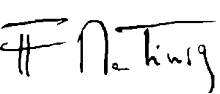
METHODS & TOOLS

Practical knowledge for the software developer, tester and project manager **ISSN 1661-402X**
Summer 2010 (Volume 19 - number 1) www.methodsandtools.com

Are Software Developers Worth More than Accountants?

Methods & Tools is located in Switzerland, a country famous for its chocolate, watches... and banks. I was therefore participating to a banking IT conference last month. The CIO of a very large private banks revealed that 15% of employees of his company were working for the IT department. He described them as the "mechanics" supporting all the business. And I thought this was nice. Then a CEO of a retail bank came to present the results of a survey of Swiss bank top managers. They were asked what would make their bank different or better than their competitors. None of the answers mentioned IT. And I thought this was not nice. Thus came the question: do software developers make more difference than accountants for their CEOs? Are they just something you need to have, but don't really contribute to the performance of the organization? The software development function is separated from the other operational activities and developers are considered as a support function as accounting could be. This comparison is not so awkward, as accountants belong also mostly to the introverted psychological category, like developers. And accountants could also deliver information that could change the way a company works, finding for instance which products or customers are truly profitable. In a book from Watts Humphrey, he quotes Dick Garwin, the designer of the hydrogen bomb, saying: "You can get credit for something or get it done, but not both." Software developers may belong more to the second part of this alternative. Despite all problems that impact our projects, we deliver solutions that allows organizations do perform better, but we don't get all the recognition that we deserve. Some might be happy with our current lack of visibility, but then we should not complain if we are often considered only as a cost variable that should be minimized and not elements that could increase revenues. To achieve this objective, it is important that developers get closer to their users and improve their knowledge of business. Users don't want a cool Ruby on Rails Ajax apps, they want solutions for their problems. If we want more consideration from the management, we have to express more our positive impact in organizations. After all, even some accountants managed to do this.



Inside

Aspects of Kanban .... page 3
Test Language -Introduction to Keyword Driven Testing .... page 15
A High Volume Software Product Line .... page 22
Better requirements definition management is better for business .... page 31
The Core Protocols, an Experience Report .... page 39
Tool: eValid .... page 52
Tool: Hudson .... page 57
Tool: FitNesse .... page 60
Tool: VoodooMock: Mock Objects Framework for C++ .... page 65
Conference: Jazoon .... page 74

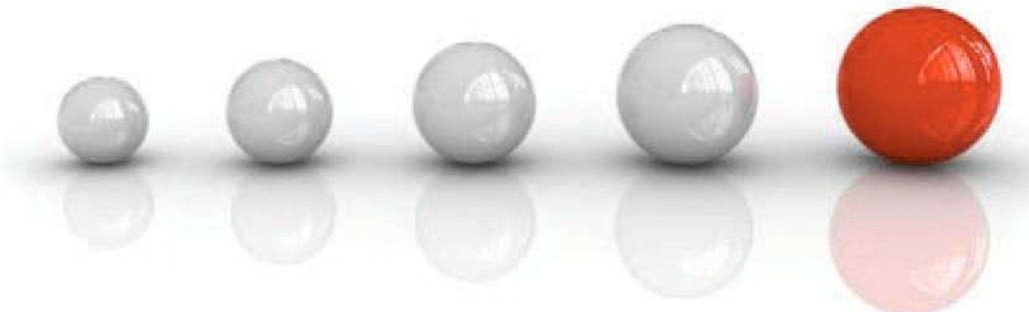
MKS Intelligent ALM - Click on ad to reach advertiser web site

Looking for measurable value from
Application Lifecycle Management?

• A major autmotive supplier cut a three year development cycle by a third.

• A utilities company accelerated release management processes by 80%.

• A global geo-content provider increased its On-Time measure from 70% to 93%.



You can achieve these kinds of results too.

When software is critical,
you need a smarter approach.

You need Intelligent ALM™

1 800 613 7535 | On-demand Webinar:
www.mks.com/alm-webinar

Aspects of Kanban

Karl Scotland, http://availagility.co.uk

Introduction

The Kanban software development community can be traced back to Agile2007 in Washington DC. At that conference a number of people were talking about their different approaches to development that they were using. Chris Matts was talking about Real Options and Feature Injection, Arlo Belshee was talking about Naked Planning, and David Anderson was talking about Kanban. All three had some similarities, which inspired a group of people to go away and experiment themselves and share their experiences. The name which emerged as an identity for the group was "Kanban".

