the work to one concept with the menu bar.

Iteration 2 - Understanding the real questions

The second sprint was based on the findings from the first iteration. I was happy to see that the team did not just accept the findings from the first iteration. Instead, they sought issues with their own methodology. They found for instance that the test subjects who did not have the menu option in the top bar looked for it, but when the menu was there and identified as a menu, it was used a lot by the users. Was the menu useful or not? To answer this question, the team decided to not abandon the menu option and instead to fine tune the test case. It is easy to just jump to a conclusion from a usability study but often you need to validate the results first.

Below: The top menu in one of the prototypes from sprint 2:

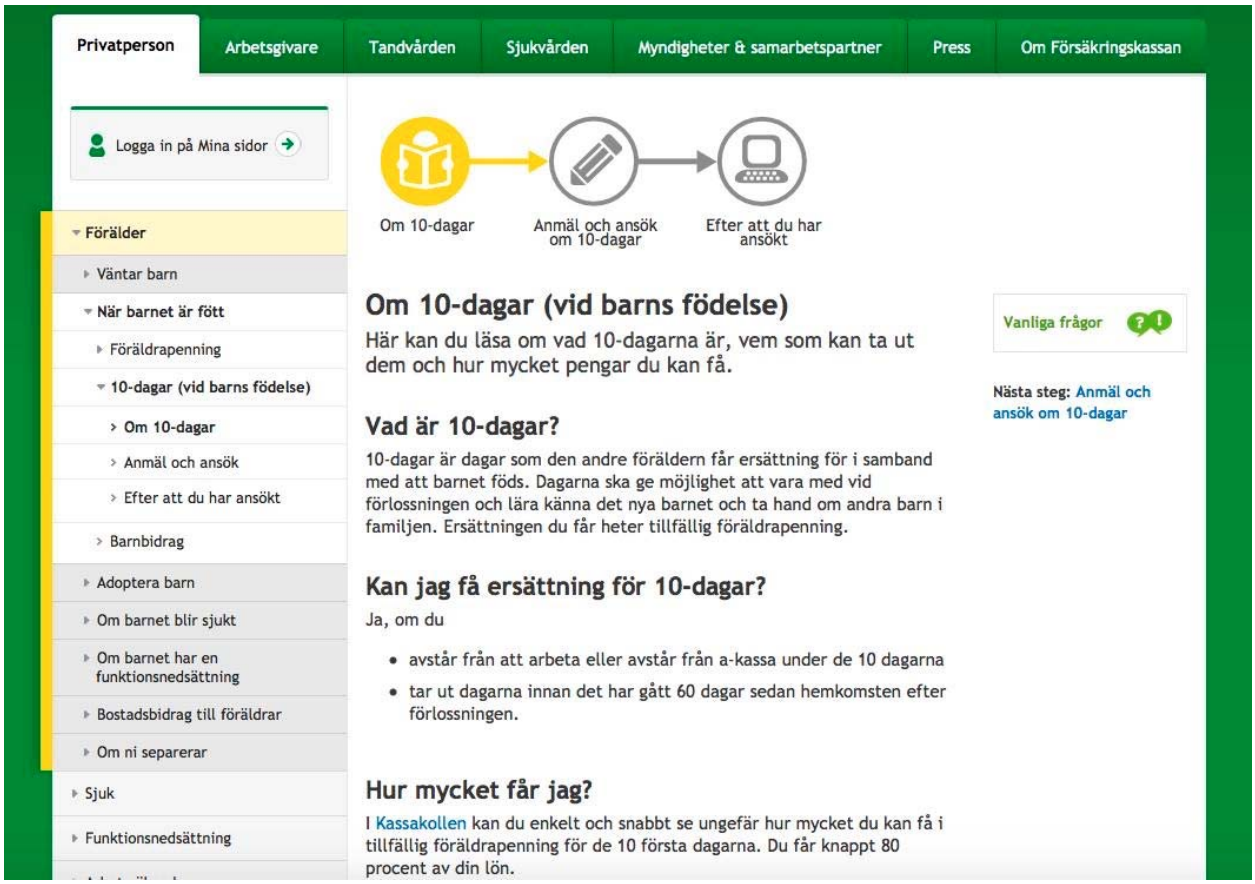


Since the team had not found that the testing of the number of levels gave that much at this point, they decided to work with just one version: a two level hierarchy.

