# METHODS & TOOLS

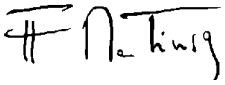
**Practical knowledge for the software developer, tester and project manager** **ISSN 1661-402X**

Fall 2010 (Volume 18 - number 3) www.methodsandtools.com

**Software Crisis? What Software Crisis?**

History says that the term Software Engineering was coined in 1968 at the NATO Software Engineering Conference in Garmisch, Germany. The aim of this conference was to tame the "software crisis". Since then, software development seems to have been continuously in crisis and we still debate today if we are software engineers or software craftsmen. Many reports exist on the fact that software projects have difficulties to respect initial budget and schedule, which is confirmed by real life experience. Multiple approaches have been devised and promoted to change this situation. While we can always discuss the problems that plague our industry, I found interesting to compare it with a "true" engineering area: construction. In Switzerland we can read every week in the newspapers the mention that some public construction project is not respecting its budget, schedule or often both. Do you see a debate about the fact that tunnel or road building is not engineering? Have you read about the " building crisis"? I don't. Building engineering has been presented in the past as a model for software engineering. If we could define precisely the requirements like the construction industry, we would be able to achieve the same results. This was the foundation of the methodologies crafted in the 20th century (SSADM, Information Engineering, Merise, RUP, etc.)

Tunnels and software project could have a lot in common. When you start digging, you have an overall idea of where you want to go, but you don't know exactly what you will meet in the middle. This is why the Agile Manifesto prefers "Customer collaboration over contract negotiation" and "Responding to change over following a plan". Getting back to the concept of software "crisis", I think that there is no crisis. Every project meets obstacles. Some teams manage to overcome them, some not. Our software obstacles are the uncertainty of requirements and the heat that dissension between people can create. No process or tools can help overcoming this, only collaboration and the ability responding to change with incremental development. Even if we cannot plan the changes that every project will face, we should remember that laser-guided technology allows tunnel boring machines that starts from each end to meet after a journey of many kilometers underground. Choosing trusted technology and using good practices are a prerequisite to reach success in every project context. Otherwise, you project will already be in crisis before its beginning.



## Inside

# Distributed Teams and Agile

Craig Knighton, craig @ marcatopartners.com
Marcato Partners, http://marcatopartners.com

## Principles and Practices

My career in software development has had many twists and turns, but looking back on it there are definitely a few key moments in which small decisions turned out to have lasting consequences. One such moment was the day back in 2002 when our CTO told me that during the last board meeting we had been asked to develop a plan for taking some portion of our development offshore. Most of my development experience had been working with other engineers that were in the same building, so it was hard for me to imagine what it would be like to create and sustain a distributed development team.

Since that time I have worked with many successful distributed teams with team members from Duluth, Minnesota to Guangzhou, China. My most recent team had team members in Minneapolis, Houston, Denver, and Minsk, Belarus. In the past, I've also worked with teams in Chile and India - while each experience is different, there are many common elements that contribute to a success of the effort. In hindsight, I can now see that our early decision to continue to use Agile methods to manage the flow of work to and from these teams was instrumental to that success, but at the time we were not nearly so confident that it would work.

When you are looking at your first distributed project, the most natural reaction is to try to figure out how to control the outcome through documentation. It's reasonable to assume that communication will be challenging (especially with other countries in different time zones) and that the risks are high, so you use a contract and detailed requirements and design to capture exactly what you want and your expectations for delivery. The road is littered with unsuccessful projects that begin this way, yet the distributed Agile teams we have worked with have all been successful. This seems counter-intuitive, yet the proof is in the results. The goal of this article is to share our experiences and the model for organizing and operating distributed Agile teams that evolved from these efforts, but the main message is much simpler - Agile is not only possible with distributed teams, but in all but a few situations, it is the BEST way to lead a distributed team.

## Principles of Success

Instead of diving right into the execution details, let's first step back and look at the principles that should drive our structure and operation. Once we take the time to do this, you should find that you have the understanding needed to fill in the gaps with your own decisions rather than blindly following our specific recipe. Here are the principles that we found led to success - let's take some time to discuss each:

- Divide by feature, not function
- People are the same everywhere
- Individuals and interactions over processes and tools
- Choose team leads carefully
- Treat vendors as partners
- Invest in the distributed lifestyle

*Divide by Feature, not Function*

If you are looking at an organization chart with a typical reporting structure, then there is a natural tendency to consider distributing along the boundaries of these functional teams (product management, development, testing, technical writing, etc.) Outsourcing vendors will tell you that this will work fine, but while this view of your organization captures the reporting structure, it actually has very little to do with how work flows within your teams and how commitments form and get delivered.

While tempting, we chose instead to try to divide along the lines of functional feature teams. The most important goal is to group developers and testers together, but we also had great success distributing detailed story/requirements development to the remote team as well.

*People are the Same Everywhere*

Another possible structure for distributing work (especially when using outsourcing) is to divide along the boundaries of maintenance and support for existing legacy products versus strategic investment in new products and technologies. Your local team will compete vigorously to retain whatever work they view as interesting or exciting and will gladly let go the rest. While this solution seems politically expedient, this approach can lead to frustration throughout the organization when the remote teams lack the expertise and proximity to the support organization to provide the expected timely response.

Turnover within a team is a real issue no matter where the people are - the single most significant expense is in building the expertise of a team member in the domain and code base. If you distribute the unseemly work to other locations, then you should assume that you will have retention issues there as well. Instead, assume that every member of your teams no matter where they are located needs the same things - interesting and meaningful work and an understanding of the impact their work will have on the company as a whole.

*Individuals and Interactions over Processes and Tools*

One of the central values in Agile is “Individuals and Interactions over Processes and Tools” - if you look at how this value gets applied in methodologies such as XP or Scrum, what you see is that practices such as co-location, pair programming, and daily scrum meetings all serve to heighten the frequency and quality of team interactions. Given that this is so central to success with Agile, is this a fatal flaw to applying Agile to distributed teams?

In short, no - many of the other principles help to compensate for the separation in space and time. While Agile certainly works best with everyone on the team in the same room, we will show you later in the article what you need to do to extend this same high quality communication rhythm to a distributed team. Rather than being a detriment, staying true to this Agile principle is the key to managing a successful distributed team.

There is a time during team formation when it’s a good idea for team members to meet and get to know each other face to face. We chose to travel to Chile and Belarus at the beginning of those relationships to build rapport within the team, and because it was a long term relationship we also alternated having team members travel between the geographies every 6 months or so. We also frequently use web cameras during discussions and sprint reviews just so we can all see each other. We’ll discuss this in more detail below when we describe the distributed lifestyle.

*Choose Team Leads Carefully*

There is nothing more poisonous to any project, distributed or not, than team leaders that are not invested in making the structure work. Project work is always demanding, so if leaders are looking for an excuse to fail, they will find an easy excuse in blaming the remote teams for not doing their part. Instead, what you want to find are people that enjoy the experience of working through others to get work done and who believe they can make it work. You also need leaders like this in each locale, so having fewer groups with larger teams improves your odds that you can find the leadership talent you need.

*Treat Vendors as Partners*

Assuming that part of your distributed team includes using vendors in other geographies, this organizational divide reinforces basic trust issues that are naturally present in all companies. Informal and open ended agreements are hard to structure inside the boundaries of a company; if you now try to extend that to your vendor relationships, you can just imagine how your legal team will react.

Typical vendor contracts steer you towards building specific statements of work for each deliverable. The more specific the terms, the more likely it is that both sides of the contract will start to require more detail to be captured before the commitment is made, and before you know it you are back to “big upfront design”. Instead, look to structure you statements of work around your iterations with scope and acceptance being built into the natural iteration process.

*Invest in the Distributed Lifestyle*

The career of a software developer has many possible paths - some choose to remain “technical”, while others pursue leadership opportunities. Regardless of the path you have in mind, I am reminded of an important economic truth mentioned in “Practices of an Agile Developer”: “Machines and CPU cycles used to be the expensive part; now they are commodity. Developer time is now the scarce - and expensive - resource.” (1)

This simple fact has been true for at least 5 years now, and it has changed just about everything. Technical expertise now requires a high degree of specialization in order to justify the cost of you as a resource. You will have to work harder and train longer to achieve a level of proficiency that you can sustain over a long period of time.

This has also leads to more mobility - companies may only need your particular specialization for a period of time. If your contribution is “just” project management, product management, coding, or testing, then it is likely that those skills can be acquired somewhere else at a cheaper price. There are, however, several very durable skills that companies will always value: domain expertise, leadership skills, and process development to name a few. Regardless of what you do, if you are able to position yourself as capable of delivering what a company needs in a cost efficient manner, then you’ll have all the job security in the world. You just have to be prepared to listen to what they need and deliver it on their terms, not on your own.

In a recent meeting with a venture capitalist, I was surprised to learn that he considered the use of distributed teams in low cost countries (aka offshoring) a done deal. In his words: ”I think that there are now enough people that know how to make it work that if I were to start up a new venture, I can assume that you would choose to use offshore resources from the very beginning.”

That's a huge change from when this phenomenon first bumped into me about 10 years ago. At that time, it was a board-led dictate that we begin using cheaper offshore resources. It's a simple question of competition - when your competitors are able to spend the same percentage of revenue on development but can engage 2 to 3 times as many resources…I'll let you do the math. Yes, productivities differ, but not in the long run (6-12 months), and while you may be willing to work harder for a while to try to match pace, this also is not sustainable in the long run.

If you, like me, assume that this shift in development demographics is an irreversible reality, then the next step is to imagine how your work will change to accommodate this reality. Here's a description of a typical day in our world.

**A Day in the Distributed Life…**

When I show up at the office at around 7 AM, there were usually one or two people already there. These were the distributed team leads. I walk by their offices and wave but not interrupt, as this is their quality time. We are working with teams in Eastern Europe and have 2-3 hours of overlap every morning. The remote folks had also adjusted their schedules and tend to arrive later in the morning and to work further into the evening (6 to 7 PM local time is typical).

The leads have their headsets on and are most likely talking to their liaison in the remote team - we use Skype or ooVoo and cheap cameras and headsets. Everyone is set up on Skype and Instant Messenger as well and we always have it running when we are working. Larger team meetings also include multiple cameras at different sites around the world and GoToMeeting or Webex for desktop sharing. With all of these technologies available all of the time, impromptu questions or issues can be quickly dispatched as if the other person is in the cube next to yours. Routine is also important - regular scrums at the same time of day or larger group meetings on the same time and day really help communication flow.

If today happens to be Sprint review day, then everyone on all of the teams knows that we meet starting at 8:30 am in a virtual conference. Since the entire infrastructure for participating is there and remote access is pervasive, I might decide to sleep in and attend via phone, Webex, and camera. If I'm travelling, then I can still manage my schedule to be somewhere where I can participate.

What do the local team leads and members do? As already mentioned, a significant part of their day is spent in communication. They also review code, make architectural and design decisions, and often write code themselves. Once their remote team is done for the day, they still have half a day for their own use, and depending on their particular interests, they can use it as they see fit. Their job is to ensure that the team is successful - as leaders in a self-directed team, they are entitled to do this any way that works for them.

This is the trade-off I hoped to communicate - companies have to trade off flexibility on their part for flexibility on yours. If you need to be available to your remote team anytime and anywhere, then they need to give you the resources to do that (and they can afford to from the money they are saving). Once you have that infrastructure in place, you can participate at any time or place. You may realize that you only need to be in the office a few days a week - more power to you! You may head home from the office after lunch because you have a school function to attend but then you'll be back on the phone with your team that evening to wrap up a few project deliverables.

It's been a while since I made this transition myself, but I still remember it well. You may think at first that you have to try to do two jobs at once, but you'll quickly figure out that this is not sustainable. Invest your time instead in getting the infrastructure and flexibility you need, build a sustaining rhythm with the remote team, and then make sure you scale back your total hours to a reasonable amount of time. You should end up working a similar number of hours in a day, maybe just not all in a row or at the same place.

I'd like to wrap up by reiterating one final thought - your value in this new world comes not from what you do with your own two hands but from the value you liberate and the leverage you create by enabling 10 hands to work on your behalf. Once you get to see this in action, you will find that it is just as satisfying to see your ideas become reality through your team's efforts AND the dirty secret is that it's a lot less work! You really can ask for something before you go home at the end of the day and find it waiting for you when you get to work the next day. The people I have seen that latched onto this challenge and made it work for them have learned a skill that is valuable in any software organization in the world. Now THAT is job security.

**Creating and Scaling the Distributed Agile Organization**

As was mentioned earlier, we approached our first experience with distributed development with a fair amount of skepticism. We were concerned about what this meant to how we organize, how we manage delivery schedules, and what we would see for work quality. We chose to partner with an existing business representative in Chile. This team had shown the ability to use our tools to customize our reference application, so we knew that they could do the technical work, but they had no experience with the processes and tools needed to make offshore successful.

Fast forward to today; we've had a great strategic partnership for the last couple of years with an outsourcing firm based in Minsk, Belarus and we are in the process of transitioning from working with them to an even larger team that is part of our new merged company based in Bangalore, India. Now confident in our use of Agile, this methodology has proven excellent at managing the pace and communication needs when interacting with distributed teams. We've also figured out the structure, roles, and skillsets needed to make it work and we now have roughly two-thirds of our head count offshore. This article will focus on explaining the possible structures, those that we tried along the way, and finally the structure that has proven most productive for us.

We have over the years done a lot of experimenting with different organizational patterns. One possibility is to outsource a "function" such as product management, coding, project management, manual testing, architecture, or automated testing. I've seen organizations conclude that certain functions are less strategic to their organization, so they choose that function for outsourcing. Management literature recognizes this approach as a sound strategy, but to choose which functions are less strategic, you need to make certain value judgments about the work that these functions perform.

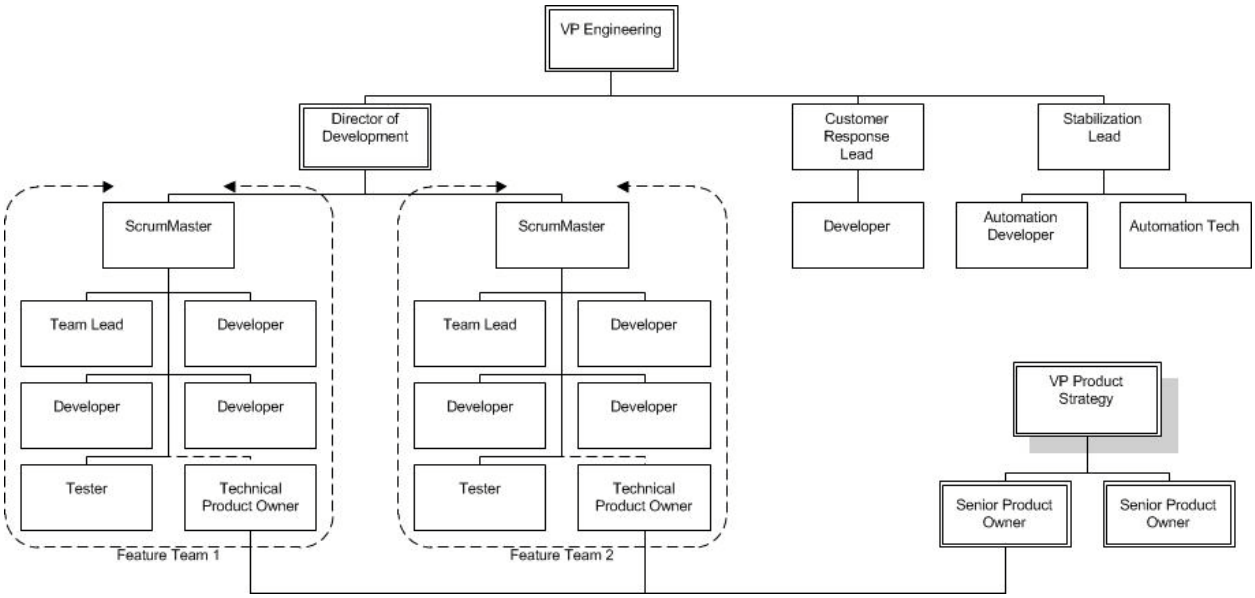
We have avoided this option for a couple of reasons - first, it doesn't scale very well. If you decide to only outsource testing, then you are inherently limited to only saving money on that portion of your R&D spending. If we assume a typical 3:1 to 5:1 developer to tester ratio, then obviously this strategy can leave you with no lower cost option for 75-90% of your resources. My second reason relates to the process and organization - we are firmly opposed to separating the test resources from the developers in the feature time. We use testers as an "in line" participant that works side by side with the developers to design and test the solution. The natural barriers to accomplishing this kind of cooperation are only reinforced if there is a geographic boundary separating the teams. Since the testers need to belong to the team and sit right next to their lead and fellow developers, this has never been an attractive option.

From here there are numerous details to consider - can requirements, development, and testing be done by remote teams? How will final product acceptance be done? How will we divide the resources? Have some teams local and others remote, or have all teams have some functions local and others remote?

These decisions form a crossroads for your engineering organization. I've seen a lot of energy go into rationalizing and justifying one plan over another, and it is certainly true that any decision made here is subject to strong political influence. Rightly so, as quite literally the future of some people's jobs lies in the balance, either because the job may be eliminated or the responsibilities change to the point that they may not choose to stay.

In the simplest and coldest of terms, every organization is looking to minimize the cost associated with product development and maintenance. There are certainly situations in which the product strategy of the company helps to justify the decision to work only with a local and more expensive team, but while most companies start with this in mind as they race to get their initial product to market, few companies choose to pursue this strategy in the long run.

The net result is that we use a hollow local organization comprised only of the following roles:




**Senior Product Owner** - the product manager that is responsible for the product or product lines; this person works with the director to negotiate release schedules and scope and writes the high-level stories (often epics) that drive the release content. This person may also have technical product owners that are embedded in the feature teams to provide local expertise and the additional detail needed by the team.

**Director** - this person is responsible for the setting the overall rhythm and sustaining the process; this person schedules and moderates the Sprint reviews, estimation meetings, and Scrum of Scrums and works with the product owner(s) to determine the release rhythm, set release scope and schedule, and to make sure there is always sufficient product backlog ready for each Sprint.

**Feature Team Lead** aka **ScrumMaster** - this person is responsible for the work product and for all communication with the remote feature team. This is usually a senior developer that has also made the transition to a leadership role; they drive the architecture and design, review code, and manage the flow of stories into the team throughout the Sprint. This person is responsible for coordinating all team functions including kickoffs, daily scrums, and retrospectives although they may offload some or all of this responsibility to the remote team leads.

**Customer Response Team Lead** - this person is responsible for triage and resolution for all immediate reaction and customer driven work that cannot be handled by the normal release rhythm. They may also provide the configuration management function for the team as they sit at the nexus of multiple code branches and are responsible for back- and forward-port activity needed to keep these branches functionally complete.

That's about it. Everything else, including the stabilization lead, is a candidate for outsourcing. In fact, let's dwell for a moment on one role that may be a surprise to you - technical product owner. First, to clarify - this role is responsible for converting epics determined by the senior product owner into digestible stories that are granular enough to fit within the Sprint interval. This person needs a detailed understanding of the existing product and will produce detailed

user interface designs and requirements based on the overall direction of the senior product owner. They need to understand the users of the system, but they do NOT need to understand overall market direction or demand.

When we first formed this team structure, the most common complaint during our Retrospectives was that the developers did not have enough access to the product owner. We could have opted to try to convince marketing to spend more money to hire another local product manager, but instead we opted to use development funds to bring on staff a technical product owner on site with the remote team. The difference was phenomenal - not only did we start receiving much better defined stories, but the remote teams had access to a decision maker that also understood the details of the requirements they had defined. The local product owner was part of the daily scrum and would review progress on definition of detailed stories and to answer any questions that had been escalated over night by the remote team, but over time the remote product owner became confident enough that she could handle these questions without help and just get her decisions confirmed by the local product owner later.

Now let's take a look at this from the point of view of your board or CEO. Assuming that your organization scales somewhere from 15 to 100 FTEs, you are going to see the following distribution of headcount between local and remote resources:

| Total Size (in FTEs) | Local | Remote | Percentage Remote |
|---|---|---|---|
| 10 | 4 | 6 | 60% |
| 30 | 7 | 23 | 76% |
| 60 | 13 | 47 | 78% |
| 90 | 20 | 70 | 78% |

As you can see most of the real coverage comes from scaling the team above 10 people; I have a hard time recommending the use of low-cost distributed teams for product organizations with less than 15 people as you are not getting a lot of leverage from the onshore staff. I've found that offshore efficiencies of at least 2.5 are very reasonable to expect, and in some places you can do even better than that. Using even 2 as a very conservative estimate of the cost multiplier you will experience for remote staff, let's look at the two extremes and summarize the leverage both as a potential cost savings and as increased head count for the same cost. In the case of an original local team of 10 people, you can either reduce your costs by 30% or keep your spending the same and increase your headcount to 16 people. At the other extreme, if you are comparing to a local team of 90 you can either reduce your cost by 40% or increase your headcount to 160 people! Of course, you are most likely going to choose some compromise between cost savings and increased headcount, but either way the business case is compelling.

This analysis assumes the use of an outside contractor organization to provide the resources - part of the efficiency comes from the fact that they are responsible for all aspects of compensation, performance, and benefits. As a result, your much leaner local organization is focused almost exclusively on determining what will be built and architecting, accepting, distributing, and supporting the final work product.

**Location, Location, Location…**

The most important part of making an offshore relationship work are the employees within your company that assume responsibility for the daily communication, work direction, and architectural guidance of the remote team. As we've described before, these people are allocated to the feature team in about a 6:1 ratio (I think there is diminishing returns at about this point,

but depending on the skill of the individual you may be able to push this to 8:1) and they are the conduit for ideas, directions, and decisions flowing in both directions. In short, they are responsible for the quality of the ideas and the work product and are the life blood of a distributed Agile team.