Kanban is the Japanese word for visual card and can have a number of interpretations with respect to software development. Firstly, it could be used to refer to the index card commonly used by Agile teams. Secondly, it could be used to refer to an Agile team's task board, or story board. Finally, it could be used to refer to the whole system within which an Agile team works.

In his book "Toyota Production System" [1], Taiichi Ohno says, "The two pillars of the Toyota production system are just-in-time and automation with a human touch, or autonomation. The tool used to operate the system is kanban." With this perspective, a Kanban System for Software Development refers to the whole system, and not simply the tool or the board. The community chose to name the systemic approach after the tool that inspired much of the thinking.

Viewing Kanban as a systemic approach leads to Systems Thinking. John Seddon, in his Vanguard Method [2], says that management thinking defines the system which defines performance. In order to improve the system, we should first understand the purpose of the system, create measures which help to determine whether the system is meeting that purpose, and then put a method in place to enable the system to meet that purpose.

Kanban is a way of creating a method and generating metrics, in order to improve capability to meet a purpose. The remainder of this article discusses five aspects of a Kanban System; workflow, visualisation, work in process, cadence and continuous improvement. These aspects are not practices to be followed, but key areas to consider when thinking about the method used to change and improve an organisations delivery capability.

Workflow

Workflow is the understanding of how business or customer value travels through the Kanban System. The Agile community recognised that software development is a knowledge creation activity which includes randomness and variation and the "Inspect and Adapt" mantra is the response which makes the impact visible such that the feedback can be used to learn and respond accordingly. We can take this further by understanding the mathematics and science behind the randomness and variation and exploiting this to our advantage.

Recognising the workflow through an activity such as Value Stream Mapping can give us this additional transparency which we can use to influence our process. Value Stream Mapping is so called because its focus is on understanding how units of value flow through a system. For software development, this can be generalised into how units of value *expand* into smaller units of work, which then *collapse* back to deliver the original value.

For example, a specific Benefit to a customer may expand into a number of Features, which may expand into various User Stories, which may then each expand into Tasks. The Tasks subsequently collapse back together to realise the User Stories, which realise the Features, which ultimately realise the Benefit.

Understanding the workflow thus consists of knowing what we consider to be of value and what expand and collapse points there are to deliver that value through the system. Making the expand/collapse patterns explicit helps us to deliver the value more effectively across the whole value stream.

It could be argued that the value often goes through a typical "waterfall" workflow. This usually has baggage associated with it, however, so instead we can generalise the following workflow:

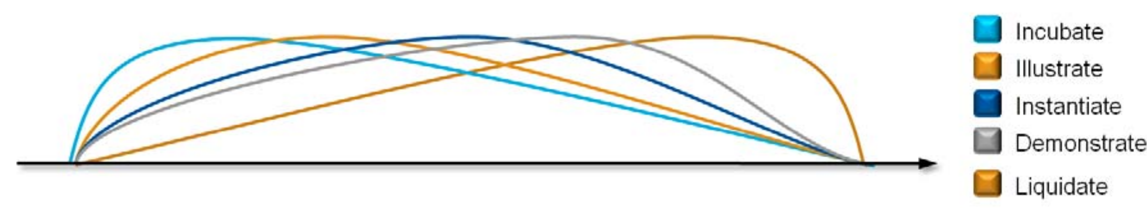
Incubate > Illustrate > Instantiate > Demonstrate > Liquidate

Incubation is when something is still an idea. It may grow and evolve on its own, but without significant investment. Illustration is when the idea starts to take shape into something which can be described more concretely, typically with user stories and examples (or tests). Instantiation is when the idea is built and tested. Demonstration is when it is completed and ready to be accepted by the business. Liquidation is when the idea is released and realising its value.

Another way of thinking about discovering a workflow is to view it as process archaeology. A process often has many layers, and by digging through those layers we can surface what is really going on. This will typically involve talking to the team members about how they really work, and it will often result in something other than what was expected, as problems that were previously hidden are surfaced.

Common items to look for in a workflow include queues and batches and failure demand. Queues and batches are points in the workflow where the work is being processed. Queues are where work is building up because there is not enough capacity to process it and batches are where work is being held to be handed over and processed in a large volume. Failure demand is where is work the result of not doing some, or not doing something right. Rather than optimising a value stream for failure demand, the failure demand itself should be avoided.