Below: A two level hierarchy menu. The structure of the information means a deep hierarchy, but during the tests only the upper two levels were displayed in the prototype.

The team also decided to include a restructuring of the content of one of the pages for the test. A challenge within the project is that the content is derived from the laws governing public social insurance in Sweden. This means that the actual wording cannot be changed that easily and that there can be many consequences when doing so. Since the content has been rewritten not so long ago, the project had therefore decided not to change the actual wording on the pages but the structure of the content.

Below: the upper part of the current page on benefits for parents the days after the birth not yet adapted for mobile users.



Below: the upper part of the wireframes for page on benefits for parents the days after the birth.

Some interesting questions also surfaced during the first test iteration. Here are the highest prioritized:

- Why were the breadcrumbs not used for navigation?
- Did the information structure affect the lacking use of the menu option?
- What is the ideal size of top bar and content?
- What should be found in top bar, content area and footer?
- How do we ensure that the user knows where he is in the structure?
- How should a page be organized in order for the users to find the information?
- How should information be presented

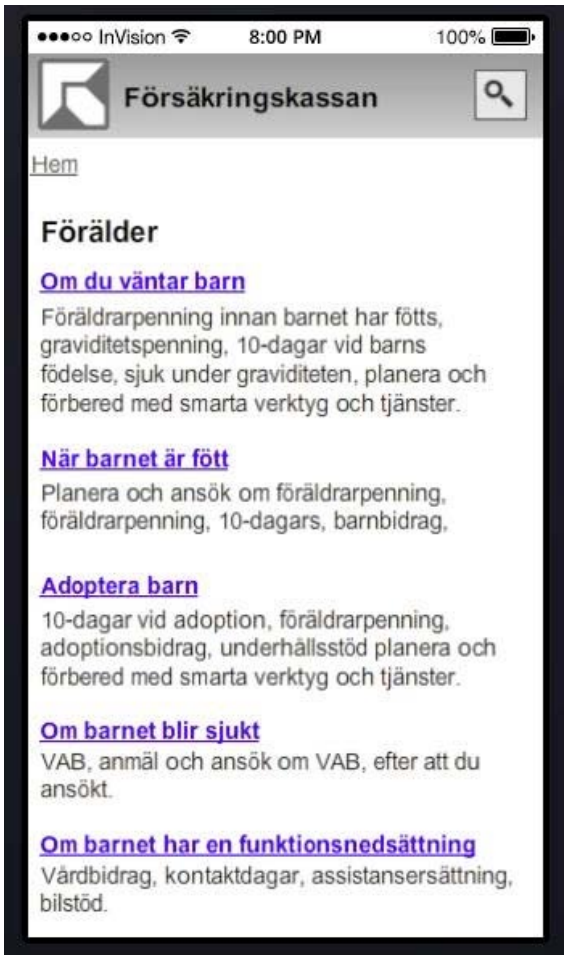
In order to test these questions, we needed a richer prototype. A digital tool for interactive prototyping was introduced and the team created the two concept prototypes to prepare for the new usability studies. We would now be able to see with an external usability study if our assumptions seemed more likely or if we would have to go start all over again. It is however better to know that after 5 weeks than a week before launch (or even worse: after launch).

Before the second set of usability tests, the team made some changes to the prototypes.

Concept without menu in top bar

The concept now included the following pages:

Below: Start Page with entry points to different life situation. If the customer is in the life situation of parenthood, there are multiple options, which can be seen on the right side of the page.



Below: Third level in hierarchy on the left side. As you can see, the information just for the future parents is large. This is one of the challenges to be tested in the pre-study: can we make the customers understand where they are and what to select among all the possible options. On the right side the upper part of the page for benefits for parents around childbirth. The links are anchor links to sections on the lower part of the page.