For these people, flexibility is key. To be successful, the will need to find a new lifestyle that is different than the typical 9 to 5 working hours. This is possible, if not mutually rewarding, as long as your let them figure out what is going to work best for them and their team. It's a sure sign of trouble if you see them coming in to talk to the team early in the morning and they are still there at 5 PM every day - you need to encourage them to find a sustainable lifestyle.

Sustainability is essential - when it's new and fun and the project is just getting underway, it's energizing to be doing something different, but this is when the bad habits start. They schedule meetings early in the morning and late at night and think that they also need to be there in the office all day to stay connected with everyone else. In short, this is not possible to sustain - they need to adjust to become something of a remote employee themselves. The closest comparison might be a regional sales force, and just like any company with a remote sales force knows, you need to create regular communication channels and provide the telecommunication infrastructure for them to work anytime, anywhere.

Let's assume you do all of these things right and you now change only the location of the remote team or teams - you would expect the same result, right? I recently experienced watching just that situation play out as we switched from a long term sustainable structure to one with portions of the team on the west coast of the US and other portions in India. Where the leads had been able to maintain a stable communication overlap of 2-3 hours per day in a regular time period, they now had to work hard to juggle schedules on almost a daily basis to find ways to communicate. We would have meetings until midnight and then need to be in the office again by 7 the next day. The team thrashed its communication methods and times constantly to share the inconvenience roughly equally between all involved.

I've heard of other teams that try different alternatives - the offshore provider will say they can solve this problem for you and will likely propose one of two things: they will but one of their employees onsite to be the liaison or they will tell their team in the remote geography that they need to work strange shifts to accommodate their client. Now, I've been fortunate enough to work with many people in many countries, and so far I have to tell you despite cultural, language, and economic differences, they are all just people. There is no reason to think that these people want to live a lifestyle that you would find intolerable. Instead, in the first alternative you only manage to insert another communication layer involving someone with less knowledge of your product and an increased likelihood of churn because they are not your employees. In the second alternative, you will experience a flight of quality as the most talented people in your remote team look for alternatives that will give them a sustainable lifestyle.

My conclusion? Despite years of wanting to believe that I could make any situation in any geography work, I now believe that the best predictor of long term success in a distributed team is location. If you can't create stable communication patterns, then it's only a matter of time until the circumstances deteriorate. If you are only in it for the short run (3-6 months), then it probably doesn't matter, but if you want to be able to retain quality employees at both ends of the distributed chain, then you need to make sure that they have at least 2-3 hours of overlap every business day. While other locales may have raw labor cost advantages, geographies such as South America, Eastern Europe, and someday soon Africa will continue to have structural advantage in providing lower cost, distributed Agile teams.

## Tools and Techniques

If you are new to Agile, then one of the hallmarks of learning Agile is that you are taught to use simple tools and techniques to emphasize the interactions within the team. Daily standup meetings, story cards, task boards, agile estimation - these ceremonies are all designed to bring people together around simple visuals and to simplify the mechanics so that more time is spent as it should be - working together and solving problems as a team.

While it is possible to continue in this way with a distributed team, my experience is that more formality is needed earlier in order to support effective team communication. If I have a team in a significantly different time zone, then usually that team holds their own daily scrum at the beginning of their day and then I will scrum with any other US-based resources and the team lead at a mutually acceptable time later in their day.

You will also find that you want to select and deploy one or more Agile tools designed to support the full application lifecycle needed by your team. While reviewing the pros and cons of each goes beyond the scope of this article, there are several good alternatives from which to choose. At a minimum you will want either a suite or best-of-breed collection that provides the following capabilities:

- Version Control / Configuration Management
- Continuous Build
- Project Management / Tracking including product and sprint burndown charts
- Requirements Management
- Defect Tracking
- Automated Testing
- Unit Testing

I'm currently working with an Agile team where the team members that participate in estimation are in 4 different cities around the world. We started the process by estimating in our heads (or writing it down on something) and then we would randomly pick someone to go first, but we always felt like this approach tended to anchor everyone in whatever the first one or two people said as their estimate, and I was never sure if people were really communicating their first impression.

I was thinking that the ideal tool would be an online application that would allow people to vote and then reveal all of the votes together - and it turns out that such a tool has existed for some time: http://www.planningpoker.com

The site is maintained by Mountain Goat Software and uses their modified Fibonacci sequence. We now run our estimation meetings on that site, but also used GoToMeeting to share a desktop where we would have the story detail. We like this as the discussion almost always clarifies assumptions before the vote and it's nice to capture the important parts of that dialogue directly in your story in the system of record.

There are similar tools available for capturing stories and managing storyboards. Here are a few alternatives to consider if you are interested:

- Digaboard: http://www.digaboard.net
- Pivotal Tracker: http://www.pivotaltracker.com
- Scrumy: http://www.scrumy.com
- More at: http://www.opensourcescrum.com

You will also find online graphical story boards that are either built-in or available as add-ons to each of the mainstream Agile tools.

**Wrapping Up**

Agile and Scrum provide a concrete set of practices and ceremonies that support a set of core values designed to improve the likelihood of delivering commercially successful software. Although each group that adopts Agile tends to modify those practices as they see fit, it is hard to know when you cross that line to where the system is no longer working as intended.

After years of using the model described in this article, our conclusion is that distributed Agile does work and that if you spent a day working with one of these teams you would see the same high-functioning, healthy teamwork as you see in any co-located team. Because of this, we are confident that extending Agile to distributed teams can be done in a way that remains true to the values and practices of Agile. We also believe that once you have used Agile to manage a distributed team you will agree that it is a great way to increase the likelihood of success with that team.

**References**

Subramaniam, Venkat, and Andy Hunt, Practices of an Agile Developer: Working in the Real World, Pragmatic Bookshelf, 2006, p. 36.

# Decomposition of Projects: How to Design Small Incremental Steps

Tom Gilb, Email: Tom @ Gilb.com
http://www.Gilb.com

The basic premise of iterative, incremental and evolutionary project management (Larman 2003) is that a project is divided into early, frequent and short duration delivery steps. One basic premise of these methods is that each step will attempt to deliver some real value to stakeholders. While it is not difficult to envisage the steps of *construction* for a system, there is often difficulty in carrying out decomposition into steps when each step has to *deliver* something of *value* to *stakeholders*, in particular to end-users. This paper gives some guidelines, policies and principles for decomposition. It also gives a short example from practical experience. The evolutionary project management method (Evo) is used (Gilb 2005).

## Introduction

See figure 1, which outlines the difference between the Waterfall method, incremental methods and evolutionary methods. This paper uses the Evolutionary Project Management (Evo) method as described by Gilb (2005).

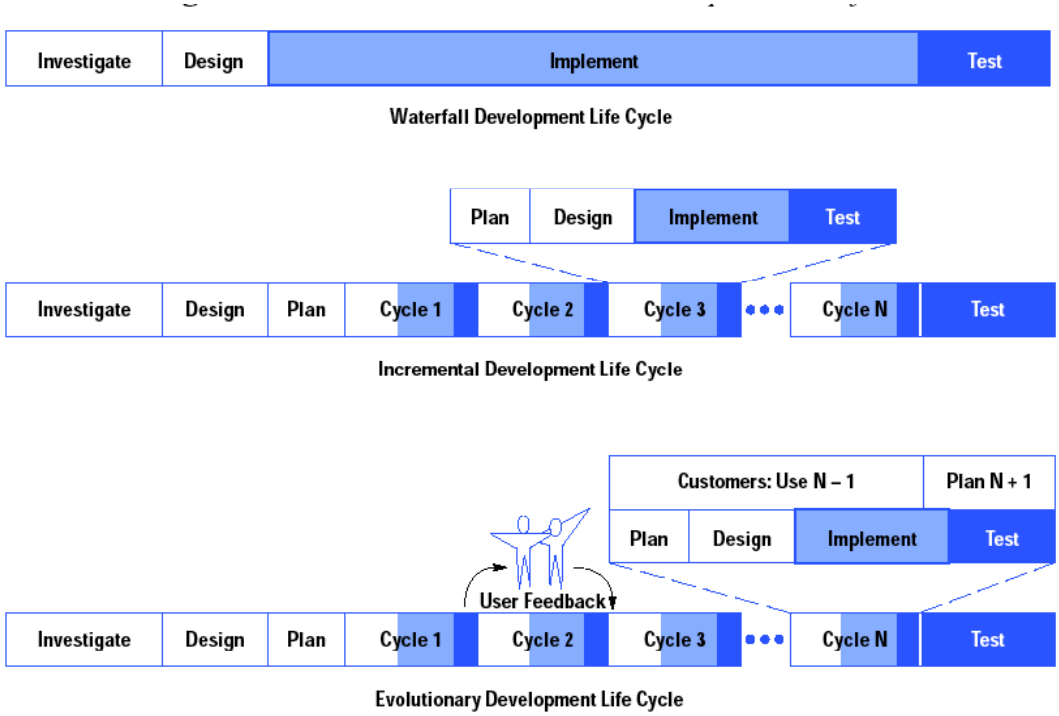



Figure 1. Life Cycle Concepts (Cotton 1996)

There is evidence that evolutionary methods provide benefits. For example, Cotton (1996) states, “Benefits of Evo: The teams within Hewlett-Packard that have adopted Evolutionary Development as a project life cycle have done so with explicit benefits in mind. In addition to better meeting customer needs or hitting market windows, there have been a number of unexpected benefits, such as increased productivity and reduced risk, even the risks associated with changing the development process.”

The experiences of a Norwegian company producing a market research tool, Conformit, also proved the benefit of using Evo. See Table 1.

| Description of requirement/work task | Past | Status |
|---|---|---|
| Usability.Productivity: Time for the system to generate a survey | 7200 sec | 15 sec |
| Usability.Productivity: Time to set up a typical specified Market Research report (MR) | 65 min | 20 min |
| Usability.Productivity: Time to grant a set of end-users access to a report set and distribute report login information | 80 min | 5 min |
| Usability.Intuitiveness: The time in minutes it takes a medium-experienced programmer to define a complete and correct data transfer definition with Confirmit Web Services without any user documentation or any other aid | 15 min | 5 min |
| Performance.Runtime.Concurrency: Maximum number of simultaneous respondents executing a survey with a click rate of 20 seconds and a response time <500 ms given a defined [Survey Complexity] and a defined [Server Configuration, Typical] | 250 users | 6000 users |

Table 1: The end result of about 10 weekly result deliveries to one stakeholder, in 1 of 4 web survey product areas improved simultaneously. This was encapsulated in Product Release 8.5 of Conformit (Johansen 2004).

The basic premise of iterative, incremental and evolutionary project management (Larman 2003, Larman and Basili 2003) is that a project is divided into early, frequent and short duration delivery steps. However, system designers often find it difficult to decompose a system into Evo steps. This paper will give some guidelines (in the form of policies and principles) for decomposition. It will also give an example from practical experience.

**A Policy for Evo Planning**

One way of guiding Evo planners is by means of a 'policy'. A general policy is given in figure 2 (you can modify the policy parameters to your local needs):

**Evo Planning Policy**

P1: Steps will be sequenced on the basis of their overall benefit-to-cost efficiency.
P2: No step may normally exceed 2% of total project financial budget.
P3: No step may normally exceed 2% of initial total project length.

Figure 2. Example of an Evo planning policy

**Profitability Control:** The *first* policy, 'P1: Steps will be sequenced on the basis of their overall benefit-to-cost efficiency' tries to ensure that we maximize benefits by delivering according to the official specified requirements values of the customer (or more generally, the stakeholder), while minimizing associated costs.

If this policy is not in place, technologists consistently choose steps, which are technologically *interesting*, and are not guided by the 'voice of the customer'. An Impact Estimation (IE) table can be used to calculate which potential steps are to be done before others (Gilb 1998). See the IE table example given in Table 2 from Johansen (2004).

Table 2: A 4-day step planned, and 4 days actually used. The step was estimated to improve 'User Productivity' by 20 minutes = 50% of the distance to the target goal of 25 minutes, from its current level of 65 minutes. User Productivity was originally 85 minutes, which minus 20

minutes (the current status improvement) gives the Past Goal of 65 minutes before this planned step. 65 minutes minus (Goal) 25 minutes = 40 minutes (100%) of improvement needed to achieve the target. Actual achievement on this one-week step was 38 minutes, or 95% of the remaining distance to Goal level (Johansen 2004).

| Current Status | Improvements | | Goals | | | Step 9 Design = 'Recoding' | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Estimated impact | | Actual impact | |
| Units | Units | % | Past | Tolerable | Goal | Units | % | Units | % |
| | | | Usability.Replaceability (feature count) | | | | | | |
| 1.00 | 1.0 | 50.0 | 2 | 1 | 0 | | | | |
| | | | Usability.Speed.New Features Impact (%) | | | | | | |
| 5.00 | 5.0 | 100.0 | 0 | 15 | 5 | | | | |
| 10.00 | 10.0 | 200.0 | 0 | 15 | 5 | | | | |
| 0.00 | 0.0 | 0.0 | 0 | 30 | 10 | | | | |
| | | | Usability.Intuitiveness (%) | | | | | | |
| 0.00 | 0.0 | 0.0 | 0 | 60 | 80 | | | | |
| | | | Usability.Productivity (minutes) | | | | | | |
| 20.00 | 45.0 | 112.5 | 65 | 35 | 25 | 20.00 | 50.00 | 38.00 | 95.00 |
| | | | Development resources | | | | | | |
| | 101.0 | 91.8 | 0 | | 110 | 4.00 | 3.64 | 4.00 | 3.64 |

**Financial Control:** The second policy, 'P2: No step may normally exceed 2% of total project financial budget', is designed to limit our exposure to the financial risk of losing more than 2% of the total project budget, if a step should fail totally. If this policy is not in place, technologists will insist on taking much larger financial risks, *with someone else's money*. They will wrongly, but sincerely, claim it *has* to be done, and there are 'no other alternatives'. This is usually due to lack of motivation, lack of strong management guidance, and lack of knowledge of how to divide steps into financially smaller expenditures. (We shall discuss how to handle the problem of decomposition later.)

This is really a 'gambling strategy'. How much can your project lose, and still survive, to complete the project as initially expected?

If technologists cannot responsibly give value for money with small delivery steps, there is every historical reason to suspect they should not be trusted with larger budgets. The 2% number is based on observation of common practice, but it is not 'holy', and intelligent deviation is expected. It is dependent on how sure you are that a step may succeed or fail. The more conservative the step, the larger the financial risk you may be willing to consciously take.

"The Cataract approach [an Evo variation] to build planning may be likened to a rapid prototyping scenario in which the requirements for each build are frozen at the start of the build. This approach, however, is more than just grouping requirements in some logical sequence and charging ahead. Build plans must be optimized on the product, process, and organization dimension, [you] implement the highest priority requirements in the earlier builds. Then, if budget cuts occur during the implementation phase, the lower priority portions are the ones that can [most] readily be eliminated because they were to be implemented last." (Kasser and Denzler 1995)

**Deadline Control:** The third policy, 'P3: No step may normally exceed 2% of initial total project length', is similar in principle to the second, except that the critical resource that is being controlled is calendar time. In this case, decompose the duration of each step by calendar time: as a rough guide, a week at a time is enough. Again, the 2% of time-to-deadline should be applied with discretion – intelligent deviation may be made. But, it is important to have a firm general guideline, or sincere people will always argue – wrongly - that they need more time before any delivery is possible. So, the 2% forces the issue. People have to argue their case well for deviation. And if they constantly deviate, they are to be suspected of incompetence. Give them help!

Cusumano and Selby (1995, page 200) state that within Microsoft, "Projects reserve about one-third of the total development period for unplanned contingencies or 'buffer time' allocated to each of the milestone subprojects." <<<Adding a buffer for some contingency is a good idea, you rarely know exactly what problems are going to be encountered, but notice that Evo will give you some reassurance that you are 'on track' and you should have delivered some early, highest-value parts of the system to your customer, which ought to help with customer relations (Of course, Microsoft have the business model of working towards major product releases).>>>

**Some Examples of Evolutionary Step Size**

The '2%' guideline is based on observation of common evolutionary increment sizes. For example, IBM Federal Systems Division reported (Mills 1980) that the 'LAMPS' project had 45 incremental deliveries in a 4-year period. About 2%. And, cycles were about monthly for 4 years.

This '2%' cycle is above all the 'deployment' component where step content is delivered to the **recipient** and the **host system**. It is the frequency with which the project interfaces with the 'stakeholder. It is the frequency with which we expect to get feedback on the correctness of both our own evaluations, and the users evaluation. If we allow much larger step increments, the essence of evolutionary delivery management is lost. Worst, irretrievable time can go by totally wasted, without us being aware of the loss, until it is too late to recover.

Note that we expect extensive user (and other stakeholder) interaction during development. This interaction refers not only to the feedback of requirements from one delivery to future deliveries, but also to the intimate involvement of the users during the implementation life cycle of each delivery. Evo embraces the premise that the more the real users are involved, the more effectively the system will meet their needs.

Thus, the Evo process includes a role for the users in virtually every step of the delivery life cycle and involves them, at a minimum, in the key decision processes leading to each delivery." (Spuck 1993, Section 2.3)

Microsoft (Cusumano and Selby 1995) stressed, in connection with their 24-hour evolutionary cycles (the 5 PM Daily Build of software) that it was vital to their interests to get feedback daily, so as not to lose a single day of progress without being aware of the problem.

"[A] key event [in Microsoft's organizational evolution] was a 1989 retreat where top managers and developers grappled with how to reduce defects and proposed … the idea of breaking a project into subprojects and milestones, which Publisher 1.0 did successfully in 1988. Another was to do daily builds of products, which several groups had done but without enforcing the goal of zero defects. These critical ideas would become the essence of the synch-and-stabilize process. Excel 3.0 (developed in 1989 and 1990) was the first Microsoft project that was large in size and a major revenue generator to use the new techniques, and it shipped only eleven days late" (Cusumano and Selby 1995).

While on the subject of Evo step duration, note that Microsoft has a variety of cycle lengths, up to two years for a variety of purposes. Note also that evolutionary delivery is a 'way of life' for this very successful company.

"Many companies also put pieces of their products together frequently (usually not daily, but often biweekly or monthly). This is useful to determine what works and what does not, without waiting until the end of the project – which may be several years in duration" (Cusumano and Selby 1995).

Hewlett Packard (Cotton 1996) noted that their average step size was two weeks: "There are two other variations to Tom Gilb's guidelines that we have found useful within Hewlett-Packard. First, the guideline that each cycle represent less than 5% of the overall implementation effort has translated into cycle lengths of one to four weeks, with two weeks being the most common. Second, ordering the content of the cycles is used within Hewlett-Packard as a key risk-management opportunity. Instead of implementing the most useful and easiest features first, many development teams choose to implement in an order that gives the earliest insight into key areas of risk for the project, such as performance, ease of use, or managing dependencies with other teams" (Cotton 1996).

**Principles for Decomposing a System into Evo Steps**

Almost everyone has various problems reducing their initial concepts of a system into Evo steps of a suitable '2%' size. It is, however, almost invariably possible to do so. Everyone can be taught how to do it. But many highly educated, intelligent, experienced 'engineering directors' cannot quite believe this, until they experience it themselves. This is a *cultural* problem, not a technological problem.

Some principles for decomposing a system into Evo steps are given in figure 3.

**Principles for decomposing a system into Evo steps**

1. *Believe* there is a way to do it, you just have not *found* it yet!
2. *Identify* obstacles, but don't use them as excuses: use your imagination to get *rid* of them!
3. Focus on *some usefulness* for the user or customer, however small.
4. Do not focus on the design ideas themselves, they are distracting, especially for small initial cycles. Sometimes you have to ignore them entirely in the short term!
5. Think; one customer, tomorrow, one interesting improvement.
6. Focus on the *results* (which you should have defined in your goals, moving toward target levels).
7. Don't be afraid to use temporary-scaffolding designs. Their cost must be seen in the light of the value of making some progress, and getting practical experience.
8. Don't be worried that your design is inelegant; it is results that count, not style.
9. Don't be afraid that the customer won't like it. *If* you are focusing on results *they want*, then by definition, *they* should like it. If you are not, then *do*!
10. Don't get so worried about 'what might happen afterwards' that you can make no practical progress.
11. You cannot foresee everything. Don't even *think* about it!
12. If you focus on helping your customer in practice, *now*, where they *really* need it, you will be forgiven a lot of 'sins'!
13. You can understand things much better, by getting *some* practical experience (and removing *some* of your fears).
14. Do *early* cycles, on willing local mature parts of your user community.
15. When some cycles, like a purchase-order cycle, take a long time, initiate them early, and do other useful cycles while you wait.
16. If something seems to need to wait for 'the big new system', ask if you cannot usefully do it within the 'awful old system', so as to pilot it realistically, and perhaps alleviate some 'pain' in the old system.
17. If something seems too costly to buy, for limited initial use, see if you can negotiate some kind of 'pay as you really use' contract. Most suppliers would like to do this to get your patronage, and to avoid competitors making the same deal.
18. If *you* can't think of some useful small cycles, then talk directly with the real 'customer' or end-user. They probably have dozens of suggestions.
19. Talk with end-users in *any* case: they have insights you need.
20. Don't be afraid to use the old system and the old 'culture' as a launch-platform for the radical new system. There is a lot of merit in this, and many people overlook it.