It is important to remember that workflow stages are not equivalent to people or roles, and that having transparency of stages in a workflow does not imply having silos or specialisation. Instead, by focussing on letting the work flow across the stages, we can move towards a one piece flow, where a multi-skilled, cross functional team can work as a single cell to progress the value through the system. A generalised-specialist approach means that team members can both focus on one particular stage, while still being involved across the whole process, in the same way the "Type C" development is described in the "New New Product Development Game" [3].



Seapine Agile Expedition - Click on ad to reach advertiser web site

Visualisation

Visualisation is the means by which we can understand the work and the workflow by using a kanban board to create a powerful visual management tool that shares a mental model which is visual, interactive and persistent.

In a recent TED Talk [4], Tom Wujec explains how this works when he talks about three ways that the brain creates meaning. Firstly, visualisation creates a mental model because of the way that different areas of the brain process different visual inputs such as shape, size, and location. Secondly, interaction enriches the mental model further through engagement. Finally, persistence allows the mental model to be part of an augmented memory which can evolve over time.

This leads to the idea of boundary objects. Brian Marick wrote an introductory paper [5] in which he talks about communities and practice and interest. A community of practice is formed around a work discipline, while a community of interest is formed around a common problem or concern. Communities of interest are made up of members of different communities of practice. A boundary object provides a means for communities of interest to communicate across their different practices.

Marick lists several properties of a boundary object which can be useful to bear in mind when building a kanban board; it should be a common point of reference for the community of interest, represent different meaning to different members of the community and help translation between the meanings, support coordination and alignment of the work within the community, be a plastic working agreement which evolves as the community learns, and address different concerns of the community members simultaneously.

A kanban board can be considered to be a boundary object when it is a social artefact which is a focal point for a team. By visualising a team's work, it becomes a common point of reference. The representation of the work, and its status, enables communication and coordination between all team members, as well as visualising the workflow and policies that are the team's working agreements. The kanban board should be able to evolve with workflow and policies over time. Thus a kanban board represents the shared mental model which is created collectively and collaboratively, and helps clarify the meaning of what the board is representing.

Another relevant set of ideas to visual management are those raised by Dan Pink when he talks about the surprising science of motivation. In his book "Drive" [6], he says that rather than the carrot and stick approach of extrinsic motivation, a better approach is intrinsic motivation, which consists of three elements; autonomy, mastery and purpose. Autonomy, or the "desire to direct our own lives", is achieved when team members can see what needs doing, understand the working agreements, and choose themselves what they should do. Mastery or "the urge to get better and better at something that matters" is achieved through being able to interact with the kanban board to evolve and improve it. Purpose, or "the yearning to do what we do in the service of something larger than ourselves", is achieved when the persistence of the kanban board makes it clear what the value of the work is and why it is being done.

A kanban board is a visualisation multiple pieces of data. In his classic book "The Visual Display of Quantitative Information" [7], Edward Tufte introduces a set of principles for the effective display of data and it is insightful to review some of these ideas.

Tufte talks about a number of different types of graphical designs. Time series is probably the most common, where time is along the horizontal axis, and another data type along the vertical. This is probably the least relevant design, because a kanban board is typically a snapshot of the current status. Similarly, a space time narrative, which tells a story in a spatial dimension over time, may not be the most obvious choice. It does raise the question of visualising the narrative of the work over time though, which could be interesting.