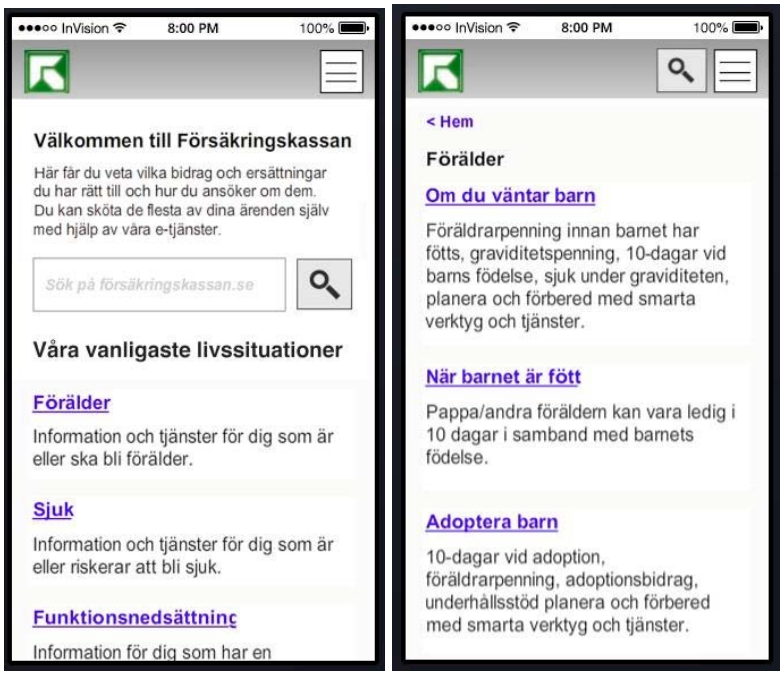
Below: If an anchor link is clicked on the page about benefits around childbirth, the customer can read the rules that apply to this benefit. As you can apply for this in many ways, the two major forms are highlighted as calls to action, while there is a link to the less common options. The reason for highlighting two options is that one requires a e-id, which not all customers have but which is recommended for a complete digital service.

Concept with menu in top bar

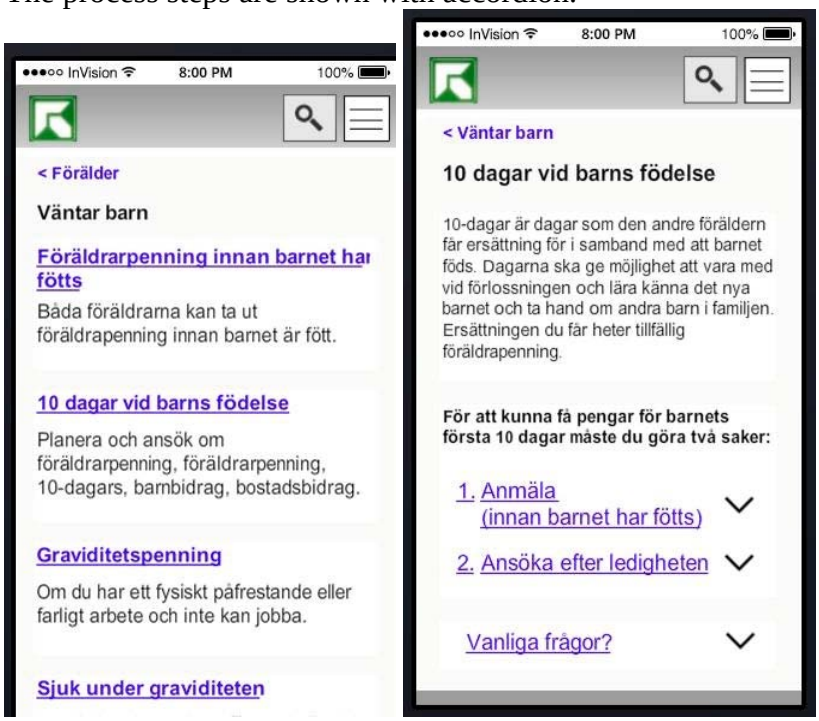
The following changes were done:

- Clearer entry points on the start page
- Fewer entry points on the start page
- The breadcrumb only showing the level above the actual page
- Accordions used to hide details

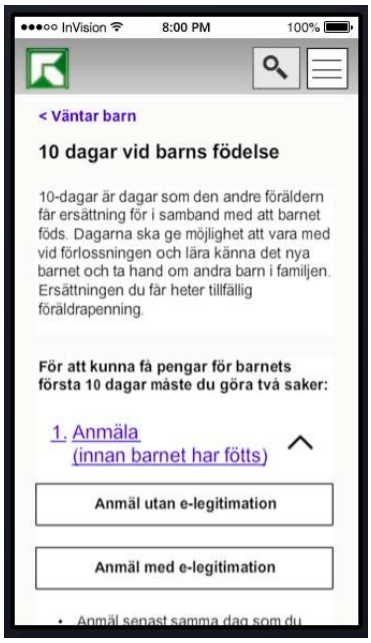
Below: Start page and page about benefits for parents.