Figure 3. Principles for decomposing systems into Evo steps (Gilb 2005)

The list given in figure 3 was compiled by review of the author's decades-long practices and experiences. One or more of these hints should enable you to find a practical solution. Finally, when you have enough successful experience to realize that there always seems to be a way to

decompose into small evolutionary steps, you will give up the mentality of saying *'impossible!'* and concentrate on the much more constructive work of finding a reasonable solution. The mentality that decomposition 'can't be done' is built on a number of misconceptions, so let us deal with some of them.

**Misconceptions about Evo Step Decomposition**

By decomposition, we do not mean that '2% of a car' is going to delivered. We will probably be dealing with 'whole cars'. But we may start with our old car and put better tires on it, to improve braking ability on ice.

The problem of getting 'critical mass', meaning 'enough system to do any useful job with real users', is usually dealt with by one of two methods:

I: Make use of existing systems (while ultimately evolving away from them)

II: Build the critical mass in the 'backroom', invisible to the user.

Then, as the large components (the components that take longer to 'ready' for delivery than your Evo step cycle length) become ready, consider delivering them in future regular delivery cycles. The conceptual problem of 'dividing up the indivisible', which people wrongly perceive as a problem, is explained by pointing out that:

- We do *not* ask you to act like Solomon and demand to 'split the baby in half'
- We are *not* talking about system 'construction'
- We *are* talking about delivering 'results', not technical components, in small increments.

So, the systems are *organically whole*. No unreasonable chopping them into non-viable components is being asked. But we are still not going to overwhelm the user with all the capability of the system. For example, you probably learned PC software, like your word processor or email capability gradually. It might have been delivered to you as five million software instructions from Microsoft, with 20,000 features, most of which *you* do not need. But you could ignore most of them, and focus on getting your current job done.

You probably picked up additional tricks gradually. You found them by experimentation with the menus. A friend showed you some tricks. You might be one of those weirdoes who actually read a manual. Maybe you looked new capability up, using the Help menu. Maybe you got help from a help desk. You probably did not go on a two-week course, with an examination, to learn to be proficient at a high level. Nevertheless, the entire system was 'delivered' to you all at once. Then a new version became available, a million lines of code at once, but again you probably used it the same way as last time, picking up new features gradually.

Microsoft is not worried about building these features as each new user needs them. They build what they must for their market in their 'backroom', and we deliver to ourselves, in our 'frontroom' at an evolutionary learning pace, based on need and ability. The key idea is the incremental *use* of a system, which increases the *user (or at least internal stakeholder such as sales or help desk)* capability. Building and construction are semi-independent topics. You *can* build a system in the same step cycle time as it takes to deploy with a user, but you do not *have* to!

## Some Additional Ideas for Decomposition

Here are some additional ideas for decomposition, which build on what we discussed earlier:

- Divide by deadline: do things early as needed by external stakeholders.
    - Divide by stakeholder: there are typically about 30 internal and external stakeholders (for example, developers, suppliers, senior managers and end-users) in any project.
    - Deliver to one of these at a time (for example, to the salesperson who must demonstrate early to customers).
    - Divide by experience of stakeholder: deliver to the very experienced stakeholders first, and the novices later (or vice versa).
    - Divide by individual stakeholder: deliver to one particular local stakeholder that you have a good relationship with.
    - Deliver to an internal example of a stakeholder (that is, someone with the same function within your organisation), so that if something goes wrong you can control the bad news.
    - Deliver to a highly co-operative stakeholder that will give you adequate time (Johansen 2004)
- Divide by geography.
    - Deliver in your site's town or country before conquering the world.
    - Deliver where critical leading edge clients are to be found.
- Divide by value: deliver the highest value first, but remember that different stakeholders have quite different values, and value is not always financial.
    - Divide by tasks in a set of tasks: divide into sets of tasks, so that some useful tasks are operable early and 'luxury' tasks, which are infrequently used, come last.
    - Divide by risk level: sometimes the highest 'value' to internal stakeholders, is to get the high risk strategies and architectures tested to make sure that they really will work, before investing more on scaling up.

## Summary

Most systems are capable of decomposition into Evo steps: an early (next week), continuous series of about one-week-to-delivery-to-stakeholder result cycles. The decomposition is by value delivery increments; it is not about 'construction' increments.

Evo decomposition can follow a policy that might prioritize profitability, or risk control, or effect on stakeholders, or any combination of interesting objectives.

Evo decomposition can be done as a process, using a set of known principles of decomposition. The most fundamental of these principles is to partially deliver improvements in performance, especially quality levels, to stakeholders.

## References

Cotton, T., "Evolutionary Fusion: A Customer-Oriented Incremental Life Cycle for Fusion", Hewlett-Packard Journal, August 1996, Vol. 47, No. 4, pages 25-38.
http://www.hpl.hp.com/hpjournal/96aug/aug96a3.htm

Cusumano, M.A. and Selby, R.W., *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*, The Free Press (Division of Simon and Schuster), 1995, ISBN 0-02-874048-3, 512 pp.

Gilb, T., *Competitive Engineering, A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage*, Elsevier Butterworth-Heinemann, 2005. ISBN 0750665076, Sample chapters can be found at http://www.Gilb.com as follows:

Chapter 5: Scales of Measure:
http://www.gilb.com/community/tiki-download_file.php?fileId=26

Chapter 10: Evolutionary Project Management:
http://www.gilb.com/community/tiki-download_file.php?fileId=77

Gilb, T., "Adding Stakeholder Metrics to Agile Projects", *Cutter It Journal: The Journal of Information Technology Management*, July 2004, Vol. 17, No.7, pp31-35.

Gilb, T., "Evolutionary Systems Engineering Principles", 2004, Unpublished. Available on request from the author.

Gilb, T., "Impact Estimation Tables: Understanding Technology Quantitatively", *Crosstalk*, December 1998. http://www.stsc.hill.af.mil/CrossTalk/frames.asp?uri=1998/12/gilb.asp

Gilb, T., *Software Metrics*, Studentlitteratur (Sweden), 1976. Also Winthrop (USA), 1977.

Johansen, T., "From Waterfall to Evolutionary Development (Evo) or, How we rapidly created faster, more user-friendly, and more productive software products for a competitive multi-national market", *Proceedings of European Conference on Software Process Improvement (EuroSPI 2004)*, 2004. From which evolved, Gilb, T. and Johansen, T., "From Waterfall to Evolutionary Development (Evo): How we rapidly created faster, more user-friendly, and more productive software products for a competitive multi-national market", *Proceedings of the INCOSE Conference*, 2005.
Available from http://www.gilb.com/community/tiki-download_file.php?fileId=32

Kasser, J. and Denzler, D.W.R., "Designing Budget-Tolerant Systems", *NOCOSE Proceedings*, 1995.

Craig Larman, C., *Agile and Iterative Development, A Managers Guide*, Addison Wesley, 2003. ISBN 0131111558 (pbk). 342 pages. Chapter 10 is on Gilb's Evo method.

Craig Larman, C. and Basili, V.R., "Iterative and Incremental Development: A Brief History", *IEEE Computer*, June 2003, pages 2-11.

May, E.L. and Zimmer, B.A., "The Evolutionary Development Model for Software", *Hewlett-Packard Journal*, August 1996, Vol. 47, No. 4, pages 39-45.
http://www.hpl.hp.com/hpjournal/96aug/aug96a4.htm (not directly referenced in paper but relevant)

Mills, H.D., "The Management of Software Engineering. Part 1: Principles of Software Engineering", *IBM Systems Journal*, 1980, Volume 19, Number 4. Reprinted 1999 in *IBM Systems Journal*, Volume 38, Numbers 2 & 3. A copy is downloadable from http://www.research.ibm.com/journal/.

Spuck, W., "Rapid Delivery Method", *Internal Report (D-9679) of Jet Propulsion Labs, CIT, Pasadena CA*, 1993.

Thanks to Lindsey Brodie, Middlesex University for editing advice and help!

# Delivering Working Code through Automation and Collaboration

Todd Landry,
Klocwork, www.klocwork.com

To keep pace with ever-increasing customer demands on software functionality and time-to-market expectations, software developers have had to evolve the way they develop code. The result is the emergence and rapid adoption of the Agile software development methodology.

By embracing the 'early and often' mantra of Agile, individual software teams and even entire software development organizations can deliver more technology, to more satisfied customers, earlier and more frequently than ever before. However, the reality for many teams trying to successfully execute on the Agile principals is a backlog of software defects interfering with their ability to deliver working software in short iterations.

This article will demonstrate that implementing a repeatable process for ensuring code is as bug-free as possible is key to fully realizing several of the core principals of the Agile methodology. The approach discussed is the use of automated source code analysis (SCA) technology to help developers find and fix areas of weakness in their software source code as they're coding. After providing a brief overview of Agile and discussing some of the barriers to achieving agility, this paper discusses how key capabilities of SCA technology can enhance the Agile development processes and empower Agile teams.

## Defining Agile Development

Simply put, Agile software development is an approach that provides flexibility to accommodate continuous change throughout the software development cycle. It stresses rapid delivery of working software, empowerment of developers, and emphasizes collaboration and communication between developers and the rest of the team, including business people.

Agile contrasts with the still-popular Waterfall development approach, which is front-end loaded with comprehensive scope and requirements definitions, and which employs clear, consecutive hand-offs from requirements definition to design to coding and then to quality assurance. In contrast, Agile incorporates a continuous stream of requirements gathering that flows throughout the development process. Business people are involved throughout the release cycle, ensuring that the software being developed meets the true needs of both the end-user and the business. Change to the requirements and to the overall feature set is expected to occur as outside opportunities or threats arise.

In short, Agile fully embraces change and Agile teams are structured in such a way that they can receive and act on constant feedback provided by the build process, by other developers, from QA, and from business stakeholders.

Agile is based upon a number of guiding principles that all Agile teams follow. For the purposes of this discussion, four principles - or values - are of particular interest:

- Quality software development
- Iterative flexibility
- Continuous improvement
- Collaboration and communication

**Quality Software Development**

The primary focus of the Agile methodology is to enable the development of quality software that satisfies a customer need - i.e. provides a functioning feature or capability - within a specific period of time (typically no more than a few weeks) called an “iteration” or “sprint” in an Scrum context. In theory, a product developed in an Agile environment could be market-ready after each iteration.

Delivering a series of market-ready products, each in just weeks, demands that a rigorous quality process be built into the Agile development cycle. Deliverables in each iteration must be fully developed - meaning tested, defect-free, and complete with documentation.

**Iterative Flexibility**

With a focus on speed and nimbleness, Agile is open to changes that inevitably arise throughout the development cycle. The iterative process is flexible, based on an understanding that original requirements may (or will likely) need to change due to customer demand, market conditions, or other reasons. Because business users are involved throughout the process, and because each iteration is short, new requirements can be introduced and prioritized very quickly.

**Continuous Improvement**

An Agile environment provides developers with an opportunity to learn new skills and to exercise greater autonomy to do their jobs. The iterative framework is empowering because it enables continuous improvement, with testing/quality assurance occurring as part of the process, rather than only periodically or at the end of a long process when it is often difficult or not cost effective to fix coding defects or to incorporate lessons learned along the way. Agile also makes the testing and QA process transparent to the developers who originate the code, further contributing to their learning and facilitating future improvements and coding efficiencies.

**Collaboration and Communication**

Communication and collaboration is critical in software development in general, but in an Agile development environment, its paramount. In fact, the Agile Manifesto (widely recognized as the de facto definition of Agile) emphasizes individuals and interactions as a key concept.

Ultimately, it is open communication and collaboration that facilities efficiencies in the development process. Having access to the right individuals, data and feedback when needed allows the team to deliver working software in short iterations, as the Agile process demands.

**Greasing Agile's Wheel**

As development teams work to embrace the core principals set out in the Agile Manifesto, they are inevitably encountering roadblocks including bug debt, varying developer skill sets, and code complexity. Development teams need to acknowledge these barriers and employ effective strategies that will help 'grease the wheels' to achieving a truly Agile development process.

**Managing Bug Debt**

One of the development principles put forth in the Agile Manifesto states that "working software is the primary measure of progress." [1] Working software implies software that is free of issues that break builds, cause unexpected behavior, or which do not meet the product's requirements, as well as mundane programming defects (a.k.a. bugs).

This principle is not unique to Agile. Many software development processes, including formal ones such as CMMI and Six Sigma, encourage the creation of bug-free code as a fundamental principle. These processes encourage in-phase bug containment - the practice of preventing bugs from being passed downstream from the phase in which they are created. Agile also implicitly emphasizes in-phase bug containment. Given its focus on short iterations, Agile processes must ensure that any potential software degradations are quickly identified and corrected so that the whole team can move on to the next iteration - all while creating functionally complete, working software.

Even within an iteration, Agile teams apply this philosophy through continuous integration and regressions. While this practice effectively addresses defects that can break builds or regression test suites, it is not as effective in cleaning up many of the most common types of programming bugs, which generally fall into these broad categories:

- Memory and resource management
- Program data management
- Buffer overflows
- Unvalidated user input
- Vulnerable coding practices
- Concurrency violations
- A variety of longer term maintenance issues

Bug-filled code creates downstream risk both within an iteration and in subsequent iterations, in the form of bug-debt. If code flaws are not addressed within the iteration, they pass from one milestone to the next, and the bug debt accumulates downstream. Bug debt can kill Agile projects by reducing development velocity which leads to poor implementation results and fewer impactful changes being delivered per iteration. Additionally, bugs that are 'saved up' (or which go completely undetected until further downstream) are more expensive and often more difficult to fix, and can lead to products that fail in the field, disappoint customers, and damage the brand.

It is for these reasons that in-phase bug containment is vital to the Agile process. Developers must take control of the bug identification and removal process while enhancing collaboration amongst all developers to resolve bugs as early in the process as possible.

**Accommodating Varying Skill Sets**

Ideally, your development team is made up of the best and brightest software engineers who enjoy talking to each other and getting feedback from the customer on a frequent basis. Unfortunately, this doesn't always match reality. Typical development teams are made up of a heterogeneous talent pool, combining a mix of less experienced developers with the rock stars; natural collaborators with those that prefer anonymity, etc. Balancing out the team by providing opportunities for less experienced developers to learn from more experienced team-mates and providing ways to make communication and collaboration easier will help maximize the output of working code.

**Restraining Code Complexity**

Code bases are becoming increasing complex, due in part to the following factors:

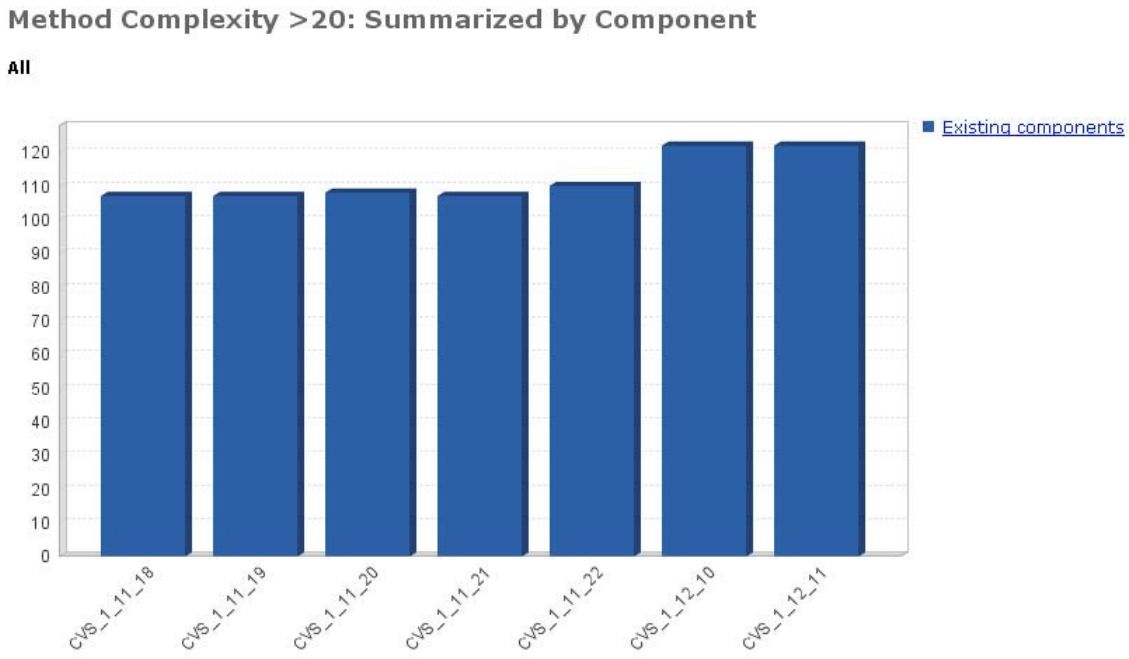
- Code Base Size
  Expanding capabilities and the demand for more sophisticated features and functionality is increasing the size of code bases which leads to more variables and control paths as well as varying language constructs.

- Code Reuse
  Accepted as an industry best-practice, code re-use speeds time to market by increasing development efficiencies while minimizing the costs associated with net-new development. It also enables development organizations to leverage the lessons learned from an existing code base. However, merging that code with new systems can lead to unexpected results.

- Multiple Coders
  Code bases don't live and die within a single version or a single iteration, and in an Agile process it is quite likely that a developer will be editing code that they did not originally create. Inheriting modules means developers are unfamiliar with the original intent of the code, its variables and the language constructs used.

These factors are each contributing to the overall complexity of code bases, which in turn complicates the software development process. This complexity can lead to slipped release schedules, increased defect rates and decreased productivity as more time is taken to accomplish simple, everyday tasks. Development teams should be vigilant in watching for these signs which indicated code complexity is becoming an issue.

With software, you can monitor the complexity of software builds using industry known metrics. For example, you can start monitoring your McCabe's Cyclomatic complexity every build. This measurement shows you how "off" you are from the standard. (McCabe's complexity dictates anything over a value of 20 as very complex.) More importantly, teams should be monitoring the trend of that metric. Seeing a large spike with one version versus the previous might is evidence that complexity issues are impacting your process.




Tracking these metrics in order to identify opportunities to reduce complexity and make code more maintainable (regardless of who is working on it) should be considered. Not addressing code complexity will have a negative impact on the velocity of a software project - which can spell disaster in an Agile development environment.

**Adapting Traditional Process Approaches**

Traditional development processes become recognized as best practices because they bring value to the process - by speeding development, helping to produce cleaner code, etc. Unfortunately, not all of these practices are Agile-friendly. Take peer code review, for example. The value of code review is unarguable. However, in an Agile context, traditional code reviews - scheduling in-person meetings with senior development staff - are highly ineffective and time consuming. These types of traditional processes need to be adapted so that development teams can continue to benefit from them without sacrificing development velocity.

**Adopting Tools and Processes to Achieve Agility**

While the Agile Manifesto's principal of "individuals and interactions over processes and tools" seems to de-emphasis the need for tools, Agile teams use many tools to support their

development - including tools for software configuration management, build management, requirements tracking, testing, project management, and more.

Unfortunately, most of the tools being used are for managing some aspect of the production process and are not targeted at helping developers overcome the obstacles they face during the coding phase. Deploying a properly selected set of developer-focused tools can help ensure high-quality code is output early in the development process. The latest evolution of source code analysis (SCA) technology - combining sophisticated bug detection with collaborative peer code review and automated code refactoring - can help development teams avoid bug debt and code maintainability issues, greasing the wheels to achieving an effective Agile process.

**Automating Bug Detection**

SCA is a bug-detection solution that requires no test cases, is fully automated, and fits well with milestones typically found in an Agile process. SCA technology has grown in popularity and is becoming a mainstream option for professional software developers to reduce the number of bugs in their code while also reducing costs and keeping software development on track.

The underlying technology associated with SCA is called static analysis and the current generation of technology solutions is capable of providing sophisticated, high-value analysis that will locate and describe areas of weakness in software source code - such as memory and resource management, program data management, buffer overflows, unvalidated user input, vulnerable coding practices, concurrency violations, and a variety of longer term maintenance issues.

SCA is distinct from traditional dynamic analysis techniques, such as unit or penetration tests, because the work is performed at build-time using only the source code of the program or module in question. The results reported are therefore generated from a complete view of every possible execution path, rather than some aspect of a limited, observed runtime behavior.

Since SCA is essentially a build-time analysis, it is most effectively used as a build milestone activity when individual developers or development teams run their builds - either at the integration-build level or the developer-build level.

**Integration Build (a.k.a. system build, project build)**

Today's SCA tools go well beyond the syntactical and semantic analyses commonly associated with source code analysis. Good SCA technology today can be expected to include sophisticated inter-procedural control and data-flow analysis with advanced approaches for pruning false paths, estimating the values that variables will assume, and simulating potential runtime behavior. The complexity of this type of analysis across a large system with millions of lines of code and an essentially unlimited number of potentially feasible code paths to consider is not trivial.

To make it all work, and to reduce the number of "false positives" (bugs that the tool incorrectly reports) and false negatives (bugs that the tool misses), vendors naturally provide integration with a project's system build - whether it is make, ant, Visual Studio, or other continuous integration tools such as Electric Cloud and BuildForge - to generate a complete view of the entire source code base. Of course, the downside to running SCA exclusively at the integration build level is that bugs created at the desktop are exposed to the main code stream and can impact other members of the team - both fellow developers and the QA team. When bugs are found downstream - even in a continuous integration context where integration builds are run much more frequently - an additional bug triaging process needs to be put in place to notify the developer of the error (by email, web reports, etc.). This adds more workflow and process, which is contrary to the spirit of Agile development.

Clearly, the solution is to push SCA to the developer desktop so that it can run in conjunction with a developer's build, even prior to him/her running unit tests.