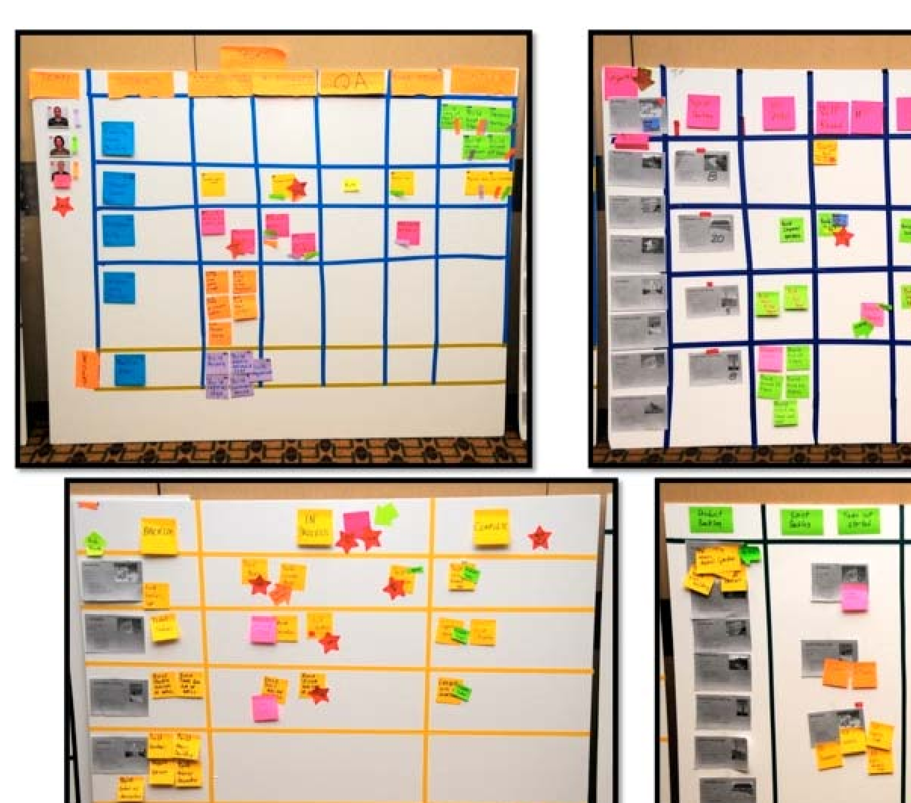
Maps also introduce some different ideas. What would a kanban board look like it showed the terrain of a project and where each piece of work was on that terrain? The most common form kanban board is probably a relational one, where the two axes show different types of information, such as scope and status.

Most of Tufte's book is spent discussing ways of improving the way that data is presented; specifically, maximising data ink, reducing chart junk, and improving data density. Data ink is the ink that actually represents data. While kanban boards generally use more than just ink, the principle holds true for making sure that as far as possible, anything on the board should hold information. The corollary to this is that anything which isn't data ink is chart junk. Grids, redundant data, or decorations and embellishments for aesthetics may create noise which masks the real story. Finally, data density is the amount of data within the given space. The eye can take in a high precision of detail, so by maximising the data ink and being clever with multi-functioning graphical elements, it is possible to visualise many dimensions in a small space.

A kanban board is what Tufte would call a multi-variant display, with the variants typically being the usual project management details, but also including the concerns of any member of a board's community of interest. As a starter, there are the popular "iron triangle" variants of scope, time, resource and quality. Other common variants are things like priority, status, issues, risks, constraints, dependencies and assumptions. More recently, teams have been talking in terms of variants such as capacity and demand, not to mention value and other economic aspects.

To visualise all these variants we can use a number of techniques. Properties such as size, colour, format, location and alignment can all create multi-functioning graphical elements to achieve a high data density, while for a physical board, material and texture can add further depth. The following examples are from a Visual Management workshop run by Xavier Quesada Allue [8] at Agile2009. They show different solutions to the same problem, using a variety of styles, techniques and materials.

GreenHopper for Kanban - Click on ad to reach advertiser web site



Work In Process

Work in Process (WIP), and the way it is limited, is the means by which we can create a pull system which balances capacity and demand through the value stream.

In a pull system, work is processed through being signalled, rather than being scheduled. This is what avoids a build up of inventory, and enables work to flow through the system as capacity allows. Satoshi Kuroiwa, a former Toyota manager, used the analogy of a chain of paperclips in a talk at Agile Japan in 2009. Pushing the paperclips will inevitably cause them to pile up, whereas pulling them will result in them moving smoothly.

Applying this to a software development workflow means that upstream work can be made available, but it is the team members' responsibility to decide when they are able to take it. The act of taking, or pulling, the work, is a signal for the more upstream work to be processed. However, when work is available but not being pulled, then production upstream will gradually throttle down to avoid any pile-up. With a push system on the other hand, work will be scheduled and handed downstream regardless of whether there is capacity to process it or not.

Work in Process (WIP) has an impact on productivity, inventory and teamwork, and by being aware of WIP, and reducing and limiting it, we can improve Kanban System.

Productivity can be measured in terms of cycle-time and throughput of valuable units of work. Cycle time is the length of time to complete a process and throughput is the amount of output from a process in a given period of time. Cycle time and throughput are both improved by decreasing WIP. A simple example of this effect is CPU load, where application performance goes down as CPU load increases.