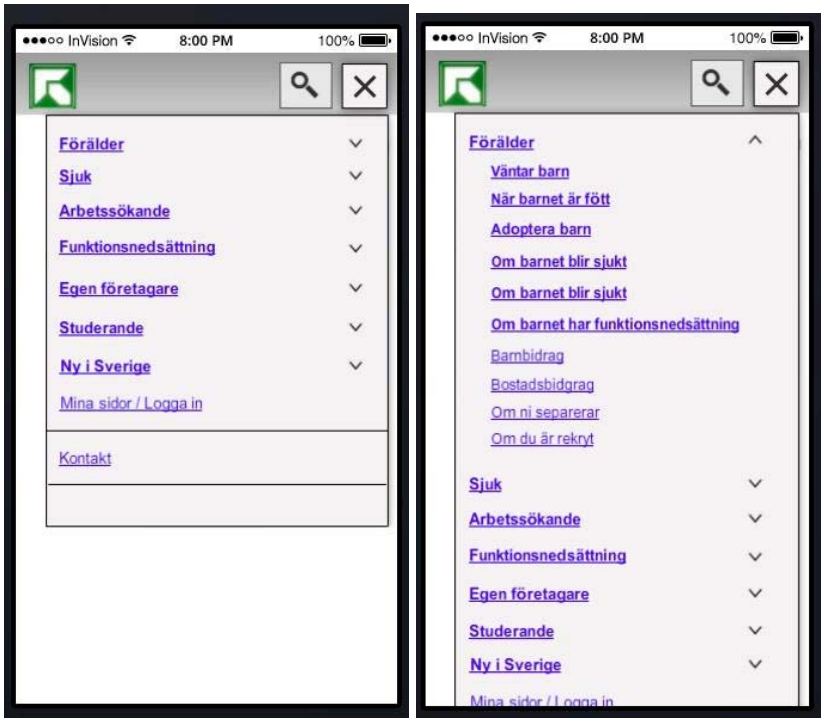
Below: Page about benefits for future parents and page about benefits for parents around birth. The process steps are shown with accordion.



Below: Page with accordion expanded so the calls to action are displayed



Below: Top bar menu option tapped (on left side) and expanded when a top-level option is expanded (on the right side)



The purpose of the second round of tests was to understand how users preferred to navigate on the web site. A challenge for Försäkringskassan is that there is a lot of information and most of it is about a specific form of benefit. As the rules are complicated and very specific to different groups, there is a challenge how much of the variation should be displayed to all customers, how much is to be found in detailed sections and which customer groups have such a complicated unique situation that they should contact the service centers? Additionally, since a substantial part of the customer group is eligible for multiple benefits, they need to be able to navigate between sections easily.

The team focused the tests around the following questions:

- Is a menu option in the top bar useful for the customers?
- Are breadcrumbs used and how?
- Is the logo for Försäkringskassan used and how?
- What is necessary for a customer to understand what is displayed when a link is clicked? This might seem like an odd question, but since the customers are often unsure about the benefits and which group they belong to, the team needed to verify how much information is needed for a link to make sense. They also needed to understand how to present the information about a linked page so that it makes sense for the customers.

The tests also included some specific areas of the site. The project had identified the benefits for parents around the birth to be a good area for tests. It is not obvious if this is a benefit for parents before or after birth and for many customers, this is the first benefit they apply for.

Web statistics had shown that many of the mobile users sought information about how to contact Försäkringskassan. This is not uncommon for all customer groups, but mobile users are more likely to seek contact details than others. The test did not focus on the actual contact page but the team wanted to make sure that this crucial area was easily found for the mobile customer segment.