**Developer Build (a.k.a. personal build, sandbox build)**

SCA at the developer desktop offers a big payoff for any organization adopting this technology, in particular an Agile team. If most bugs can be found prior to code check-in, organizations will achieve in-phase bug containment, reducing the number of bugs in the main code stream or integration build. This allows QA to be more efficient by focusing on being a customer advocate and ultimately producing higher quality software - earlier.

For SCA to operate at the developer desktop, it must be delivered within the developer's natural work environment (i.e. a favorite IDE, text editor or command line) and the analysis must be every bit as accurate and intelligent as the centralized analysis that benefits from a view of the entire code stream. Code check-in (or commit) is an important milestone in Agile and many organizations operating in a continuous integration context have a series of gates (smoke tests, unit tests, etc.) that the developer must pass in order for him/her to check-in code. SCA should be added to this series of pre-check-in quality gates.

## Facilitating Collaborative Peer Code Review

Peer code review is a necessary evil in the software development process. Often mandated by management before code can go live, the code review process is time-consuming and usually not a pleasant experience for the person whose code is being reviewed. However, the benefits are plentiful:

- Code reviews create consistency and a culture of quality for a development team. Identifying bugs and design flaws while ensuring coding standards and best practices are met means a higher quality product goes out the door.
- Developers learn from code reviews by having more experienced team members review their code and provide feedback.

In an Agile context, these benefits are important. However, the traditional method of conducting code reviews - scheduling in-person meetings - is not effective in an Agile context. Agile teams wanting to make code review part of their process should consider SCA tools with built-in code review capabilities. These tools facilitate simple, web-based peer review that allows code to be reviewed asynchronously. This approach avoids the need to get a specific group of people in a conference room at a set time to review code. Instead, code is available to a variety of team members - regardless of geographic location, calendar availability, and title - to review and provide feedback on at their convenience. When peer code review capabilities are combined with source code analysis, coding defects are identified so that reviewers can see them, comment on them, and assign actions related to any that require action.

## Simplifying Complex Refactoring

Refactoring - the process of simplifying and clarifying code without changing the program's behavior - can be a difficult and time-consuming activity, especially for those developing in often-used languages like C and C++. With a lack of tools capable of automating the refactoring process, many developers who want to refactor simply choose not to. However, implementing a process that facilitates refactoring code early and often offers some important benefits to an Agile team.

Martin Fowler, one of the original authors of the Agile Manifesto, describes refactoring as a series of small, almost negligible changes to code, that when combined, result in a significant end result. Some of the areas that refactoring impacts includes:

- Adapting to change

  Ongoing change is a fundamental aspect of Agile. By incorporating frequent refactoring into the development process, code becomes easier to work with which allows teams to adapt to change more effectively.

- Defect detection

  By simplifying code and improving its overall design, code becomes clearer. Working with clean code makes spotting quality defects and security vulnerabilities easier. And the sooner defects are found and fixed, the less impact (i.e. distracting developers, bogging down test teams, slowing down the overall process) they have on the process.

- Developer productivity

  Frequently different people will end up working on code that they did not originally create. With refactoring, code becomes easier to inherit, which means developers won't waste iterations reviewing, interpreting and 'fixing' existing code and instead focus more time on writing new code.

Clearly, refactoring has an important role to play in helping teams achieve agility. With today's source code analysis tools which offer built-in refactoring capabilities, Agile teams can automate the refactoring process to reduce the complexity around the process while reaping the rewards it has to offer.

**Achieving True Agility**

By bringing source code analysis technology with its bug-detection, code review, and refactoring capabilities into the Agile development cycle, development teams can achieve in-phase bug containment while fully realizing the core Agile principals discussed at the beginning of this article.

**Bug Free for Quality**

Software bugs are more than a nuisance. Serious bugs can cause downstream inefficiencies, product recalls, or field disasters. Source code analysis can automatically find serious bugs in your code, such as NULL pointer dereferences and memory management issues that can lead to a system crash; or buffer overflows and unvalidated user inputs that make a system susceptible to exploit by hackers. Removing these issues prior to shipping a product is critical and the earlier they are identified the better. This prevents critical issues from being passed to QA within an iteration, or passed from iteration to iteration, both of which greatly increase the risk of shipping buggy software. In addition, if bugs are dramatically reduced before code check-in, this will ensure they never impact the main code stream and will facilitate a more efficient testing and QA process. With few bugs to find or report on, testers are able to focus instead on running functional and performance tests to ensure the product is customer- and market-ready.

**Bug-Free Flexibility**

To adequately test and assure the quality of software developed in Agile environments, the testing process must also be iterative and accommodate frequent change in requirements. Thus, there is little room to address programming bugs in the testing phase. If QA discovers these bugs, unacceptable drag in the development cycle is created -testers report the bug list back to the developers who must set aside their current work to shift back to the frame of mind they were in when working on the original code. By enabling developers to check-in bug-free code, this dramatically reduces or eliminates the time-consuming "rinse and repeat" cycle of check-in, find bugs, debug, and then rework. By using SCA tools in an Agile environment, developers can spend less time fixing reported bugs and more time writing new and innovative software. Within this context, developers have the flexibility to control the quality and security of the code they create during new development, before the integration build is performed.

**Bug Free for Continuous Improvement**

It doesn't matter how an iteration appears to be progressing from a feature development standpoint - if the development team's commitment to quality code cannot be measured, the project is by default accumulating significant downstream risk with each and every build and iteration. In a fast-paced and fluid development environment, Agile teams need to have strong, automated measurement capabilities in place, such as:

- Measure and track bug fix rate at the developer build;
- Identify what bugs are leaking to the integration build;
- Establish nightly build quality milestones (or any other frequency over and above the continuous build schedule); and,

- Track which teams or components are improving over time.

The answers to these questions allow Agile teams to implement targeted continuous improvement programs that quickly identify where help or training is needed so that all members of the development team can submit clean code into the integration build. Measuring and tracking quality from the bottom up is necessary to know if iteration plans are any good and if a shippable product will be available at the end of an iteration.

**Bug Free for Collaboration and Communication**

The short-iteration process found in Agile development requires constant communication and team collaboration in order to be successful. This helps ensure rapid resolution of issues encountered, while avoiding duplication of efforts which creates unnecessary delays. Source code analysis tools provide collaborative mitigation for all reported issues. Developers, architects and other team members can change the status of reported issues or add comments, which are automatically synchronized with other developers. Utilizing this capability, various team members can collaborate on complex issues, and developers don't duplicate effort on the same bug. This ongoing visibility and sharing of data and feedback helps ensure mitigation is achieved early in the process.

**The ROI of Source Code Analysis**

Agile development teams need to be all about maximizing productivity. They need to increase - or at minimum maintain - team velocity and get as many stories completed in an iteration as possible. So how much can SCA tools help with this?

Typical ROI calculations focus on dollars saved, which is important at the senior manager and C-level. But for an Agile development team perspective, let's look at hours that can be saved. First, we must establish a few key parameters around the Agile team. For the purposes of this paper, let's assume:

- 10 developers that have standardized on 2 week development iterations
- Each iteration delivers approximately 5 completed stories
- Each story consists of 300 lines of code, so therefore a total of 1,500 lines of code per iteration are produced.

As mentioned earlier, bug debt can simply kill projects, so being able to find bugs as early as possible in the software development lifecycle is critical. On average, SCA tools find about 3 bugs for every 1,000 lines of code, and provide a time saving of 4 hours per bug found. Applying this to the typical iteration would lead to 4.5 bugs being detected, with a total time saving of 18 hours.

Code reviews are more and more becoming a mandatory part of the development process, and while they can be extremely effective, they do tend to take up considerable time and effort. Code reviews typically involve 4 or 5 developers sitting together in a room for an hour (on average) going through the code in question line-by-line. Doing this for every story in an iteration adds up to the tune of 20 hours. That is a significant amount of time that could be spent elsewhere. Case studies [2] show that when a developer can review code when and where they want to, they are 50% more productive than sitting around a table in a meeting room doing a review of the same code. This translates into a 10 hour saving per iteration.

Finally, starting work on a module you had never seen before can be a painstaking activity and can certainly hamper new development on that module. By performing some simple refactoring while writing the module, inheriting that code becomes slightly easier. Conservatively, let's assume one quarter of an hour saved for every new story that work is started on because there is no need to 'interpret' what the previous developer produced. Working through the math, refactoring can result in a 12.5 hour time saving per iteration.

When adding up the time saving for these three activities, a total of just over 40 hours for every iteration can be realized and applied directly back to real development work.

| | Time Savings/Iteration |
|---|---|
| Bug debt reduction | 4.5 bugs x 4 hours = 18 hours |
| Peer code review | ½ hour x 5 reviews x 4 developers = 10 hours |
| Refactoring | 0.25 hours x 10 developers x 5 stories = 12.5 hours |
| TOTAL | 40.5 hours/iteration |

**Conclusion**

The ubiquitous nature of software today, coupled with the pressure to rapidly develop market-ready features and products in just weeks, has led to two related phenomena:

- The widespread adoption of Agile software development principles; and,
- The adoption of various tools by Agile teams designed to help streamline and de-risk development projects.

Some of the most important types of tools that an Agile team can deploy are ones that aid in writing better-quality code. Source code analysis tools provide an automated method to detect a significant number of software bugs or security vulnerabilities right at the developer's desktop - before any code is delivered to the integration build or testing team. Combine that with asynchronous peer code review for easy collaboration and automated refactoring to ensure more maintainable code, and an Agile team can minimize project drag and run more efficiently overall. Developers can focus their time writing innovative code, while testing teams spend their time testing how the features of the project work rather than uncovering mundane code issues and retesting these again and again.

Source code analysis may be right for your Agile team, particularly if you are finding your process being impacted by quality issues or security vulnerabilities, non-Agile friendly processes, and hard to maintain code. Implementing source code analysis within your Agile environment does not have to be disruptive. You can start small and analyze only a small project or a portion of a project. Compare the results against a similar project where these tools were not used. You'll undoubtedly find opportunities to save significant time and money by using source code analysis in your Agile development process.

**References**

[1] Manifesto for Agile Software Development http://agilemanifesto.org/

[2] Diane Kelly, Terry Shepard, An experiment to investigate interacting versus nominal groups in software inspection, Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research, p. 122-134, October 06-09, 2003, Toronto, Ontario, Canada.

# The Core Protocols, an experience report - Part 2

Yves Hanoulle, www.hanoulle.be

*Somewhere in a bar at a European Agile conference.* The conversation of the previous day appeared in the Methods and Tools Summer 2010 issue [1].

After a delicious breakfast, Yves entered the hotel bar at 9:57. Allison was already there. Allison had a big smile on her face when she said: "I'm checking in, I'm glad we talked yesterday, I'm glad we continue to talk today, I'm mad, sad, glad I kept thinking about our conversation, I'm afraid I have not slept enough, I'm in"

Yves replied energetically: "Welcome. I'm checking in, I'm glad I slept well, I'm glad I talked to my kids this morning, I'm afraid I will miss good sessions this morning, I'm glad I know that the best time I have at conferences is in the hallways. I'm in."

Allison: "Welcome. Last night you used **protocol check**, the way I understood it was that this protocol is to tell someone he used the protocols in a wrong way, is that correct?"

Yves: "Exactly, it's a way to remind people about the protocol they are using. Typically you will say protocol check and then you will add what you think is wrong; Even if you are not sure about a protocol, you can still say: protocol check, aren't you supposed to say 'haidi' after someone checked in? And then someone else can say: no the correct word is 'Welcome'."

Allison: "I felt bad when I got protocol checked last night. As if I was misbehaving"

Yves: "I understand. In a way you were misbehaving, isn't it more that I caught you misbehaving?"

Allison: "Yes, like I was caught with my hands in the cookie jar."

Yves looked at her, paused for a second and then said:" I see, if you use this metaphor, I understand you feel bad. The intention of a protocol check is not to make you feel bad, not even to put myself above you. Both of which happen when parents reprimand their children. That is a mix up of authority and accountability. It's not because I keep you accountable that I have authority over you. Most people who feel bad about being kept accountable, have this because they have some kind of history with authority figures who misuse their authority to force people in keeping their commitments. Actually in a bootcamp everyone has the right and the duty to protocol check everyone else (In other words students also protocol check the trainers). For me it is a good way to make sure I understand the protocols. As I said yesterday, outside a bootcamp, people use the core a lot less strictly, that is why I will not use protocol check a lot in other situations. I do use the spirit of protocol check a lot on agile teams."

Allison looked surprised: "Really? Please tell me more."

Yves:" All people new to Scrum make mistakes when they try to implement it. They might talk too long during the standup or not answer the three questions etc. As an agile coach I want team members to feel that they can and should check each other when they are not keeping their commitments. Personally I don't mind if they use Scrum, XP, kanban or some kind of mix, I do care that they do what they say they will do. And in my experience, the only way to keep my commitments is to have people remind me what my/our commitments are."

Allison:" Isn't that putting the responsibility elsewhere?"

Yves: "Not really. For one, as part of **the core commitments** we say that we should keep our commitments and help others keep their commitments. When I did my first bootcamp, we had three people in our team who were not keeping their commitments. I was very annoyed about this and at a certain moment four of us asked for help to Vickie, to find out what we could do.

She told us, that if we kept complaining among ourselves and not protocol checking these others, we were not keeping our commitments."

Allison sighed: "Wow that is harsh"

Yves: "Yes when she said that I felt like what the ..., and then I entered the team room, and these three people were actually doing what I hoped they would do."

Allison: "Looked like you were holding them back?"

Yves agreed "Yes, I think I was. Actually once people get used to this protocol, I see a lot of people protocol checking themselves. Instead of asking what the correct protocol is, they use what they think it is and then protocol check themselves and then ask for help. Which for me is a sign they are not afraid to make mistakes." Yves kept silent for a moment and then added: "Don't forget that the core, like Scrum, is a team pattern. Keeping the core commitments is also important for the whole team. Not only is it ok for people to check me. I expect them to do so. And the funny thing is, from the moment I know that people will keep me accountable for what I say I will do, I 'm more committed."

Allison questioned: "I know that there is some research [2] that says that people are less committed when they tell someone about their resolutions, seems like you want people to talk out loud."

Yves: "Yes I know about that research, in bootcamp we have a different experience. The difference is that we are talking about teams. When you say you commit to something, as a team we depend on that and we keep you accountable for that. (As I want to be held accountable for the things I commit to.) I think that is why we have a different result when students announce their goals."

Allison: "Will you say some more about these core commitments?"

Yves: "Yes, good idea. There are eleven commitments. The first commitment is **to engage when present**. This makes me think about 'open space technology' (OST) [3], with its law of two feet."

Allison: "What about the 'butterflies and bumble bees' [4] in OST, they don't engage in the discussions?"

Yves: "Good question, it is still in sync for me because butterflies and bees make their behavior clear in an OST. Maybe they are not part of any discussion, but they still engage very well into the event. In bootcamp that is the same. Let's continue with the next part of the first commitment**: know and disclose what I want, what I think, and what I feel.**"

"Would I use check-in to disclose what I feel?" Allison wondered.

Yves:" Exactly, that would be a good idea."

Allison: "and the 'disclose what I want part' is the personal alignment?"

Yves: "That's right".

Allison kept thinking out loud: "You said 'know' What if I don't know what I want, think, or feel?"

Yves: "When you don't know what you want/think or feel, that would be the right time to ask for help. My first bootcamp I learned asking for help when I thought I did not need help." Now Allison was confused: "Asking for help when you don't need help?"

Yves: "No, asking for help when I think I don't need help, or when I don't know on what I need help on. These conversations gave me some of the most interesting coach conversations I ever had. They helped me find what I wanted or felt. And that is exactly what the next part of this commitment is: **always seek effective help**."

Allison: "This first commitment seems like a complex commitment"

Yves:" I agree, this commitment has multiple parts that are grouped together. The commitment continues with **to decline to offer and refuse to accept incoherent emotional transmissions**. This is really a hard one. It basically says: behave as an adult, it is ok to have emotions, try to be aware of them and not say one thing and show something else. In bootcamp we don't accept passive aggressive behaviour. Although in theory I think everyone would agree with this, in reality it is very hard. And yes life is so much better when people are able to do this. A lot of this goes back to what Virginia Satir wrote in Peoplemaking [5], when she talks about nurturing families."

Allison: "If I remember well she says that only 4 out of 100 families are nurturing families."

Yves:" Yes I think that is what she wrote. In Bootcamp these teams seems to support each other like this. Part of this, is because the black hat's (instructors playing management) in Bootcamp are acting as perfect bosses [6]. While they ask for commitments and they ask for delivering upon those commitments, these bosses are showing coherent emotional behavior."

Allison: "Are you done with the first commitment?"

Yves: "No, being present is also about communicating your ideas. The commitment goes on: When I have or hear a better idea than the currently prevailing idea, I will immediately either **1) propose it for decisive acceptance or rejection, and/or 2) explicitly seek its improvement.**"

Allison: "If I'm getting that right, it means that I should not keep my ideas for myself, but immediately submit them to the team."

Yves: "Yes. The point is that an idea can't stay in someone's head: it should be immediately exposed, improved, accepted or rejected."

Allison: "We humans have a tendency to deal with our ideas in a different way than with other's."

Yves: "Yes. We tend to think our ideas are the best, or on the contrary, we tend to keep them for ourselves by fear of rejection. The point of the commitment here is that it's not important to know whose idea it is. Ideas should be exposed, improved, aggregated as fast as possible."

Allison: "That's not an easy commitment, I assume that Perfection game and decider are the protocols to use?"

Yves: "Yes, and now for the last part of the first commitment: **I will personally support the best idea, regardless of its source.**"

Allison: "That sounds logical, although I can imagine it is hard to see a boss and a colleagues idea at the same level."

Yves: "That is a big reason why this commitment exists. I know of a bootcamp where a 9 year old girl came up with better ideas than her 50 year old colleague. If a group gets it, you get really great outcomes."

Allison almost whispered when she said: "What if I don't like the idea?"

Yves: "Aha, that depends: if you have a better idea you propose it, if you don't, you support the currently best idea."

Allison a little firmer: "what if I don't have a better idea and I hope I will have one later?"

Yves: "You support the current best idea!"

Allison: "And when later I have a better idea I should, immediately expose it for improvement and/or validation"

Yves "Exactly so. You have paid close attention to what I [1] said yesterday about decider."

Allison: "I loved what Mary Poppendieck added about the least responsible moment."

Yves: "The second commitment is **I will seek to perceive more than I seek to be perceived**."

Allison: "The two ears and only one mouth rule."

Yves: "Yes two ears and two eyes, again an easy one to understand and yet very hard to do."

Allison: "How do you do this as a coach?"

Yves: "I use a lot a technique I learned years ago as a trainer, asking questions."

Allison: "Isn't that manipulation?"

Yves: "It is if you only want to hear the answer you want to. When I start asking questions, I have no idea where we will end up."

Allison: "How do you know what is the right question to ask?"

Yves: "I don't."

Allison: "You don't?"

Yves: "No: I don't think the right question exist."

Allison: "So any question will do?"

Yves: "Aha, not that either. By listening to the people, really listening what they say, and observing to see when their body language is not in sync with what they say."

Allison: "At Agile 2010, Esther Derby [7] distributed "Right Questions" to change organisations. What do you think about that?"

Yves: "Actually Esther's cards say: "Are you asking the right questions?" She never actually says that her questions are the right one ;-) I would call it hard questions, questions that a lot of people try to avoid. At a deeper level, the cards also send the message that asking questions is better then telling people what to do."

Allison: "What's next?"

Yves: "The third commitment is: **use teams**."

Allison: "Isn't everybody doing this?"

Yves: "I guess everybody says this because it is the right thing to do. The core especially says this for hard parts, I add also for boring things. As with boring things we make a lot of mistakes."

The CEO in Allison worries about the financial aspect when she asks: "Is that really the most efficient way?"

Yves: "The most efficient way probably not, the most effective way, yes. As a coach, I pair up with other people. When I do this with internal people, I also make myself much quicker no longer necessary. Now I'm more effective and I leave earlier. "

Allison: "That is in the long run more interesting for my company. I agree the ROI will be better for that way of working."

Yves: "A text I wrote earlier for Methods & Tools, was written with fourteen different people. Although not an easy task, the result was a lot better then if I would have written it alone."

Yves: "The next commitment is: **I will speak always and only when I believe it will improve the general results/effort ratio**. I personally find this a very hard one."

Allison: "What part do you find hard, the always or the only part?"

Yves: "Aha, good catch; actually both. Once I am on a flow about a topic I care about, it is hard to stop talking. Although as a trainer I work most by asking questions, when I am not in a training, it's easy to fall in to the trap of talking too much. In other cases, I listen and wait too long before giving my idea. I do this as I trust a team they will figure it out themselves, it is also holding them back from a great idea. It's a fine line to walk on."

Yves: "The fifth commitment is **I will offer and accept only rational, results-oriented behavior and communication."**

Allison looked confused: "Please say more."

Yves: "The idea behind this commitment is that we want don't accept irrational behavior, not from ourselves and not from other people." "And does that work?" Allison asks genuinely interested.

Yves: "The funny thing is that it's much easier for me with booted people than with non-booted people. It's not so much they expect it, more that they behave more result-oriented. I have a harder time to stay rational when people behave dogma wise: 'it was always like that ...'"

Allison: "Will you give an example of that?"

Yves: "Yes, I will. Recently in France we wanted to buy lots of stuff in a large grocery store. As the store was out of shopping carts, we used a few plastic baskets to gather our food. When we came at the cashier, she treated us as criminals because we wanted to use them to move our food to the car. I can understand that this is normally not done. She could not understand that this was an exception and that considering we had a lot of stuff to buy and three tired kids we wanted to speed up the process."

Allison: "What happened?"