The effect can be explained by looking at Little's Law for Queuing Theory:

Cycle Time = Number of Things in Process / Average Completion Rate

Little's Law tells us that to improve cycle time, there are two options; reduce the number of things in process or improve the average completion rate. Of the two, reducing the number of things in process is the easier, and once that is under control, then the more challenging changes to improve completion rate can be applied.

A further understanding can come from Traffic Flow Theory:

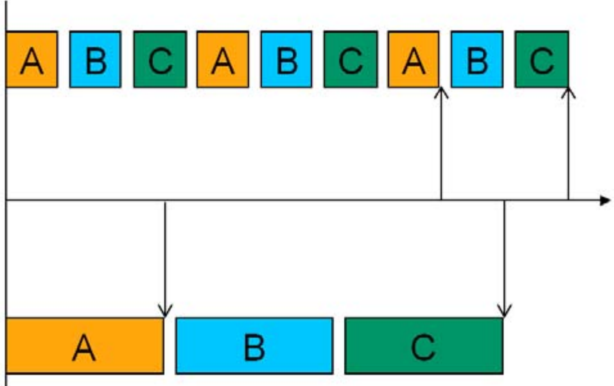
Flow = Speed * Density

Traffic jams occur as traffic density increases, and traffic speed decreases. However, when traffic density decreases, speed only increases to a point (which should be the speed limit). As a result there is a point at which decreasing WIP below a certain density will reduce throughput.

Another factor in improving cycle time and throughput is that of multitasking. Reducing multitasking is beneficial for two primary reasons.

Time is lost to context switching per task, so fewer tasks means less time lost. Gerald Weinberg, in his book "Quality Software Management: Systems Thinking" [9] suggests that 20% time is lost per additional task. Thus 1 task can consume 100% of time available, 2 tasks will consume 40% of time available each with 20% lost to context switching, 3 tasks will consume 20% of time available each with 40% lost to context switching etc.

Performing tasks sequentially yields results sooner. As the diagram below shows, multi-tasking A, B and C (on the top), delivers A much later, and even C slightly later, than sequentially (on the bottom).



Dr. Eliyahu Goldratt introduced the idea of throughput accounting in his business novel The Goal [10] Throughput accounting suggests that the business goal is to make a profit, and that this is determined by work in process, operating expense and throughput. Profit is increased by decreasing work in process, decreasing operating expense and increasing throughput.

Any features we have developed, but not yet released, can be considered inventory. Therefore, as well as helping to improve cycle time and increase throughput, limiting work in process also helps to increase profit by reducing inventory. In his keynote at Agile2009, Alistair Cockburn also introduced the idea that for software development, the unit of inventory is the unvalidated

decision [11]. By limiting WIP we are focussing on getting feedback on fewer decisions sooner. Finally, by having fewer work items in process, then the team is able to focus more on the larger goals, and less on individual tasks, thus encouraging a swarming effect, and enhancing teamwork. Limiting WIP like this can seem unusual for teams, and there is often a worry that team members will be idle because they having no work to do, but are unable to pull any new work. The following guidelines, in priority order, can be useful to help in this situation.

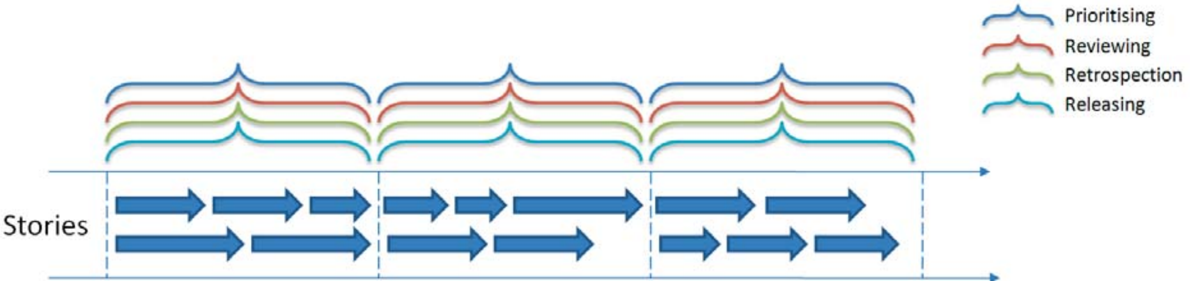
1. Work directly on existing work to progress it
2. Collaborate with team members on existing work to remove a bottleneck
3. Begin working on new work if capacity is available
4. Find some other useful work

When team members have to find some other useful work then "bubbles of slack" are formed around the work. This creates opportunities for improvement without needing to schedule them with techniques such as Gold Cards. This can be work which won't create any work downstream, but will improve future productivity and can be paused as soon as existing kanban slots become available. Investigative work such as technology spikes, refactoring or tool automation, and personal development or innovation work, are all activities which might help the team in the future.