Conclusions

As these were the first tests we did with an external test agency, we were happy that our assumptions so far had not been far from the mark. The tests confirmed a lot of what the team had seen during the first guerrilla tests. Both our first simple tests and the external tests were crucial. During the first simple tests, the team could see obvious flaws and better fine tune the prototypes. During the external tests, the tests became blinded and more validated. An important conclusion was that both methods will be used during the whole project and are crucial for its success.

The team decided that both concepts should be included in the work for the third sprint. Both concepts had flaws and positive aspects and more tuning was needed before deciding on one concept. Also, the tests had only been conducted on a small test group and they needed to be done with a larger customer group.

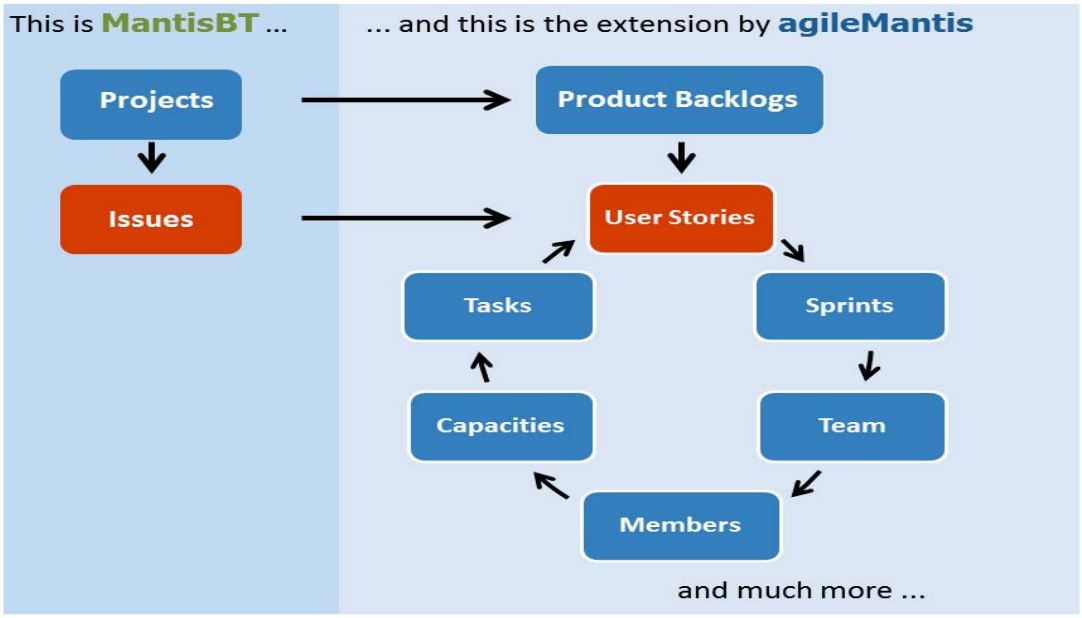
We now had one iteration left before moving into implementation…

This third iteration will be described in an article published in the next issue of Methods & Tools.

agileMantis enables Scrum for your Mantis-Installation

Dr. Jörg Baumgarten, gadiv GmbH, http://www.gadiv.de/, @agileMantis

Scrum is one of the most popular and ongoing models for Agile software development. agileMantis is an open source plugin that extends the open source bug tracking system MantisBT with Scrum. The figure below tells more than a thousand words:



Web Site: http://gadiv.de/de/opensource/agilemantis/agilemantisen.html.
Version Tested: agileMantis 2.1.
License and Pricing: Free version: Open Source, MIT-based license. Extension by expert version: paid license per time and user.
Support: intensively used “tickets”, “discussion” and “blog” items on the website. Additional support by email agilemantis@gadiv.de from 08:00 a.m. to 04:00 p.m. CET at working days in Germany.

Installation: A detailed description is found in the documentation [2]. Free version installation in short:

System requirements	Product	Version
Required	Microsoft Windows Any other system, where MantisBT runs on	7 or higher
Mandatory	A running installation MantisBT	1.2.5 to 1.2.19
Required	MySQL	5.2 or newer
	MS SQL-Server	10.50.1600 or newer
Recommended	PHP	5.3.8 or newer

1. Download and unzip agileMantis from Sourceforge [3]
2. Copy the files in the MantisBT plugin folder
3. In MantisBT with admin rights: Click **Manage** → **[Manage Plugins]**-> **[Install]** in the agileMantis -row.