Yves: "I was ready to walk away. (Which would have been irrational because we would have lost a lot of time.) Luckily my wife stayed rational so we ended up our shopping. (And never went to that same store again ;-) )"

Quickly Allison replied: "Is that part rational?"

Yves: "Good question, I think it is, if people treat me bad, I stop interacting with them. And that brings us to the next commitment: **I will disengage from less productive situations**: when I cannot keep these commitments or when it is more important that I engage elsewhere."

Allison: "As in checkout?"

Yves: "Exactly."

Allison: "Will you give an example from more important to engage elsewhere?"

Yves: "A few years ago, I was doing a tryout of a game together with my father. Fifteen minutes before we started, I received a phone call from my wife Els. She was at home, she called me to tell me we lost our baby. I stayed where I was and tried to help out with the game. I can't remember anything from that evening. The participants did not notice anything as my father saved the day. I should have check out and went home. Yes Els had support from family, but that was not the right support, I had to be there. It's the biggest mistake I made in my life, and it learned me very well that staying in a place where I can not be productive is not helpful."

Allison: "I'm sorry to hear that."

Yves: Thank you. We are OK with loosing the baby, it's the not checking out that bothers me. The seventh commitment is: **I will do now what must be done eventually and can effectively be done now.**"

Allison: "A kind of like the now habit? [8]"

Yves: "I have not read that book. Maybe one of the students that came up with this idea had read the book, I don't know. Although great ideas keep popping up, I don't think there is anything special about this one, we all know that if we can do something now, we should do it. The next commitment is actually the one that helps me with that: **I will seek to move forward toward a particular goal, by biasing my behavior toward action**."

Allison: "What does it mean?"

Yves: "Basically it says, prefer action over discussions. In the corporate world, we seem to prefer efficiency to effectiveness. Doing the things right instead of doing the right things. By doing so, we loose too much time into planning. I prefer to start doing stuff, by the time others have finished planning, I have already done half of the work. Using our velocity [9] in Scrum we can predict much better when we will be done after three iterations, then these people that are trying to created the perfect planning (and need the time of six iterations for it). I'm not sure which of the two I learned first, it does not really matter, for me they support each other."

Before Allison could say anything, Yves continued: "The ninth commitment is **I will use the core protocols or better when applicable.**"

Allison: "Or better?"

Yves: "Yes, I really like that part. The creators realize that every idea has an end-of-life. By taking this commitment you don't limit yourself. You can switch to better ideas, which is of course reality. If this would not be in there, I would drop my commitment immediately when I find a better way. By adding this, you can combine the core with better ideas. The ninth commitment also has a sub commitment: **I will offer and accept timely and proper use of the Protocol Check protocol without prejudice**."

Allison: "Aha the famous protocol check."

Yves: "The tenth commitment is: **I will neither harm - nor tolerate the harming of - anyone for his or her fidelity to these commitments**. For me this is a reminder that yes I am committed to these protocols, and also others. We have the tendency to think that we are doing our best and others are not. Even if actually everybody think this, this commitment reminds me that people will do good and I should not harm them when they do use these commitment even if I don't agree with their idea's. And the very best commitment we hold for last**: I will never do anything dumb on purpose**."

Allison: "That sound silly, of course you will not."

Yves: "It's actually a lot harder as it seems. It's in sync with behave rational, and unfortunately in the corporate world not everybody behaves rational. Remember the movie War games. [10] In the end the computer stops the nuclear war game because he realizes that nobody can ever win this game. Knowing when to stop is part of don't do anything dumb on purpose."

Allison: "Wow this a quite a collection."

Yves: "Yes, before we continue with the last protocols I want to talk about a few idea's that Jim has launched, if that is ok with you."

Allison: "I'm all ears."

Yves: "The first one is **don't flip the bozo bit** [11]"

Allison: "What is bozo?"

Yves: "Bozo is a clown, although he is well known in the US, not so much outside. The idea of don't flip the bozo bit is that we should take everybody serious. It's not because somebody does something we find strange that we should consider him a clown. It reminds me of the Retrospective prime directive [12]. Whatever we do, we should always remember that people did the best they could with the info they had at the time."

Allison: "Taking everyone serious is hard."

Yves: "The next idea might help you realize why it is needed: **Team == product**."

Allison: "Team == product??"

Yves: "Are you familiar with Conway's law [13]?

Allison: "Yes, it mean that the systems that are designed look like the organisations that design them."

Yves: "Exactly. Jim McCarthy took that to a whole new level, he says that the problems you see in a product, you will also find in the team that created the product. And you can see this at multiple levels. As the product of a management team, is the development team. As a coach I use this like this: when I see a problem with a team or a product, I look if I see a similar problem at my level. If there is distrust in a team, I wonder why do I not trust this team? Although I can not change other people, I can change myself. By trying to look at myself, I also remove the blame part from conversations and thus avoid the bozo bit."

Yves: "A less known protocol (only used during Bootcamp) is **the click protocol**."

Allison: "Click?"

Yves: "Yes click. It allows you to pause a situation and to discuss with your colleagues what would be the best options. The way this works: you say click and the bosses/clients freeze for one minute. During that minute you have time to discuss with your colleagues what you want to do. After one minute the people unfreeze (or you can unfreeze yourself by saying unclick) Although invented for bootcamp to offer you a safe place to experiment with different answer to your bosses. I know some people have used it at work and at home. And if you already said something stupid, you can use click-rewind, which offers you the possibility to take back some of the things you said and to repeat things in a different way."

Allison: "I'm not sure that when someone says something bad to me if he click-rewinds what he said I would feel better."

Yves: "I had used it in bootcamps on me both as students and trainer and there it works. I have to admit I haven't used it yet at home."

Yves: "Let's move to more well known one, **Intention Check**, In the corporate world and in my personal world, I encounter situations where people get angry at other people, because they presume the other person wants to hurt them. In my work as an agile coach, I do root cause analyses of problems between people and teams. Some of these high tensions could be prevented if people would ask for the intention of the other side. Intention check is used for this."

Allison: "I'm not sure I understand, will you give me an example?"

Yves: "Great idea. Let's say a team is running an iteration retrospective, talking about the difficulties of the project and what can be improved."

> Jane, who is the Product Owner of the team says: "We are at iteration #3 and yet we only have four stories done, which is half what we were planning. OK. I guess it's a diesel."
>
> Everyone reacts to this last sentence. Then Chris asks her: "Jane, what is your intention by saying 'I guess it's a diesel' ?"
>
> Jane pauses for a second, then says: "I want to alarm you guys about the fact that I was expecting the team to work faster after the first iteration, but it ain't so, and it looks like there's nothing to be done about it. I guess I convey it the wrong way, but I'm pretty worried right now."

Allison: "Should I see this as a special kind of protocol check?"

Yves:" Interesting question: if you see it as a kind of commitment check, then you would blame the other person. For me Intention check is really more an open question, to understand what the person really wants to say. Let me give you another example:

> During a stand-up a developer says proud that he finished the superdupee gizmo admin website.
>
> A tester responds that he already found a bug in the module.
>
> The developer feels internally angry about the remark of the tester. He feels that the testers wants to make fun of him by telling it during the stand-up and not before. The developer asks the tester: what is your intention about waiting to tell me about the bug till the stand-up?
>
> Before the tester can answer the question, the developer realizes that the question itself was asked with an intention. So he withdraws his question and says he intention checks his own question; His intention was to show that the tester wanted to ridicule him.

Allison: "He could have say click! Rewind!"

Yves: "Exactly! This last example shows you that you can (and should even) intention check yourself. That is not easy."

Allison: "I can see that."

Yves: "Another important protocol is the **investigate protocol**."

Allison: "Sound like CSI."

Yves: "No-no, not that kind of investigation. Act as if you were a fascinated inquirer, asking questions until your curiosity is satisfied. With CSI, the investigators have an agenda when asking questions, the idea is to not have an agenda here. To ask open questions. Questions like: What about X makes you Y Z? Would you explain a specific example? How does X go when it happens? What is the one thing you want most from solving X? What is the most important thing you could do right now to help you with X? The idea is to ask questions that will increase you or your partners understanding."

Allison: "Your partner?"

Yves: "Yes, because in the end you are helping him by doing an investigation. I'm no expert in Appreciative inquiry [14], students tell me investigate is very much in sync with this."

Allison: "Nice this means there are actually a lot of books available that can help with this."

Yves: "Yes, like I told you, the core protocols use idea's coming from everywhere."

Allison: "I start to understand this."

Yves: "If you do, then it's time to talk about the **personal alignment**"

Allison started smiling.

Yves: "Personal alignment is one of the core things in a bootcamp. We ask people to focus on what they really want."

Allison: "What they want in life or in work?"

Yves: "Both, although I am not talking about physical wants."

Allison raised her left eyebrow a little: "Will you give me an example of such a want?"

Yves: "I will in a moment, just bear with me for now. Imagine you want something and then ask yourself what is blocking me from having this."

Allison: "With blocking me, you mean internally?"

Yves: "That is right. A personal block is something you find within yourself. It does not refer to circumstances or other people. Now try to look for a virtue, that if you had that virtue, the lock would not blocking you."

Allison: "Would that not make the virtue my real want?"

Yves smiled: "You got it. You can do this exercise a few times, until you don't find anything that blocks you from having what you want. That is your (current) personal alignment. The most chosen alignments are: Integrity, Courage, Passion, Peace, Self-Awareness, Self-Care or Fun."

Allison: "You said this was both for work and personal, please say more."

Yves: "The idea is that what you need most, is needed at both places. You can't want integrity at work and fun at home."

Allison: "I agree that would sound strange. Why do we need personal alignment?"

Yves: "The idea is that if team members investigate each other on their personal alignment, they come into the shared vision state I talked about yesterday. Daniel Pink [15] talks about the fact that people are motivated by personal goals, I would add that teams goals motivate teams, and these team goals are derived from vision. Actually key elements to get into that share vision state, is check-in (to disclose what you feel.), decider (to make sure the team advances) and alignment."

Allison: "And that alignment is done by investigation of the current personal alignments?"

Yves: "You got it completely."

Allison: "Let's drink to that."

They finished their coffees and went to the conference hall and enjoyed the rest of the conference.

**References**

[1] http://www.methodsandtools.com/archive/archive.php?id=106

[2] http://www.hanoulle.be/2010/09/derek-sivers-keep-your-goals-to-yourself/

[3] http://en.wikipedia.org/wiki/Open_Space_Technology

[4] http://www.openspaceworld.org/cgi/wiki.cgi?WorkingInOpenSpace#anchor600799

[5] http://www.amazon.com/Peoplemaking-Condor-Books-Virginia-Satir/dp/ 0285648721

[6] http://www.mccarthyshow.com/Episodes/BossWhispering/tabid/84/Default.aspx

[7] www.estherderby.com

[8] http://www.amazon.com/The-Now-Habit-ebook/dp/B003SHERX8/

[9] http://www.scrumalliance.org/articles/39-glossary-of-scrum-terms#1110

[10] http://en.wikipedia.org/wiki/WarGames

[11] http://en.wikipedia.org/wiki/Bozo_bit

[12] http://www.retrospectives.com/pages/retroPrimeDirective.html

[13] http://en.wikipedia.org/wiki/Conway%27s_Law

[14] http://www.amazon.com/Appreciative-Inquiry-Positive-Revolution-ebook/dp/B001T0I4QW

[15] http://www.hanoulle.be/2009/09/daniel-pink-on-intrinsic-extrinsic-motivation/

Allison is a fictional person. This conversation is based on talks that Yves had at agile conferences over the last five years.

Yves asks one agile coaching question every day on http://twitter.com/Retroflection . Questions are created by John McFayden, Dusan Kocurek, Martin Heider, John Gram, Deborah Preuss, Christopher Thibaut, George Dinwiddie, Diana Larsen, Ine De handschutter, Bob Marshall, Yves Hanoulle, you ?

This article was written with the help from Jim & Michele McCarthy, Paul Reeves, Christopher Thibaut, Adam Feuer, Esther Derby, Lillian Nijboer, Michael Sahota, A big kudo's to Emmanuel Gaillot who initiated the conversational style.

Yves gives free life time support on this article: send your questions to core@hanoulle.be If you want more people to respond, you can connect to the CoreProtocols user group: TheCoreProtocols@yahoogroups.com

# tinyPM

Franco Martinig, Martinig & Associates, http://www.martinig.ch/

tinyPM is a lightweight and smart agile collaboration tool with product management, backlog, taskboard, user stories and wiki.

**Web Site**: http://www.tinypm.com/
**Version Tested:** tinyPM version 2.3, tested on Windows XP during a period from August to September 2010
**System Requirements:** Java 6, Tomcat 5.5.x/6.x, MySQL 5.x, machine with 1 GB of RAM (recommended)
**License & Pricing:** Commercial, free until 5 users, 12,50 euros/user/month above;
see http://www.tinypm.com/pricing for details
**Support:** support forum for all and e-mail for paid version
**Languages:** tinyPM is available in English, German, French, Polish and Portuguese.

## Installation

To run the installation, you need to have the Java JRE (Java Runtime Environment) working on your machine. If this is not the case, you can download it from http://www.java.com/. You need to setup some path variables in your system.

tinyPM is written in Java and thus should run on every machine with a Java Virtual Machine. The installation procedure will depend on you system and if you have already Tomcat installed. On my Windows XP system, I choose the standalone installer of tinyPM version bundled with Apache Tomcat and HSQLDB. Installation is a quick procedure, only asking for license approval, directory choice and shortcut creation in the menu. tinyPM does not create a shortcut on the desktop.

Using the "Start tinyPM" option in the program menu will start the Tomcat server and open tinyPM in your browser. It happens that the browser is "faster" than the Tomcat initialization process and creates a "page not found error". In this case, you just need to wait for the end of the Tomcat start-up process and reload your browser page pointing to localhost. The first time that you use the software you need to go to http://support.tinypm.com to register and fill some information to receive the license for your free five users version. You are then able to download a license file that you need to start using tinyPM.

For a clean exit of the application, the "stop tinyPM" command will stop the Tomcat server instance.

Another command from the Start Menu folder allows you to open the tinyPM database. You may need it when you upgrade. tinyPM runs initially with HSQLDB that should not be used as a production database. Documentation describes how you can switch to MySQL.

## Documentation

Documentation is available on line on http://documentation.tinypm.com/. The blog posts that discuss the features augment it. The interface is rather intuitive and consistent, thus allowing getting rapidly a grasp on the main tools features without having to read the manual. A getting started could help you in case you need it.

## Configuration

There isn't so much infrastructure configuration to do when you use tinyPM. A single setting screen allows managing application infrastructure like license key management or identifying the mail server information to inform project members of certain events.

## Features

- **Roles**

The role management screen allows to create your own project roles (product owner, scrummaster, developer, etc.) and attribute activity permissions, like "creating a project" or "delete user story". These will allow users with these roles to perform the activity in tinyPM. You have a complete flexibility in defining user permissions here.

- **Users**

The user creation screen records basic user information and which notifications should be sent to the user after the occurrences of certain events, like the deletion of a user story for instance.

- **Project**

Creating a project is rather simple and project members can be associated to the project when it is created. In the project setting screen, you can define your budget metrics like the estimation rules for user stories or the project budget. This screen also allows you to assign to each user story an automatic list of tasks, like the creation of functional tests or documentation.

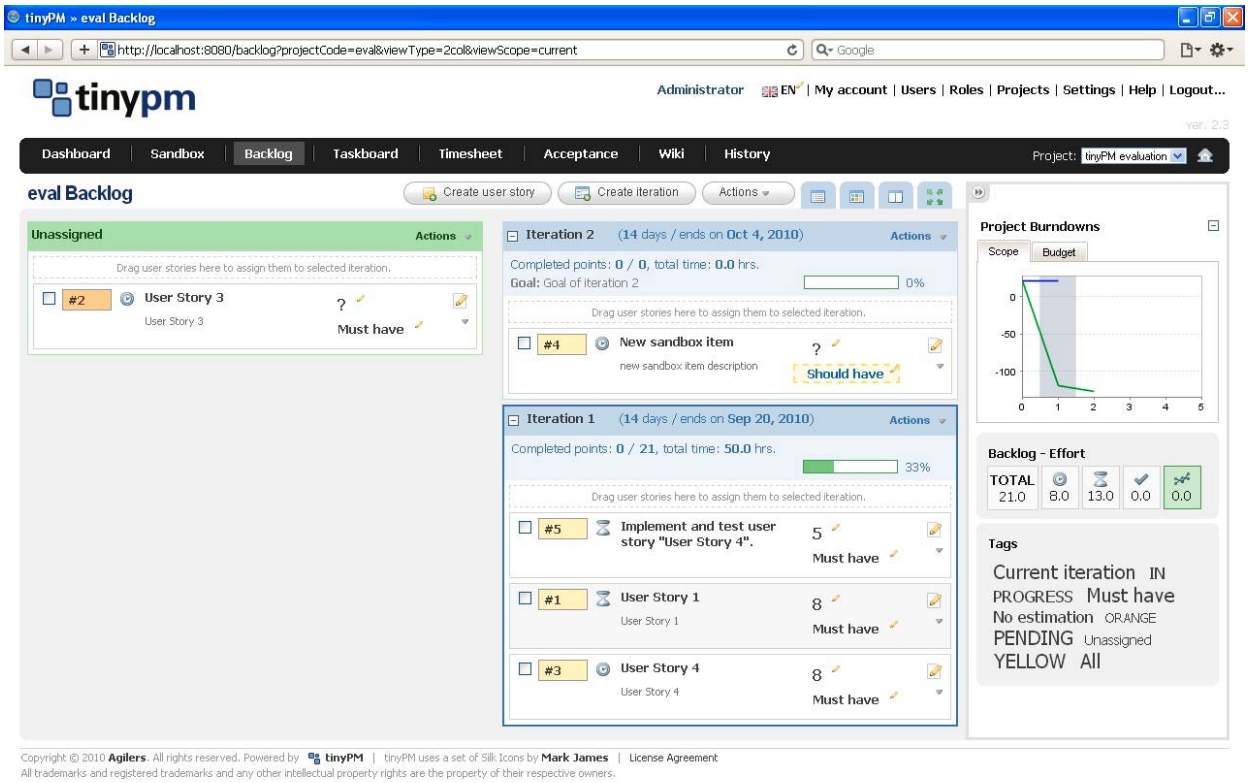



Figure 1. Backlog Screen

This avoids to manually creating the same list of task when you create user stories. Most of these settings are optional. This means that if you don't care about budget tracking or do not estimate tasks, you don't have to define them and tinyPM will hide respective parts of its interface.

The project area is organized around the life cycle of a project. A sandbox allows managing features ideas. These sandbox's ideas can be created inside tinyPM or imported from Jira, UserVoice or an e-mail address. They can be later transformed into user stories.

You define user stories and attribute them to iterations in the backlog area where these two items are created and managed. Story points, a story owner and an acceptance user can be associated to a story.

The different tasks associated with user stories are managed in the taskboard until completion. Tasks can be automatically created after a user story is defined if you decided so in your project setting screen. They might be upgraded to the user stories status. You just have to drag them to make them move between the different status available: pending, in progress and completed.

The timesheet section allows to record the hours spent and an estimation of the hours left before completion for tasks. This is later represented on budget chart and iteration burndown chart.

Once completed, user stories can be accepted to achieve a "done" status. A wiki allows the team to share information.

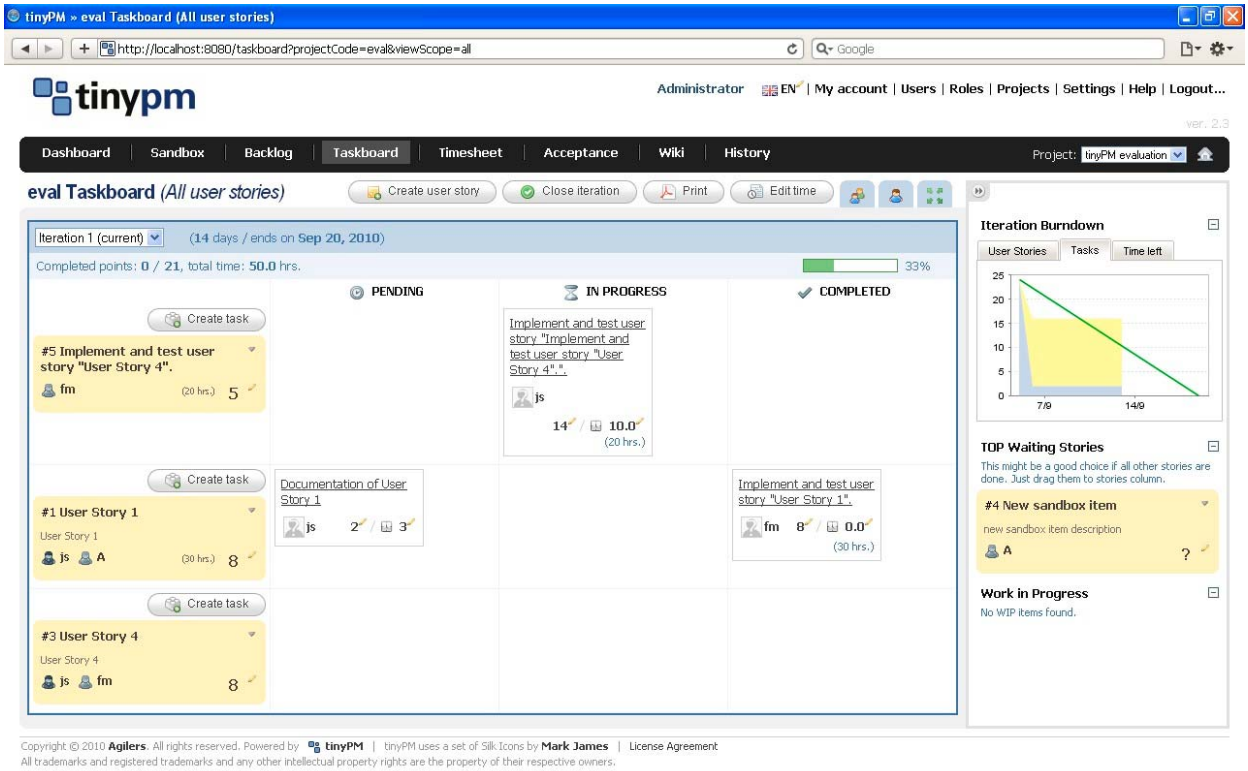



Figure 2. Task Board Screen

You can attach files (pictures, text, etc.) to stories, tasks and wiki pages to better document the requirements. Story cards can have different colors and can be tagged which helps organizing bigger backlogs.