Cadence

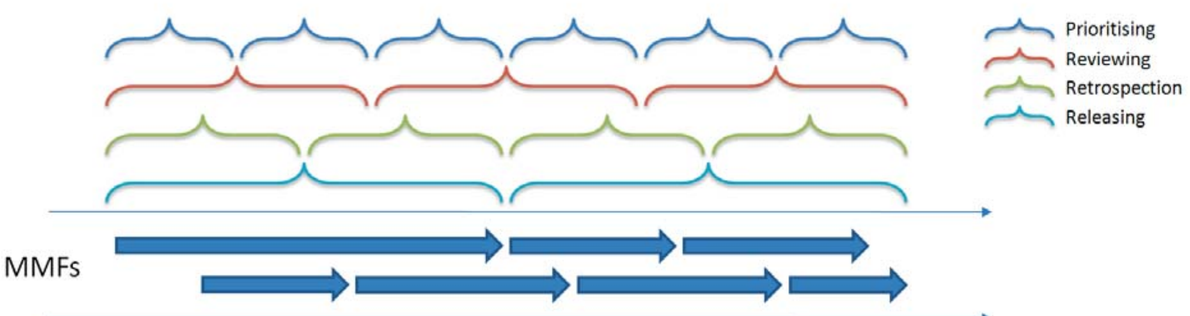
Cadence is the mechanism that teams use to establish a reliable and dependable capability. A consistent cadence demonstrates a predictable capacity and gives some confidence in coordinating the upcoming work when it is being triggered rather than scheduled.

Vanilla Agile time-boxing is one specialised form of cadence. It is a metronomic cadence with a single tick. All the main process events are based around this single tick which occurs on the time-box boundaries. In addition, the unit of work, commonly User Stories, should be small enough to be scheduled into the time-box, and subsequently completed in the same time-box. However, while User Stories in process can be limited within a time-box, they don't always fit into one exactly. Additionally, while releases can occur at the end of each time-box, User Stories are only *potentially shippable* product increments, but may not be *coherent* product increments.



The various events can be decoupled, however, such that they happen separately at different rhythms. This creates a polyrhythmic cadence, more like a Drum Circle, where each drum represents a different event. The rhythm is more complex than the single tick of a metronome, and can be more varied. Units of work can be larger Minimal Marketable Features (MMFs), which while needing to be as small as possible, are not constrained be being required to fit into a time-box. Instead, a MMF is able to flow over a number of process events while it delivers a releasable *coherent* product increment.

Prioritising, planning, reviewing, retrospection and releasing all still happen regularly, but because they are de-coupled, they can happen independently, at differing rates, which may provide more freedom in creating a natural process.



A cadence is usually 'harmonic', in that there is a neat overlap between the different rhythms, and generally keeps a regular 'time signature' to create consistency. However, it does not have to be, and a look at some definitions of cadence from dictionary.com can show why.

- In music, the ending of a phrase, perceived as a rhythmic or melodic articulation or a harmonic change or all of these; in a larger sense, a cadence may be a demarcation of a half-phrase, of a section of music, or of an entire movement
- Music. A progression of chords moving to a harmonic close, point of rest, or sense of resolution.
- The flow or rhythm of events, esp. the pattern in which something is experienced: the frenetic cadence of modern life.

Thus cadence is what gives a team a feeling of demarcation, progression, resolution or flow. It is a pattern which allows the team to know what they are doing and when it will be done. For very small or mature teams, this cadence could by complex, arrhythmic or syncopated. However, it is enough to allow a team to make reliable commitments because recognising their cadence allows them to understand their capability or capacity.

The appropriate cadence for a team will be influenced by their transaction and coordination costs. Transaction costs are those associated with performing an activity. For example, the cost of making a release is a transaction cost. Coordination costs are those associated with the logistics of an activity. For example, the cost of getting people together to manage a release is a coordination cost.