Documentation: All documentation for installation, a beginner tutorial and a detailed technical description are found in a wiki as well as with downloadable pdf documents on Sourceforge [4].

Demo version: From the Sourceforge page, you can start agileMantis to try its look and feel. Both the free and expert versions can be tested there.

Walk through Scrum with agileMantis

We will now provide a short introduction to Scrum and describe why and how agileMantis supports it.

Why Scrum?

During the past ten years, software development strongly transitioned from “waterfall” driven procedures with huge concepts full of deeply detailed and very long term planned procedures to Agile development. Scrum and others approaches are about to replace these old forms of software development simply due to their better performance and flexibility. Especially Scrum is nowadays perfectly organized by a network of parent organizations, schools and trainers and by its well-defined phases and artefacts.

How agileMantis brings Scrum to MantisBT

agileMantis is designed to bring all Scrum Artefacts and organizational topics to the users of MantisBT. MantisBT is a pure open source bug tracking tool. If a MantisBT user wants to switch to Scrum, the only thing to do is to plug in agileMantis. All existing MantisBT data can then be integrated in the Scrum process. The agileMantis plugin manages all Scrum information and MantisBT data sets are never altered.

Initializing Scrum in MantisBT: Create Product Backlog, Team and a first Sprint

After the installation, you can immediately setup your individual Scrum environment with three basic steps.

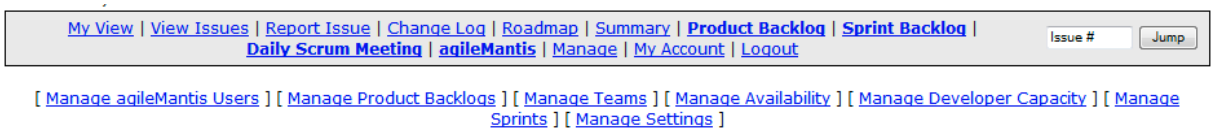


Figure - Extension of the MantisBT menu by agileMantis with activated agileMantis administration menu.

Step 1: Create a **Product Backlog**: You already have projects in MantisBT. Now, you can define a Product Backlog and give it a name. Assign the MantisBT projects to the Product Backlog. Select Issues of these Projects into the Product Backlog. agileMantis now treats and displays these Issues as User Stories.

Step 2: Create a **Team**. You already have users in MantisBT and just have created a Product Backlog. You can now define a Team and give it a name, assign the Product Backlog to the Team. Assign users to the Team and give each member a role (Developer, Scrum Master, Product Owner..) and agileMantis rights.

Step 3: Create the first **Sprint**. You have just defined a Product Backlog and a Team. Now you can define a Sprint and give it a name, assign the Team and the Product Backlog to the Sprint.

With this your Scrum Team can start working. You can operate with as many Product Backlogs, Teams and Sprints that you want.

Performing Sprints

Scrum follows an elementary rhythm. With agileMantis we swing into this rhythm.

Planning part 1: Select the Product Backlog and check for new User Stories. You can use planning poker for new User Stories and enter the estimated Story Points for each Story.

Planning part 2: [Optional] Enter the days and time that each Team Member has available for the following Sprint in the scheduler.

Planning part 3: Create or select the new Sprint. Choose User Stories from the list offered by the Sprint's Product Backlog and assign them to the Sprint. For each Story, you can create Tasks and assign them to a developer and optionally enter the planned work. The scheduler issues warnings when a developer is assigned to more work than the developer can perform. Using the velocity of previous Sprints, such planning robustly prevents that the team is over- or underloaded with work.

Finish planning: Enter the start and end dates of the Sprint, its goals, and then press "Commit". That is the magic moment. The members can now perform completely all Tasks until the Sprint's end.

Performing development: After the planning, the User Stories and their tasks are automatically displayed on the Sprint Backlog for the free version or the Taskboard for the expert version. The initial status of all Tasks is "toDo". To perform a Task, a developer switches its status to "at work". You can now enter the amount of performed work from time to time. When a Task is finished, its status is changed to "done". When all Tasks of a Story are "done", its status is set to "resolved".