Changes made to user stories and tasks are versioned,. You can therefore track what has changed, when it happened and who did it in the history section. You can be also be notified about changes via e-mail or you can subscribe to your personal RSS feed that will show changes in your projects made by other team members.

## Charts

tinyPM allows to track your project at different levels (project and iteration) using different references, either the initial effort or the current total effort. Charts will appear on the different project screens according to the project settings that define the baseline to compute them and the context, either the project or the iteration level.

Depending on the level of detail that you choose for your project (defined budget, tasks estimation) tinyPM charts will also adapt and give you more or less detailed view on your project status.


Figure 3. Burndown Chart for Iteration

**Conclusion**

tinyPM is a well-organized tool that allows you to rapidly manage your agile projects in a simple but functionally complete way. Although it uses agile terminology, the tool is also suited for every type of project if you prefer to translate "user stories" in "requirements" and "iterations" in "phase".

# JUnit

Axel Irriger, axel.irriger [at] gmail [dot] com
cirquent GmbH, http://www.cirquent.de

JUnit is a Java library for testing source code, which has advanced to the de-facto standard in unit testing. By using JUnit, you can assert that methods in your Java code work as designed, without the need to set up the complete application. From JUnit 4.0 on, the minimum supported Java version is 1.5.0. JUnit version 3.8 and earlier supports Java up to version 1.4. To apply the JUnit principle for testing in specific domains such as XML or databases, special extensions exist.

**Web Site:** http://www.junit.org and http://junit.sourceforge.net (http://www.sourceforge.net/projects/junit/ respectively)
**Version Tested:** JUnit 3.8.2 and JUnit 4.8.1 on Windows XP with Java 1.6.0_20
**License & Pricing:** Open Source
**Support:** User mailing list (http://tech.groups.yahoo.com/group/junit/), various forums and printed books

Since annotations add a fundamental new programming feature to the Java programming language, JUnit explicitly supports those with version 4.0 and above only, so both major versions (with and without annotation support) will be covered in separate sections.

**Installation**

The JUnit distribution is downloaded as a single JAR file, which can be placed in any folder. To actually start using JUnit, the JAR must only be put in the classpath of the application to test.

If you are using Apache Ant (http://ant.apache.org) as the build tool in your environment, just include junit.jar to your CLASSPATH. Then, you can use the <junit/> Ant task to actually execute tests. In your build.xml file you can add the following to one of your targets to enable JUnit testing:

```
<junit>
  <test name="my.test.TestCase"/>
</junit>
```

This will execute the class *my.test.TestCase* as a JUnit test. The JUnit Ant task does support both JUnit 3.x and JUnit 4.x versions.

If you are using Apache Maven (http://maven.apache.org) as the build tool in your environment, just add the following to your pom.xml:

```
<dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.2</version>
      <scope>test</scope>
    </dependency>
</dependencies>
```

This will add JUnit to the build lifecycle. If you already have a “dependencies” element, just add the single “dependency” element to your POM. The unit testing will start by executing the following command:

```
mvn test
```

By changing the version above in pom.xml, you can easily switch to different JUnit versions. Maven supports both JUnit 3.x and 4.x.

**JUnit 3**

JUnit version 3.x is the tool of choice if you need to test your application in a Java 1.4 environment or earlier context. To understand the basic mechanism of JUnit, take a look at Figure 1.

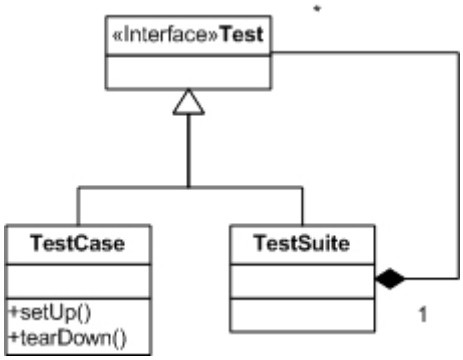



Figure 1 Design of JUnit 3.x

The basic principle of JUnit is the *Test* interface. Everything is a test and thus, tests can be grouped together. This concept and its implication will be discussed later on. If you want to actually test a piece of code, you create a class, extending *TestCase* and add various test methods to it. Consider the following, simple Java class:

```
public class DummyPojo {
      public int addValues(final int firstInt, final int secondInt) {
         return firstInt + secondInt;
      }
}
```

To test this class, you can create a TestCase like this:

```
public class DummyPojoTest extends TestCase {.............    /* (1) */
      DummyPojo pojo;

      public void setUp() { ..............................    /* (2) */
         pojo = new DummyPojo();
      }

      public void testAdd() { ............................     /* (3) */
         int result = pojo.addValues(5, -5); ..............    /* (4) */
         assertEquals(0, result); .........................    /* (5) */
         assertNotEquals(“Result is 10!?”, 10, result);      /*  (6) */
      }
}
```

In (1) you create a class which extends the base *TestCase* class. Then, in (2), a generic *setUp* method is called. This one is provided by *TestCase* and is invoked per test method to be executed. The method is meant to initialize everything which is needed for the test methods invoked later. This may be instance variables, authentication and such things.

The same applies for the method *tearDown*, which is not shown here. This method is called after a test method was executed to clean up the test environment, such as removing data from a database. In line (3) a test method is declared. Each test which should be executed automatically must begin with "test". Line (4) then executes a method under test. The lines (5) and (6) then assert expected values. That is, if the value 0 does not equal *result*, the test is broken and JUnit will stop proceeding. Line (6) adds readability by providing a message which is shown if the assertion breaks.

By using such an approach, you can not only test the "good cases" of an application, but also the "bad cases" and invariants, as well. This way, the contract of a class can be asserted fairly easily, that is the complete expected behavior.

If you have many tests for various classes, testing everything and executing all tests can be cumbersome. Therefore, JUnit offers a principle to logically group tests and thus simplify its execution. To logically group *TestCase*s together, you create a *TestSuite*. And since a *TestSuite* also implements the *Test* interface, various suites can also be grouped together. Using this strategy, a logical hierarchy of tests can be created.

Such a *TestSuite* can then be executed like any other test in JUnit. If a *TestSuite* is executed by JUnit, all tests configured for this *TestSuite* will be executed. By applying this mechanism, you can easily test a whole subsystem by only starting one test, which is very convenient:

```
public class AllTests extends TestSuite
{
  public static Test suite()
  {
    TestSuite mySuite = new TestSuite( "My subsystem test" );
    mySuite.addTestSuite( mypackage.DummyPojoTest.class );
    // ... add more tests here
    return mySuite;
  }
}
```

**Extending JUnit 3**

If you want to provide additional functionality, you can easily extend the framework by extending the *TestCase* class. Using this method, you can add additional methods for asserting things or simplify 3rd party framework integration.

**JUnit 4**

If you are already familiar with JUnit 3.x, the transition to JUnit 4 is rather simple. The following transition table helps to migrate from JUnit 3.x to JUnit 4.x:

| **JUnit 3.x** | **JUnit 4.x** |
|---|---|
| Base-Class *TestCase* | (no longer necessary) |
| Method "*setUp*" | Annotation *@Before* |
| Method "*tearDown*" | Annotation *@After* |
| Test-Method "testXYZ" | Annotation *@Test* |
| (no equivalent existing) | Annotation *@BeforeClass* (Once-and-only setUp) |
| (no equivalent existing) | Annotation *@AfterClass* (Once-and-only tearDown) |
| try { … fail() } catch (Exception e) { assertTrue(true) } | Annotation *@Test(expected == Exception.class)* |
| Custom implementation for testing long-running operations and timeouts | Annotation *@Test(timeout = 1000)*<br><br>Timeout setting in milliseconds. |
| Custom implementation to ignore test using *TestSuite* | Annotation *@Ignore* |

To understand its usage, here is a simple example:

```
public final class MyTest {
      private DummyPojo pojo;
      @BeforeClass public void initialize() {

      System.out.println("This is printed only once");
      }
      @AfterClass public void destroy() {

      System.out.println("The last thing printed");
      }

      @Before public void setUpMyTest() {
      pojo = new DummyPojo();
      }

      @Test public void adder() {
      final int result = pojo.addValues(5, -5);

      assertEquals(0, result);
      }

      @Ignore @Test public void multiplier() {
      // This test seems broken
      final int result = pojo.addValues(-5, -5);

      assertEquals(25, result);
      }
}
```

**Documentation and Literature**

The general documentation of JUnit is available from the projects homepage. A more tutorial-like approach can be found at
http://java.sun.com/developer/Books/javaprogramming/ant/ant_chap04.pdf.

If you prefer printed documentation, the commercially available book “Pragmatic Unit Testing In Java With JUnit” (http://www.pragprog.com/titles/utj/pragmatic-unit-testing-in-java-with-junit) is a good choice. As a desk reference, consider the “JUnit Pocket Guide” (http://oreilly.com/catalog/9780596007430).

**Summary**

JUnit is the de-facto toolkit for testing Java applications and, due to its simplicity, should be the first-choice when testing your applications. Besides the power which can be leveraged out-of-the-box, added value is provided when integrated with third-party extensions, such as:

- dbUnit, which can test database operations like CREATE, INSERT or database triggers
- xmlUnit, which simplifies testing XML based applications, such as comparing XML documents, evaluating XPaths, et cetera.

To me, JUnit is in invaluable tool which every Java developer should know about and use heavily in his or her daily work.

# Bromine

Rasmus Berg Palm, Jeppe Poss Pedersen, Visti Kløft,
Bromine Foundation, http://brominefoundation.org

This article is about Bromine, an open source functional web test automation tool. We'll go through the background for Bromine and explain what Bromine is and why you should (or shouldn't) use it. Finally we'll introduce two real world scenarios in which Bromine is used.

If you have worked with test automation of web sites you have most likely heard of Selenium. For those of you who have not, Selenium is an open source test automation tool that has a lot to offer, in particular capture / replay functionality and means to run your test in a distributed environment. This is the foundation that Bromine builds on. In short Bromine can be described as a management tool for Selenium. It offers is a way to organize your test scripts and easily run them in different browsers and operating systems.

**Web Site:** http://www.brominefoundation.org/
**Version Tested:** Bromine 3 RC2
**License & Pricing:** GPL version 3 & Free
**Support:** support forum at http://forum.brominefoundation.org/

Once installed you can access Bromine through your browser. From here you can create new projects that can contain an array of requirements and test cases. For each test case you can upload a Selenium test written either in Java or PHP. Once you have uploaded the script head over to the Test Lab where you will be able to run either a single test case or a complete suite of tests which you have organized under a requirement. If you have the necessary capacity, the tests will be run in parallel saving you time compared to running them sequentially via the IDE or on a single machine through Selenium RC.

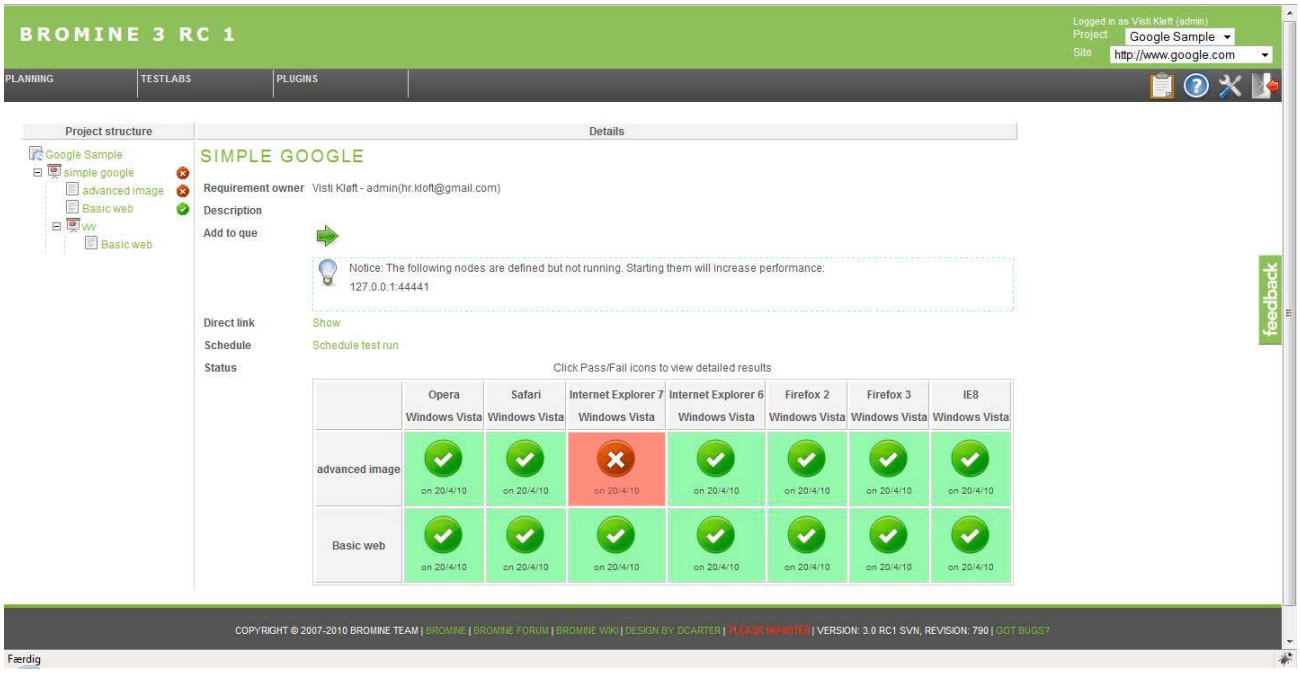



Figure 1 - Bromine showing the results of two test cases in seven OS/browser combinations

## Background

So, how did Bromine come to be? If you have ever worked as a software tester you know how daunting it can be testing that the same menu link still works, build after build. This is where Selenium came into the picture. It is open source, so there was no need to invest in a proprietary automation tool which could easily have cost several thousand dollars per user in acquisition and then come the service agreements on top of that. However, we soon realized it was complicated to organize, run, and share tests between testers as Selenium did not have anything to offer on that part. We then began the development of what was then called TRM (Test Result Manager). This was back in 2007. From here on we kept adding features to TRM and then in early 2008 we decided to rework the whole thing and renaming the project Bromine. We decided on Bromine because it is the 35th atomic number in the periodical table next to Selenium. This phase of the development continued to about mid 2009 where we decided to completely rework Bromine again, this time around using a PHP framework (CakePHP) hoping that it would attract more people to contribute as it would be easier to adopt the programming style. This led to Bromine 3 Beta being released autumn 2009, which entailed a signification amount of improvements over the last installment of Bromine. Today you can download Bromine 3 RC2 from http://seleniumhq.org or http://www.brominefoundation.org/ and in the coming months we are planning to release Bromine 3 Final.

## What is Bromine and why should I use it?

Bromine is not perfect. Actually, it's quite far from what we would consider perfect. It will fit some organizations just great and others less so. We'll try to explain what the pros and cons of Bromine are in this section. But don't take our word for it; give it a try. It's free after all.

First off Selenium does functional web testing and Bromine makes using Selenium easier. Selenium is probably the most used functional web testing tool out there. It works by starting a browser, injecting it with a lot of JavaScript functionality and then executing JavaScript through the browser which simulates user actions. A command could be 'click("submit button")' or 'verifyElementPresent("some div")'. The words in quotation marks are what are called locators, i.e. some string that identifies a part of a web site. This is mostly element ids or xpaths. Selenium comes in a couple of shapes. The most used are Selenium IDE, a Firefox plugin that does record and replay, and Selenium Remote Control (RC), which is a Java server that listens for commands and upon receiving them opens and controls a browser and sends the results back for each command. Lots of more info on selenium can be found at http://seleniumhq.org

What Bromine does, and does well, is executing selenium RC tests, storing the results and presenting them in an easily understandable way, all as an easily accessible PHP/MySQL web application. This is the core functionality and this is what we've spent most of our time coding. The rationale behind this is that when we first started using Selenium RC, these were the most frustrating issues we faced. What good is a test if it's hard to run and the results are difficult to save and interpret? From this core functionality we have begun to develop the business side of Bromine; requirement and test case handling. This side is fairly bare bones at the moment, but we're planning on to focus our efforts in this direction over the next couple of releases.

A standard workflow in Bromine goes something along the lines of this: **Management** creates requirements and details which operating systems and browser combinations these should be tested in. **Test manager** creates test cases to cover these requirements. The **tester** records a script in selenium IDE doing what the test case specified. Tester exports the test script into Java or PHP and uploads it to Bromine.

The tester selects the test case (or requirement, or even an entire project) and presses the run icon, which will launch the tests in the required OS/browser combinations (given that you have test machines setup having the required OS/browser combinations of course).

The tests are scheduled, much like a printer job queue and executes when the test machines are available, i.e. not running any other tests. The test script drives the browser and Bromine captures every command sent between the test script and the RC server as well as the status of the commands. This is then saved as a test run on the given test case and OS/browser combination. The test case will be flagged as passed or failed based on the results of the latest test run, and in turn any requirements the test case belong to will be flagged as passed or failed as well. The poorest result takes priority so that a single failed test case in a requirement with lots of test cases will mark the requirement as failed.

Bromine is unique in that it acts as a GUI for Selenium, closing the gap between technically minded tester and less technically minded management. In addition, it is free and open source. We don't feel that there are any other tools out there that fit this description.

Included in the Bromine RC2 release was Hudson and Sauce Labs integration, which are some features we're pretty excited about.

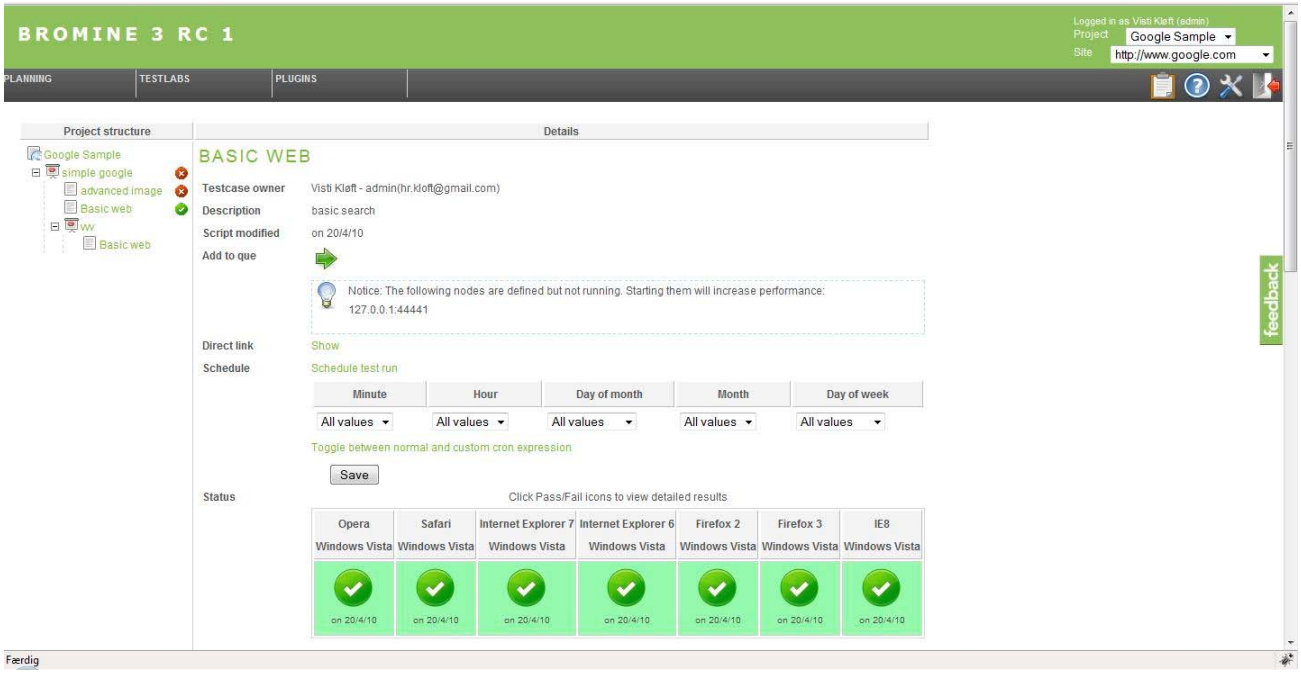



Figure 2 - Bromine with the scheduling pane open for a single test case

## A real life scenario at Test Lab I/S

Test Lab I/S is owned by Visti Kløft and Jeppe Poss, two of the Bromine founders, and specializes in automated testing. The following is an example of why and how Test Lab uses Bromine, based on a couple of large projects. Unfortunately it's impossible to be more detailed due to NDA's with the customers.

How does Test Lab I/S use Bromine in the real world? Bromine helps us solve issues in big test setups. We have tested large web applications which included large number of sites, many localized versions, browsers, and large number of test cases.