Thinking in terms of transaction and coordination costs can provide the basis for establishing an appropriate cadence for the various events such as prioritisation, planning, reviewing, retrospection and releasing. Focussing on reducing these costs can subsequently allow the cadence to change as delivery capability improves.

The end goal of reducing costs and improve cadence is to be able to quickly, reliably and frequently release valuable software. In doing so, we can help to further reduce costs. David Anderson uses the example of over-ground and under-ground trains. Over-ground trains, which run less frequently, tend to require more planning by looking at a timetable and travelling to the station at the right time to avoid unnecessary waiting. Under-ground trains, which run more frequently, tend to require less planning because it is safe to turn up and catch a train quickly. Thus by releasing quickly, reliably and frequently, we can reduce the need for much of the planning overhead.

Once a cadence has been established, and a delivery capability understood through measuring cycle-time and throughput, then predictability can be achieved through setting Service Level Agreements (SLAs). A SLA is not a promise, or a target, but a way of providing information about when deliverables can be expected. It is a gentleman's agreement, rather than a contract, that when a team accepts a piece of work, it should be delivered with a known time period.

By releasing frequently, to a known cadence, with an agreed SLA, a team can build trust that it is delivering to its full capacity

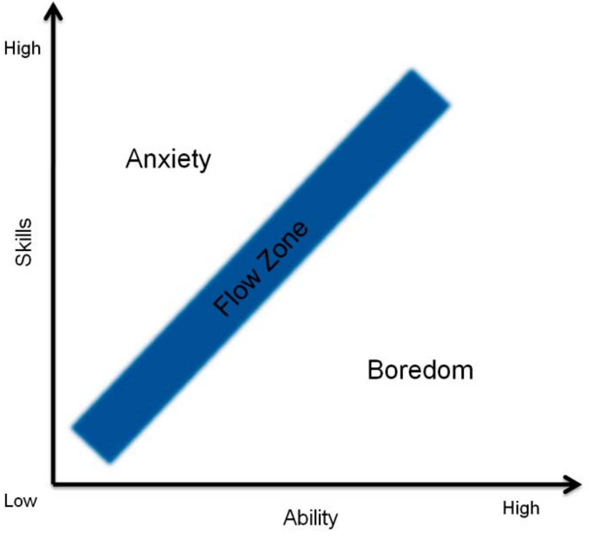
Continuous Improvement

Continuous Improvement is how a team constantly develops a Kanban System's capability to meet its purpose. A Kanban System should create an economy of flow, rather than an economy of scale, and the ultimate goal is to eliminate the Kanban System. In their book "Learning to See" [12], Mike Rother and John Shook use the phrase "Flow where you can, pull when you must". A Kanban System allows the work to be pulled, but in order to really achieve flow the team members should be always looking for ways to keep the work moving, rather than keeping themselves busy.

One approach to continuous improvement is to reduce WIP limits. When a Kanban System appears to be working smoothly, lowering a WIP limit is analogous to lowering the waterline. It will expose the rocks, and new bottlenecks and constraints will be discovered. As a result teams can work to remove the new bottlenecks and constraints until work is flowing through the system smoothly again.

Another approach to continuous improvement is through retrospectives and other spontaneous change events (sometimes known as kaizen). When teams naturally refine and grow their capability, they often discover that they consistently have free space on their Kanban Board. This is a sign that they can retrospectively lower their WIP limits as a result of an improvement.

These two approaches can be related to the states described by Mihalyi Czikszentmihalyi in his book "Flow: The Psychology of Optimal Experience" [13]. Pre-emptively reducing a WIP limit is equivalent to moving a team through a state of anxiety, where the skills required are greater the current ability. Retrospectively reducing a WIP limit is equivalent to moving a team through a state of boredom, where the ability becomes greater that the skills required. Both paths are valid and can be used in context.



Implications

Viewing Kanban Systems from these aspects creates a meta-language to help describe and think about any process. Kanban is not a methodology, but something which can be applied to an existing way of working to understand it from the perspectives of workflow, visualisation, work in process, cadence and continuous improvement.

As a simple example, it is possible to describe the typical Agile time-box in terms of limiting work in process. Don Reinertsen gave me the analogy of a bucket of water as being a container for work in process. If the bucket is being continuously filled with water, then there are two approaches to avoiding the bucket from overflowing. The first is the equivalent of a time-box. If we understand the rate at which the water fills the bucket, then we can set a cadence to empty it before it overflows. The second is the equivalent of setting explicit WIP limits. If we have mechanism to signal when the bucket is nearly full, then we use that to empty it before it overflows.