Finishing the Sprint: Close the Sprint at its end date. That's it. You can perform the Review using the listed User Stories and start the next Sprint.

agileMantisExpert compared to agileMantisFree

As you can see, using agileMantis requires quite little input to get considerable benefit of automatic aid to the process. However, agileMantis comes in two forms: a free version and an expert version. What are the major differences?

The free version

agileMantisFree is a free open source tool that is written "MantisBT-like". This means that its usage, look and feel are those of MantisBT. It is keyboard-driven, has no graphics and no statistic presentations. It is a "tables-only" tool.

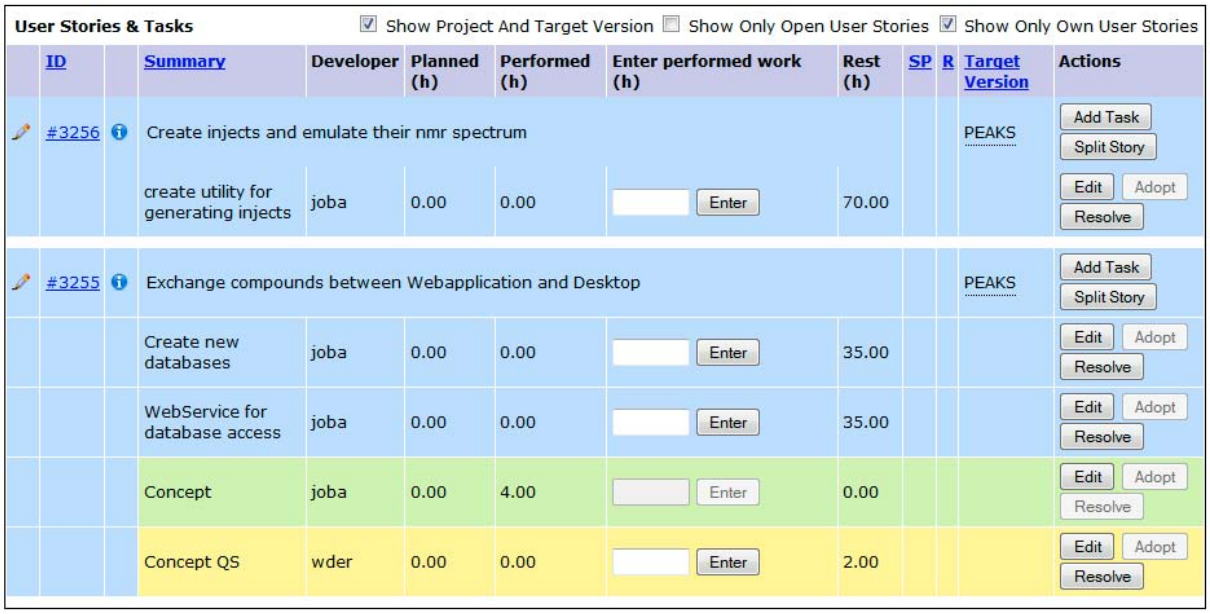


Figure - in the free version, the Taskboard appears in table form as “Sprint Backlog”.

The expert version

agileMantisExpert is a commercial tool with paid licenses. It extends the free version and is also installed as a plugin in MantisBT. As a Java applet, it requires JRE 1.7.0_51 or newer.