In Bromine you organize your test cases and with one click you can start hundreds of test runs in different browsers. Imagine a project with a hundred test cases that needs to be tested in four browsers (FF3, IE7, IE8 and Safari) on three environments (test, QA and production). This gives 400 test runs each time we need to run our regression tests and that creates a few issues:

- Capacity issue (load balancing)
- Maintenance of the test scripts
- Overview of the current state of the application (does all the features work?)

**Capacity issue**

Running 400 test cases takes time and to solve this issue load balancing is needed. Bromine does just that. Bromine looks at the available Selenium RC (called nodes in Bromine) and will try to run as many tests as possible at the same time.

**Maintenance of the test scripts**

Having a hundred test scripts requires a lot of maintenance and instead of doing our own revision control you can just use SVN (or similar tools). All the test scripts in Bromine are in a certain folder and if you put that folder under revision control you can easily maintain your script without having to learn another new tool.

**Overview of the current state of the application**

Bromine gives us an excellent overview of the current state of the application. While it's not out there doing exploratory testing for us, we have it setup to test for known bugs as well as the core functionality of the application under test. Are all the pages online? Does each page links to the right pages? Does the rate, buy and signup functionality work? Having certainty on these issues frees up time for tasks humans are good at; finding tricky bugs and making developers miserable.

**A real life scenario at Roskilde Festival**

The following case study is a blog post by Anders Nickelsen (twitter.com/anickelsen), the leader of the QA-team in the IT-development group at Roskilde Festival, reproduced here with permission.

**The organization**

The Roskilde Festival located in Roskilde, Denmark, is the second largest music event in Europe and entertains more than 100.000 guests over 4 days (+4 day warm-up) annually. The festival is made possible by approximately 25.000 volunteers that work either before, during or after the festival. Amongst these are 40 volunteers working throughout the year to develop and operate 10+ specifically designed web-based information management systems used by the festival.

**The challenges of volunteer developers**

As with many open-source projects, the group of volunteer developers is very dynamic, since people can leave or join rather spontaneously. Therefore, a key element is to provide new developers a simple entry into the development process, and ensure that systems stay stable even if developers drop out.

Contrary to many open-source projects, developers are not end-users, so there is no obvious itch to scratch [1] until new developers have experimented with the systems and technologies. Also, once new developers have made contributions to the systems, they need to see it used by end-users in production soon after to avoid losing interest in the project.

**Quality assurance by continuous integration**

To provide sandboxes for experimenting and short 'time-to-production', we have built a versatile quality assurance framework to allow new developers to dive right into changing codes and try out new things, without the risk of breaking any of the running systems. Through a series of verification steps, both manual and automated, all system contributions are continuously integrated into the production mainline in a lean and reliable way.

**The development process**

Our setup builds around the philosophy that "developers need flexibility, operators need stability". To achieve this we use a chain of environments. As contributions flow through these environments - from the developers hand into production mainline - the developers flexibility becomes limited as focus is turned to stability of a release. The chain consists of the following 4 environments that are focused around per-system subversion repositories.

*Development environments*: VirtualBox images with replicates of the production environment that can be used locally by developers to try out new things. The developers commit contributions from the images into subversion projects.

*Preview environment* (alpha-test): Central servers with exports of the subversion projects that allow developers and change managers to check out subversion revisions to preview individual changes and run unit tests

*QA environment* (beta-test): Central servers with exports of the subversion projects that allow beta-testers to verify releases through integration tests and acceptance tests, and release managers to test release procedures. QA environments are separated from preview environments because they are also used for end-user education, which requires them to be more stable and less frequently updated.

*Production environment*: Central servers that host production versions of the systems, which are used by end-users.

**The test framework**

The test part of the continuous integration framework consists of the following tools:

- Hudson
- Selenium
- Bromine 3
- Sauce OnDemand by Sauce Labs

Hudson automatically triggers basic testing in the preview environment on every commit to Subversion. Commits are considered atomic and only include one change each, which is validated manually - also in the preview environment. Atomicity enables easy rollback of changes and preserves traceability.

Integration tests and end-user acceptance tests are recorded using Selenium IDE and collected in Bromine. Before the release of a new version, Hudson triggers Bromine to execute the entire test-suite of Selenium tests from the previous release as a regression test on the new system deployed in a QA environment. The tests are executed either on dedicated internal Selenium clients, or in the Sauce Labs cloud via Bromine's integration with Sauce OnDemand. Bromine produces result reports which provide an overview of the success rate of the tests. The result reports are automatically integrated into Hudson, which makes the test result overview fast and clean. Futhermore, end-user testers make exploratory testing in the QA environment to validate the released bug-fixes and features.

When the new release is approved, the Bromine test suite is updated to include new Selenium tests for the new bug fixes and features.

After release, the wheel turns once again with planning of a new release, where everyone - new as old - can participate, with small or big changes, thanks to the integrated quality assurance framework.

**Conclusion**

Bromine is an open source functional web testing tool. It is based on Selenium which is also open source and probably the most used web automation tool out there. Bromine's main strength is its ability to make the life of the Selenium powered tester easier. Bromine provides an easy way to organize and run your test scripts. It also gives you the opportunity to tie in the results of the test scripts with requirements and test cases. Bromine is a web application, which is easy to access and use for a large number of users. A headless Linux server with 256mb of RAM will probably host Bromine satisfactorily for an average small business (up to 5 users online simultaneous, more will require additional memory). Bromine integrates with some of the cool players in the industry like Hudson.

Bromine is not a magical tool which will make all your problems vanish in the blink of an eye. It's just a tool and tools are nothing without methods. You'll need to remember your ISTQB syllabus, and recall that automation is hard work and requires more abstraction than record and playback. It definitely pays off in the end though, as we hope our case studies have shown.

[1] The Cathedral and the Baazar, Eric S. Raymond, O'Reilly Media, ISBN: 978-0596001087

# Agilefant

Pasi Pekkanen, Reko Jokelainen, {firstname.lastname}@soberit.hut.fi
School of Science and Technology, Aalto University, Helsinki, Finland

Agilefant is backlog management tool suitable for both product- and project-oriented organizations.

**Web Site:** http://www.agilefant.org/
**Version Tested:** Agilefant 2.0.2
**License and Pricing:** Open Source, MIT-based license, free
**Support:** Website forums

## Installation

The prerequisites for installing Agilefant on your computer are the Java runtime environment version 1.6, Apache Tomcat 5.5 or 6, and MySQL 5. For full installation instructions, see http://www.agilefant.org/wiki/display/AEF/Installation+guide

First, download the package from http://www.agilefant.org/wiki/display/AEF/Downloads. Inside the zip package, you will find a *war* package (web application archive) and the SQL script for creating the database (*create-db.ddl*). Create a database named *agilefant*. Create a MySQL user named *agilefant* with the password *agilefant*, and grant them access to the database. This way, you do not need to change the database properties for the package (see the *Configuration* section for more). Run the create-db.ddl script in MySQL console ('SOURCE create-db.ddl;').

After the database is set up, copy the agilefant.war file to *webapps/* under the Tomcat's installation directory. Restart the Tomcat service. If you changed nothing in the Tomcat's configuration, you should be able to use the Agilefant. Just open http://localhost:8080/agilefant in your browser.

## Documentation

Agilefant has little documentation, but the user interface is designed to be easy to use. The best place to find more information on how Agilefant works or solutions to problems is the forum at http://www.agilefant.org/wiki/display/FORUM/Home.

## Configuration

All options affecting the runtime of Agilefant can be changed in the Settings tab of Agilefant.

## Introduction

Agilefant is an open source backlog management tool enhanced with hierarchical requirements handling capabilities. Agilefant provides three different planning levels for backlog management: product, project and iteration backlogs. Besides traditional flat-list backlogs, requirements can be expressed in tree form, in order to maintain traceability and transparency. In addition to backlog and requirement management, Agilefant contains different composition views to the items in different backlogs, which can be useful for small to mid-sized development organizations. These composition views include time tracking features, personal work management and portfolio handling.

In the following chapters we start from an idea and show how it can be represented, refined, implemented, and monitored using the Agilefant.

**Refining an Idea**

Say an eCommerce online store application has been developed for some time now. Multiple versions have already been released and the development of another release is about to begin. One of the sales team members has had an idea of developing a feature, which displays what other people have bought in the web store. He has communicated his idea to one of the Product Owners (PO) in the development organization. By Product Owner we mean a person who acts as a link between the business people and development organization and is responsible for the product's requirements.

The PO immediately creates a story called "suggestions what others have bought" to Agilefant's product backlog. The product backlog represents all ideas, requests, features, user stories and other desires placed on the product, while a story is an item representing one of these desires.

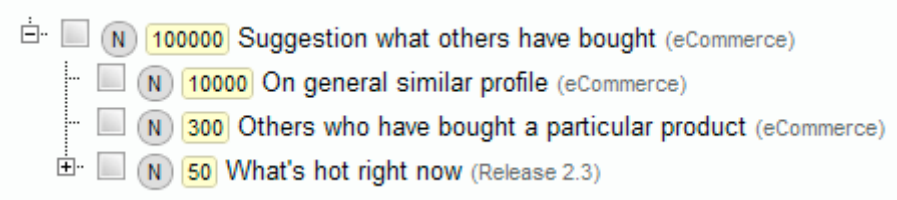



Figure 1. A view to the story tree

With a technical team, the PO investigates the new story; these sessions are often called 5%-workshops or story times. During the session they decide that the idea is too vague to be implemented as such, and enter a story point estimate of e.g. 100 000 points. *Story points* are used to estimate the relative size of a story, meaning that a story with 4 story points is twice as big as story with 2 story points. In this case, 100 000 story points denotes that the story cannot be currently estimated, or is an epic as some might say.

During the same session the PO and technical team agree that the original idea contains at least 3 different features. They create three child stories to the original story in the product backlog. The child stories are "suggestions on general from people with a similar profile", "suggestions from others who have bought a particular product" and "what's hot right now". Technical team estimates these stories and the PO places them in relative priority order. This is done by using the Agilefant's product backlog, which is actually a story tree. In the story tree, child and sibling stories can be created to a given story, and name, description, story point estimate, assignees, state, labels, and backlog of a story can be updated as well. In addition, the stories in same branch can be prioritized.

One of the stories, "what's hot right now", is estimated to be 50 story points in size. With the help of metrics from previous releases we can see that this seems to be in the scope of the upcoming release. Thus the PO assigns the story to the upcoming release project.

The release, or project, view also contains a story tree, which is a sub-set of the product backlog, consisting of the stories, which have been assigned to the given release. During other sessions, the PO and the technical team continue to refine the "what's hot right now" story. From previous iterations in Agilefant, they can see that the team can complete around 15 story points per iteration. Thus, the story is split to 10 child stories, which are estimated to be less

than 10 story points each. These 10 stories are small enough to be implemented in iterations, and don't need to be split to finer pieces. In Agilefant these stories are called leaf stories. *Leaf stories* are stories with no child stories. They are the leaves of a tree, whereas their parent stories are the branches.

In addition to story trees, the leaf stories are shown in the project leaf story list, which is a flat list, or a traditional backlog as some might say. This list shows all leaf stories in a given project and the leaf stories, which reside in the iterations of the given project. The leaf story list allows prioritizing the stories, which will be worked on by the development team. Moreover, the leaf story list provides a convenient way to assign stories to different iterations.

**From an Idea to Action**

The Product Owner has selected three of previously mentioned leaf stories to an iteration and is currently doing sprint planning with one of the development teams. During the sprint planning the development team members create tasks to elaborate the implementation details for the stories. *Tasks* describe how a particular story will be implemented, whereas a story describes what will be implemented.

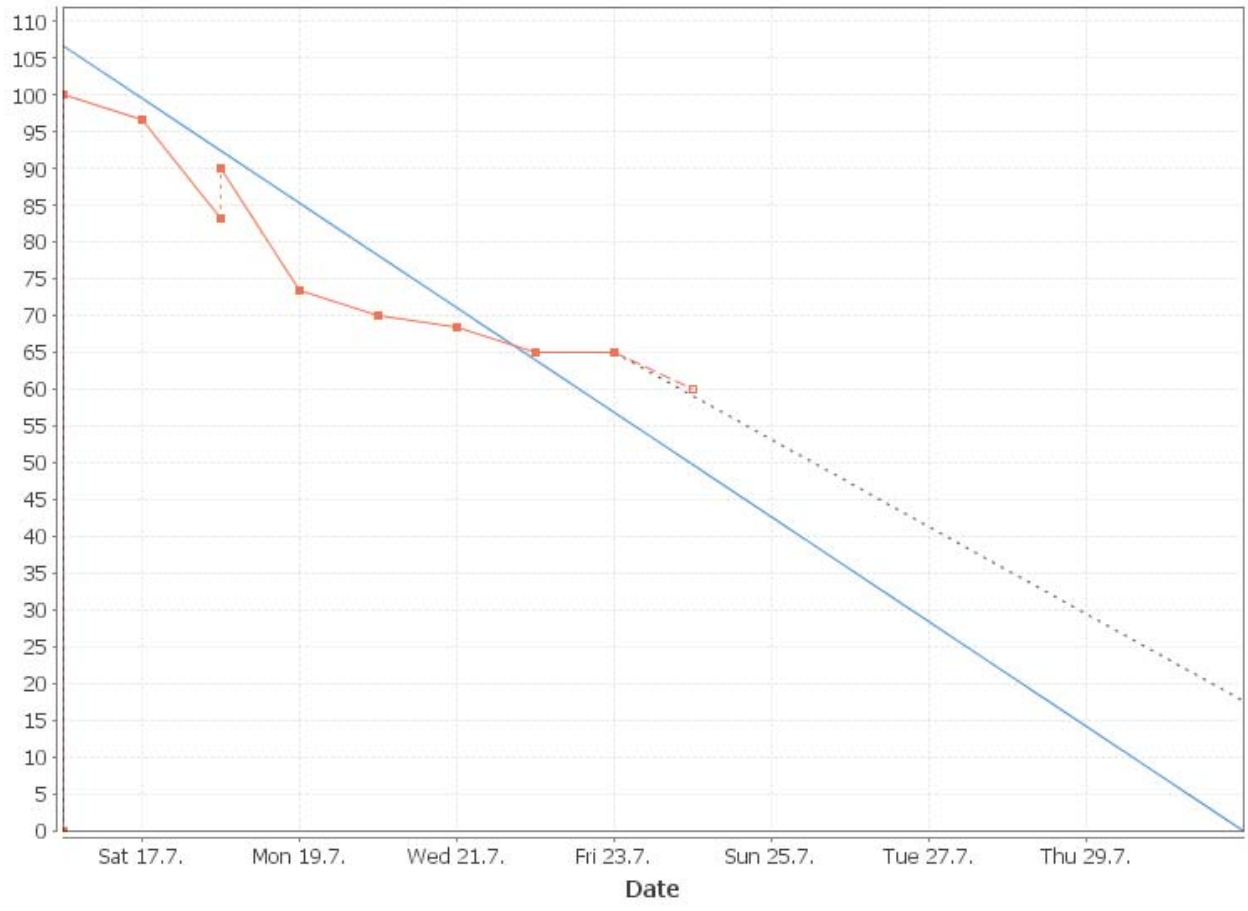



Figure 2. Iteration burn-down

Once the team has finished breaking down the stories, they estimate the time each task will take to implement. The estimation provides a rough picture of how long the implementation will take, and the PO can further adjust the iteration backlog's priorities based on that information. In Agilefant this initial estimate for a task is called *original* estimate.

Once the iteration begins, the development team members assign tasks and stories to themselves to indicate who's working on what. To further communicate the progress of the work, a state can be selected for tasks and stories. Available states are: not started, started, pending, blocked, ready, and done. While working on the tasks, the team members update the tasks' effort left estimates according to their best knowledge. *Effort left* marks the amount of time that the team members think they need to accomplish the task. This estimate may very well be higher than the original estimate as the developers become more informed of the situation once they have actually started working on the tasks.

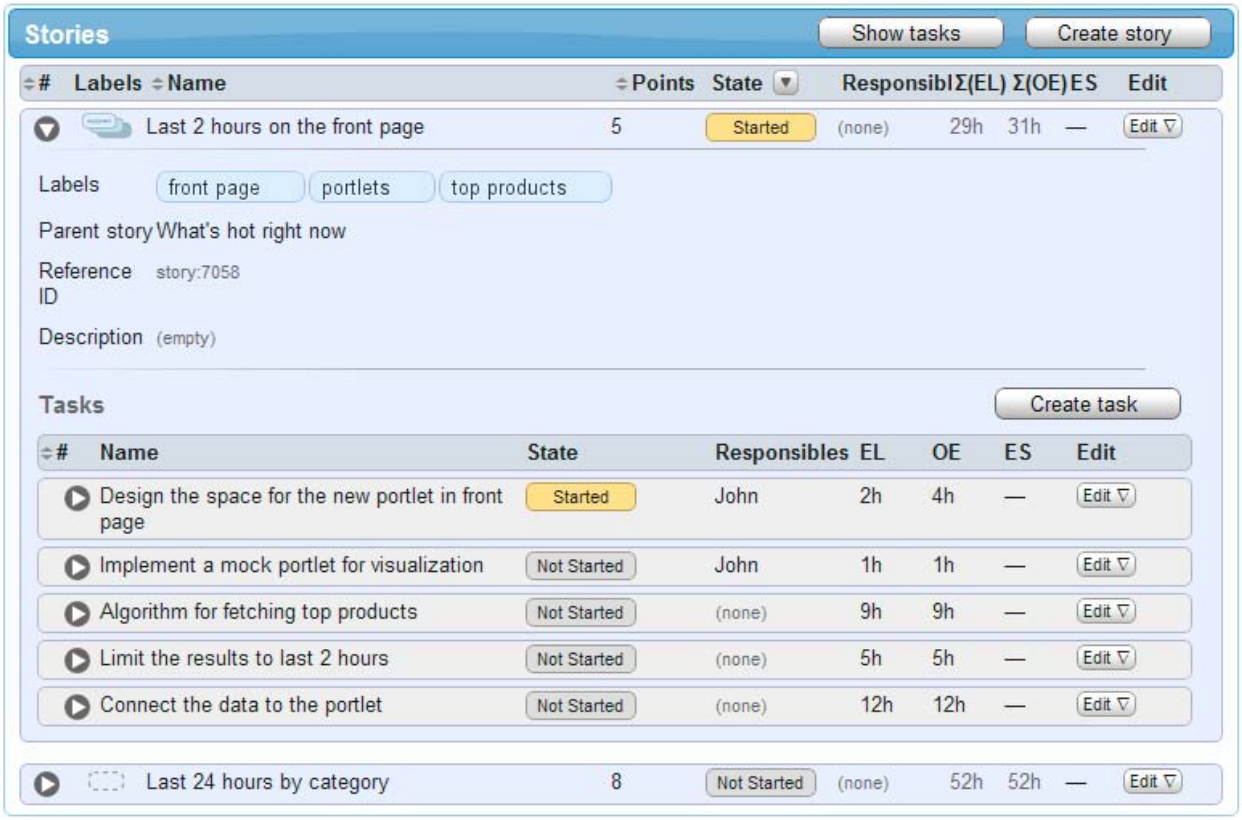



Figure 3. The iteration backlog

An important element of the iteration view is the iteration burndown. It is a graphical representation of task progress in the iteration over time. It shows a linear reference based on the original estimates. The burndown provides a convenient tool for tracking iteration progress both for the development team and for external stakeholders.

**Tracking the Progress**

Once the iteration has ended and iteration demo has been held, the PO reviews the state of higher level stories. Even if all child stories of a story are marked done, the parent story might not be complete. Often stories are split only partially and refining the story is continued when needed. Thus, manual inspection of the stories is required.

Project progress can be tracked from project burnup chart. *Project burnup* is a time series displaying planned work versus completed work. Planned work is the story point sum of all leaf stories that reside in the project backlog. Similarly, the work done is the sum of all leaf stories marked as done. As stated earlier, stories assigned to iterations are also shown in the project backlog and thus, when a story is completed in an iteration, the progress is shown immediately in the project burnup.

In addition to project burnup, Agilefant provides the possibility to track story level progress. For example, the sales person, who originated the idea “suggestions what others have bought”, might wish to follow its progress. Agilefant calculates two different progress metrics for each story. First, the *simple metric* shows total leaf story points and total *done* leaf story points in that story. Second, the *advanced metric* takes into account the branches under the story in question.

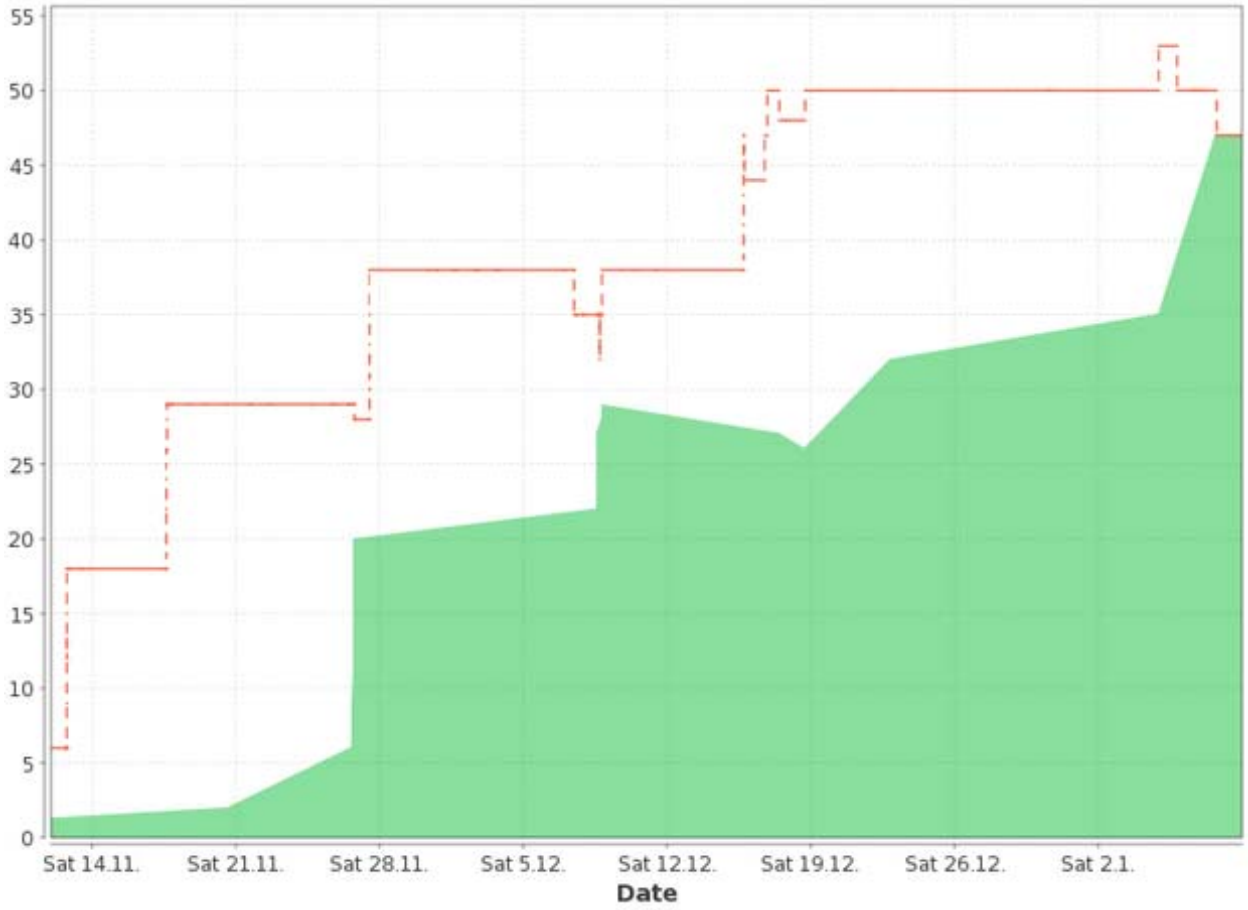



Figure 4. The project burnup

**Summary of the Concepts and Basic Views**

As described earlier, Agilefant has a product oriented approach. Projects reside in products, and iterations in projects. Projects and iterations have a fixed time span and users can be assigned to them. Stories describe the work to be done, or on the other hand the features to be implemented. Their size can vary from simple needs to a multi-feature bulk. Stories that are selected to be implemented are moved to iteration level, where they can be elaborated as tasks, which depict the actual work that needs to be done (e.g. design the test). Tasks are estimated in hours, which are easy to grasp for developers. Stories are given ballpark estimates in story points, which describe the relative size of the stories.

Product backlog is actually a tree consisting of the product’s requirements. Projects have both, a story tree and a flat list of the project’s leaf stories. Iteration level contains only a list of leaf stories.

## Using Agilefant in a Project Organization

The previous narrative was written from a product organization's point of view. Although the concepts and backlogs' names better suit the needs of a product organization, Agilefant is not restricted to those. Project organizations can benefit from Agilefant, too, by using it in a slightly different manner. Products can be used to represent a specific customer, whereas projects can be used as releases or milestones. The iterations may or may not suit project organizations, depending on the amount of projects and teams. Since there is no "team iteration" in Agilefant currently, a new one must be created for every project.

## Other Functionalities in Agilefant

In addition to the aforementioned functionalities, Agilefant has also a couple of other useful features. For organizations that need time tracking, there is a *timesheets* feature allowing you to log hours to products, projects, iterations, stories and tasks, and generate reports on the effort spent. You can also export the reports to an Excel sheet.

The *Daily Work* page is an aggregate page, where everything that a single user is currently working on is gathered to one place. It shows the assigned stories and tasks from the currently ongoing iterations. Users can easily keep their own work queue of tasks to keep in mind, what's currently important. Portfolio view is also an aggregate page. *Project portfolio* is a special page gathering the ongoing and future projects to a list, where you can track them. You can create other portfolios with customizable widgets to suit your needs. Portfolios can be public or private.

## Conclusion

Different stakeholders in a software development organization have various information desires. Some wish to view the high level picture, while some are only interested in individual iterations or stories. Agilefant satisfies these various information interests by offering different planning perspectives, requirement abstraction levels and composition views.

Iteration backlogs represent actual work done by the development teams. Projects backlogs are a composition of work assigned to different iterations, and work waiting for further refinery, or to be assigned to an iteration. Product backlog contains all needs, ideas, goals, and requirements that have been expressed by different stakeholders towards the product in question. Together, these three types of backlog form nested planning horizons.

The hierarchical stories provide a traceable path through the different planning horizons. With the story tree a developer can connect his work to a larger scope and an external client can see what is currently happening in his high-level feature.

Different backlogs and possibility for hierarchical stories form Agilefant's core. This core is supported by several radiator views. The Daily Work view displays what everyone is currently doing and how occupied they are with those activities. The Timesheets view provides possibility to log and track time spent. The Portfolio view combines information from different products and thus offers an organization-wide perspective.

# SoapUI

Axel Irriger, axel.irriger [at] gmail [dot] com
cirquent GmbH, http://www.cirquent.de

SoapUI is an application and framework to simplify the testing of web applications and web services. Test cases can be entered using a graphical user interface. They can be executed either by using the graphical user interface or in an embedded fashion with either Apache Ant or Apache Maven.

**Web Site:** http://www.soapui.org and http://www.eviware.com/ (http://sourceforge.net/projects/soapui/, respectively)
**Version Tested:** soapUI 3.5 on Windows XP with Java 1.6.0_20
**License & Pricing:** Open Source and Commercial License available
**Support:** Forums and web site, various resources on the net

## Installation

The soapUI distribution can be downloaded from http://www.soapui.org. The distribution is a compressed archive, which contains an automatic installer. Binary distributions exist for Microsoft Windows, Linux, MacOS and Solaris.

After installation, it can be started and presents itself with the graphical user interface (see Figure 1).

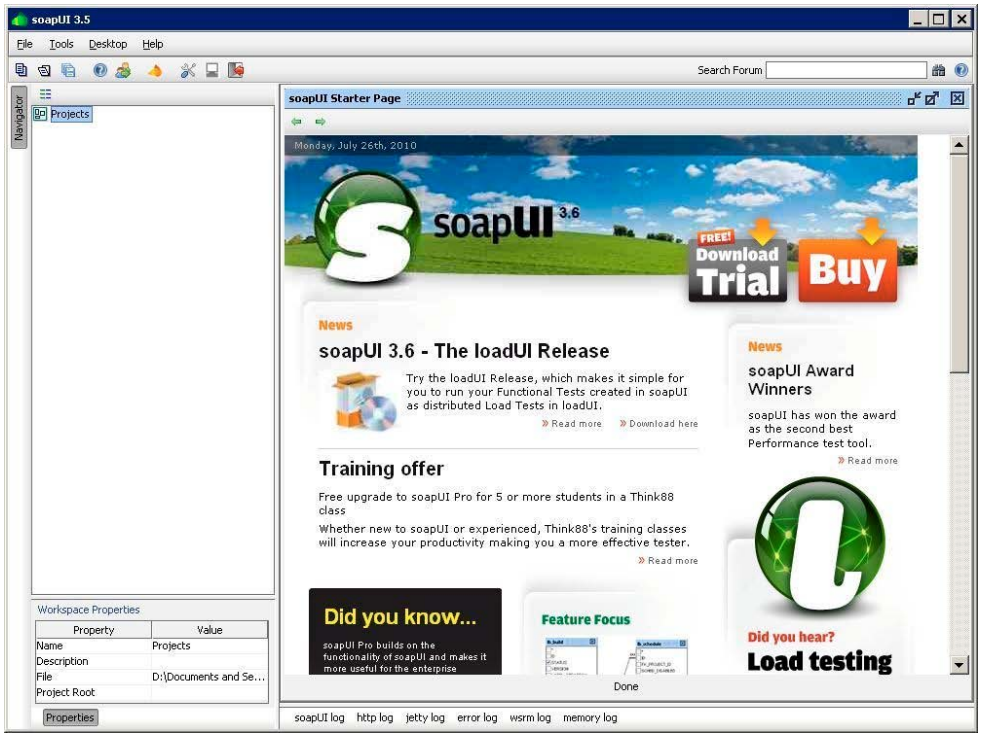



Figure 1 soapUI 3.5 Graphical User Interface

## Configuration

The work with soapUI is organized around a project (see Figure 2). As you can see from the dialog, you have the option to directly add a web service description (WSDL) file, if you are to test a web service. Nevertheless this is not necessary, especially if you test a web application.

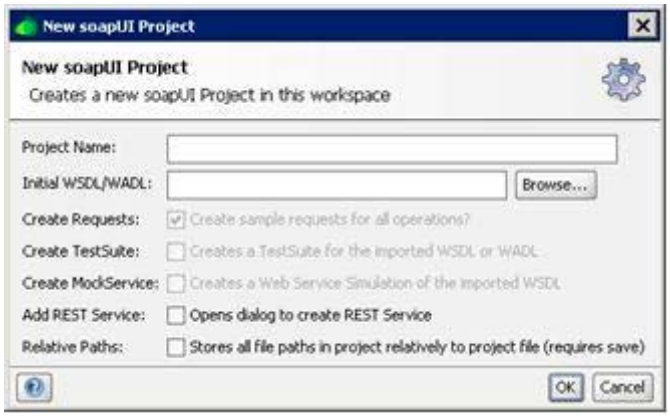



Figure 2 New Project Dialog

If you enter a WSDL, you can instantly create sample requests from the defined XML schema, a test suite for all the operations in the service and a mock, which simulates the real service.

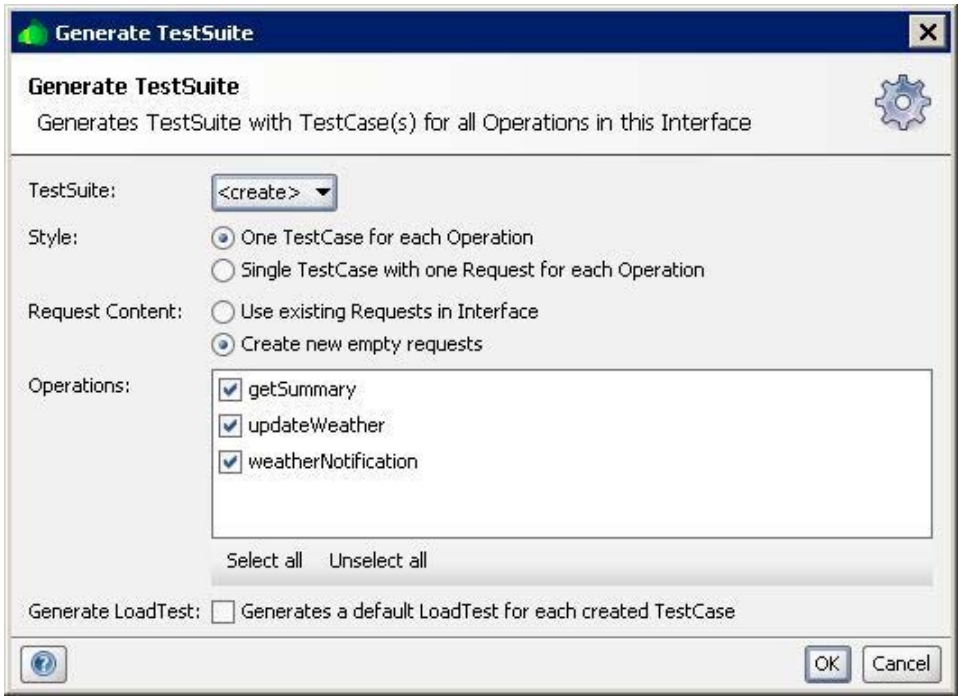



Figure 3 New TestSuite dialog

Within a project, you can have several test suites, which are a logical group of similar test cases (see Figure 3). The definition of "logical group" is nonetheless up to you. In some cases, it's enough to have one test suite for a whole application; in other cases it might be necessary to define a test suite per application use case. However you define your structure, the surrounding container always is the project. Therefore, it's a good idea to store the soapUI project definition alongside the application or service under test.

Within each test suite are one or more test cases. Each test case is to test exactly one functional artifact.

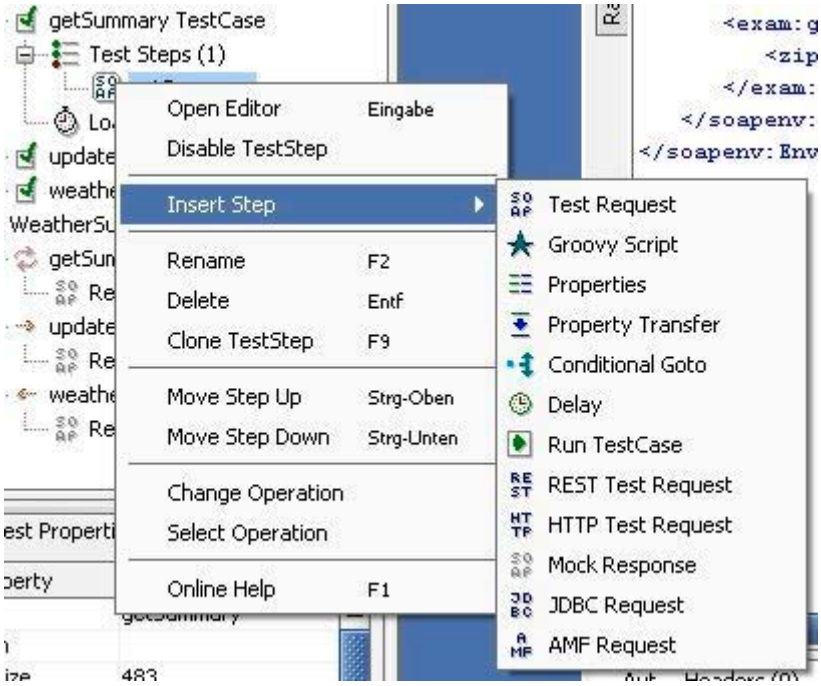



Figure 4 Supported test steps

A test case is a sequence of steps, being carried out by soapUI (see Figure 4). Since functional testing and load testing are two separate things, you can either add load test steps or functional test steps. While being possible, you are not advised to mix both, since it complicates readability.

Each step you define is mapped to an action, which is carried out by soapUI. Such an action can be the invocation of a HTTP URL, to load a web page, or the transfer of portions of one response document to a new request (for instance, to carry along a session ID).

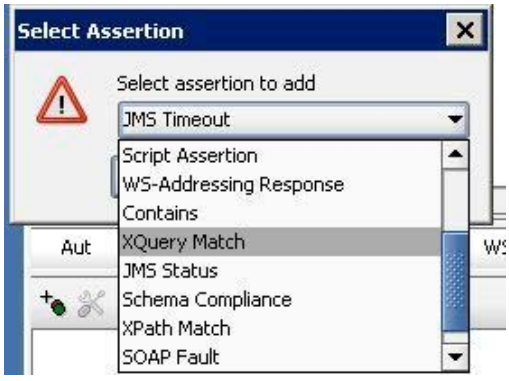



Figure 5 Some supported assertions

For each test step, it is possible to add assertions. An assertion is a check of an expected value against a found one. That is, after logging in to a web site, there is often some sort of greeting for the user. If you know (which you do), with which user you are logging in, you can check whether this particular greeting is present.

Besides such pure functional assertions, non-functional assertions are possible in terms of page load duration.

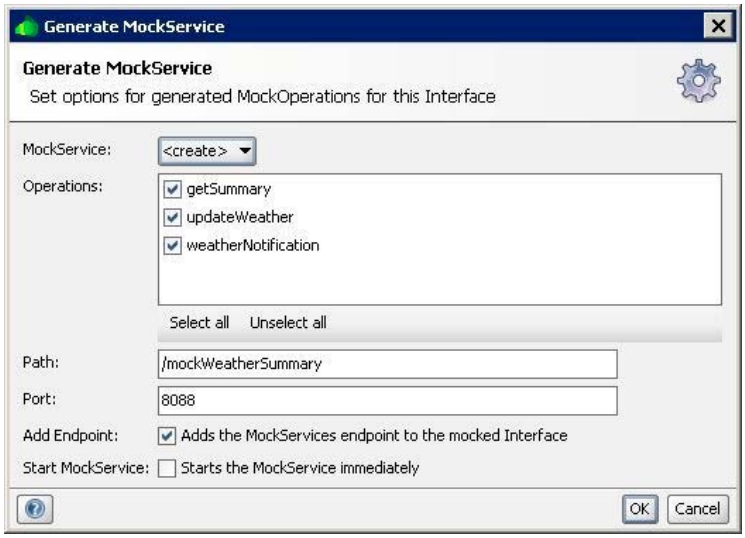



Figure 6 New MockService dialog

Since it is not always the case that you have direct access to the web service you are testing, soapUI supports using a mock, which can be automatically created, when creating a new project (see Figure 6). This mock offers all operations but can be populated with pre-defined responses for requests and simulates the web service behavior.

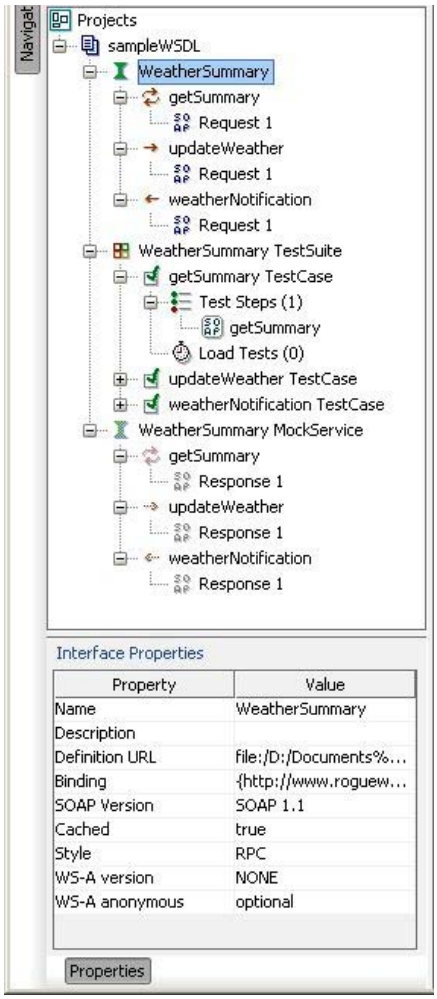



Figure 7 Project Overview

When your test case and test suite are populated, it is also possible to save this configuration set to a XML file. That way, not only regression testing is possible but also test automation. When your project definition is done, you can see everything in the project overview.

From here, you can see under the first section “WeatherSummary” automatically generated requests from the WSDL schema definition. The section “WeatherSummary TestSuite” is an automatically generated test suite, which contains tests for each operation of the service. The third section (“WeatherService MockService”) contains the mock service, which is created for all operations. For each operation you can define one or more responses, which are sent to the client in case of XPath assertions or a simple invocation counter.

**Automating SoapUI**

soapUI comes with a graphical user interface, which is great for daily usage and the definition of test suites. To actually execute tests, it is possible to either use the GUI used in test suite creation or to embed the test execution with Apache Ant or Apache Maven. This way, the test suite can be stored alongside the application in the version control repository and reused in a continuous integration environment,

**Plugins for SoapUI**

The soapUI approach is to provide not only a test application but a test framework. This framework can be extended using the provided API. Within the core web-enabled stack, loadUI is quite helpful, as it supports load testing a web site. There is also a plugin for testing REST (REpresentational State Transfer) web services. These are, in contrast to SOAP web services, centered around the URL, which describes the objects to operate on and the action to invoke and not the body document.

Web services are often not only used via HTTP, but also via JMS (Java Messaging System). For that transport mechanism, integration with HermesJMS is available. Taking a step back from the web space, soapUI also delivery various plugins for other test purposes, such as JDBC (Java DataBase Connectivity). They use the graphical user interface of soapUI to simplify test creation.

**Documentation and Literature**

The quite extensive documentation for soapUI is available on its web site, http://www.soapui.org. Also, eviware as the primary developer of soapUI, supports with community and “pro” forums.

**Summary and Conclusion**

The testing space is always in the move and soapUI is no exception. Coming from the testing of web applications and web services, the framework now supports a lot more. When focusing on web applications, it provides all relevant operations needed to replay whole web site conversations. The web service stack is well supported with SOAP and REST based services, the ability to invoke XML schema validations for SOAP web services and XPath queries on any XML document. Besides that, other operations, such as JDBC, can be integrated in a test as well. This flexibility, combined with a simple and functional graphical user interface, makes soapUI a highly recommendable test tool, which should also be considered for integration with a test automation framework, such as Hudson CI.

**Conquer Complexity in Embedded Software Engineering** White Paper: Bridge the Gap Between Modeling & Development. With today's complex products, poor management of model-based development and testing can lead to unnecessary rework, wasted time and risk to the business. Learn how to address these challenges and how to solve the complexity of modern software engineering. Discover the solution needed to harmonize simulation and modeling with the rest of the engineering lifecycle (and why that matters).

http://www.mks.com/mt-bridge-the-gap

---

**Tools for Agile Software Development**. This website presents a list of open source and commercial agile software development tools used for:

* Scrum agile project management and lean Kanban
* BDD Behavior Driven Development
* Continuous integration
* Code and database refactoring

http://www.agilesoftwaretools.com/

---

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

---

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