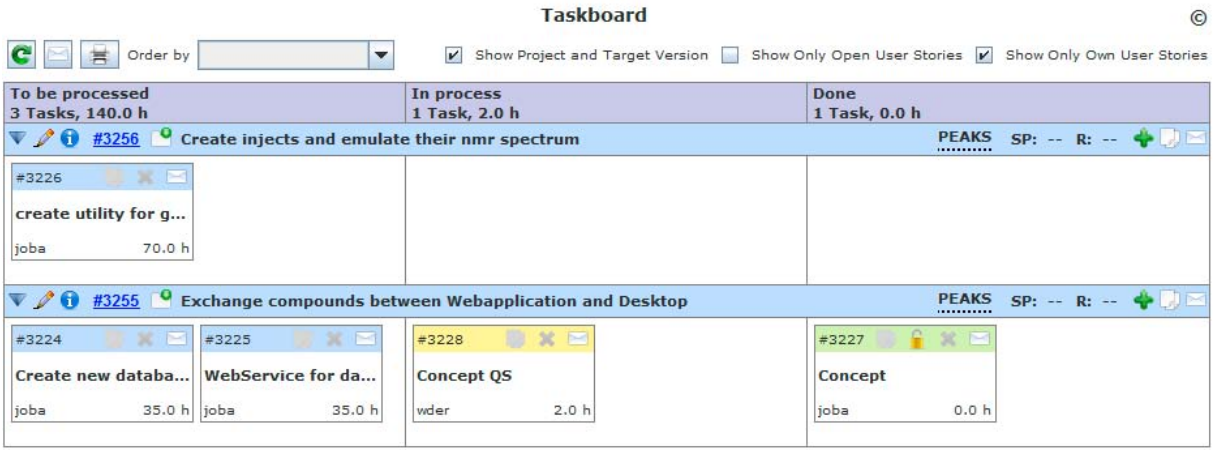


Figure - In the expert version, the Taskboard is a mouse driven rich access platform for information management.

The expert version is written to achieve optimized ergonomics and presents a much richer and more comfortable interface with far better overview. It is mouse-driven and offers a visual Taskboard together with an optimized Daily Scrum Board. The full content of each User Story is accessible through “quickAccess” popups from everywhere in the application. It keeps a versatile set of graphical monitoring presentations of velocity, stress and success of the current and former Sprints, etc. Last but not least it has individual immediate user support by mail.

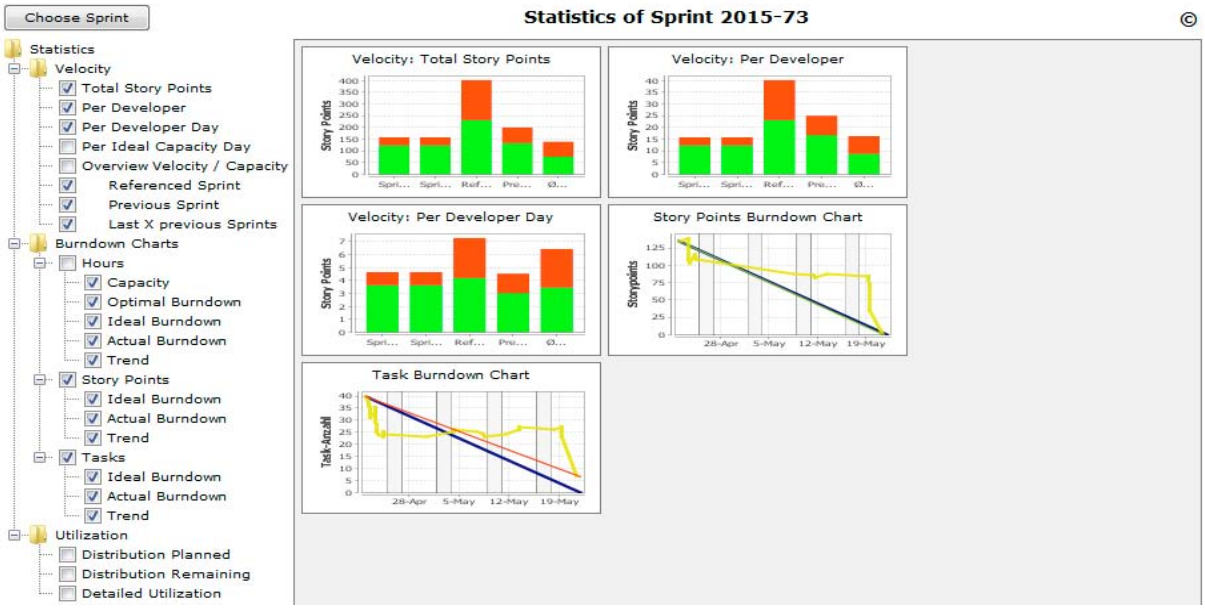


Figure - A rich set of graphical evaluations helps to optimize strategy and performance.

Licenses can be ordered for 3, 6, 12 or 24 months for 3, 5, 10, 25, 50 or 100 users. More details are found on the agileMantis home page under “Ordering the License Key”.

References

- http://gadiv.de/de/opensource/agilemantis/agilemantisen.html.
- http://sourceforge.net/p/agilemantis/wiki/Installation/.
- http://sourceforge.net/projects/agilemantis/
- http://sourceforge.net/p/agilemantis/wiki/Home/.

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