These aspects can be used as levers, adjustable in either way, to tune a process. This is a different approach to describing a process in terms of practices which are more like knobs to be dialled up to ten (or eleven). The current configuration of the levers can be used to describe the current location of a team's process on its journey of continuous improvement, a bit like a trail marker identifies a location on a forest path.

Having a wide range of configurations of processes, using these aspects of a Kanban System, means that we can employ different processes in different contexts. We then work to improve those processes as we improve the underlying contexts. Using a ski slope metaphor, we can begin with a "Nursery Slope" process for an immature team or organisation which requires lots of safeguards in place due to low skill level, and over time move the team towards an "Off Piste" process when the team or organisation are very mature and require much less safety due to their high skill level.

Being able to begin with a "Nursery Slope" process and move towards an "Off Piste" process creates an *evolutionary* style of introducing change. This is on contrast to a *revolutionary* style of jumping straight into the implementation of a new process. An evolutionary approach is appropriate for contexts where there is strong resistance, or where a revolutionary change will highlight more issues than it is possible to resolve effectively. Large enterprises, with legacy technologies, complex architectures and political silos, may struggle to make the leap to a having multi-skilled, cross functional teams delivering production code every few weeks.

Whatever approach is taken, it should be remembered that method is only a means to achieving purpose and measuring capability towards that purpose. Rather than focusing on being Lean or Agile which may (and should) lead to being successful, we should focuses on becoming successful, which will probably involve being Lean or Agile. The end goal is to be successful and a Kanban System is a means to that end, not an end in itself. To finish with a quote from "The Toyota Way"[14] by Jeffery Liker, "kanban is something you strive to get rid of, not to be proud of".

References

[1] Toyota Production System: Beyond Large-scale Production, Taiichi Ohno

[2] Freedom from Command and Control: A Better Way to Make the Work Work, John Seddon

[3] The New New Product Development Game: Hirotaka Takeuchi and Ikujiro Nonaka

[4] http://www.youtube.com/watch?v=wPFA8n7goio

[5] http://www.exampler.com/testing-com/writings/marick-boundary.pdf

[6] Drive: The Surprising Truth About What Motivates Us, Daniel Pink

[7] The Visual Display of Quantitative Information, Edward Tufte

[8] http://www.xqa.com.ar/visualmanagement/

[9] Quality Software Management : Vol. 1 : Systems Thinking: Systems Thinking, Gerald Weinberg

[10] The Goal: A Process of Ongoing Improvement, Eliyahu Goldratt

[11] http://alistair.cockburn.us/get/2754

[12] Learning to See: Value Stream Mapping to Add Value and Eliminate Muda, Mike Rother and John Shook

[13] Flow: The Psychology of Optimal Experience, Mihaly Csikszentmihalyi

[14] The Toyota Way: 14 Management Principles from the World's Greatest Manufacturer, Jeffrey Liker

Advertisement - TV Agile - Click on ad to reach advertiser web site

TV Agile

http://www.tvagile.com/

TV Agile is a directory of agile software development videos and tutorials, categorizing material on all agile aspects: Scrum, Lean, XP, test driven development, behavior driven development, user stories, refactoring, retrospective, etc.

Test Language - Introduction to Keyword Driven Testing

Ayal Zylberman and Aviram Shotten
QualiTest Group, http://www.qualitestgroup.com/

Introduction

Keyword Driven Testing (KDT) is the next generation test automation approach that separates the task of automated test case implementation from the automation infrastructure.

Test Language is not a test automation approach. Test Language is a comprehensive test approach that hands over the responsibility of automation plan, design and execution to the functional testers by using KDT based solution.

Background

Several years ago, I was a Test Manager for a large organization in the defense industry. The team was consisted of 10 test engineers, 2 of which were test automation experts and the others were functional testers. The test automation experts were focused on developing test automation scripts that covered some of the functional processes of the application.

It took me only few days to realize that things are not efficient.

The same tests that were executed automatically were also tested manually. The functional testers had low confidence in test automation scripts, since the test automation experts did not have the required knowledge about the application under test. The test automation development could start only after the application was ready for testing, and thus was relevant only for small part of the tests (mainly regression test).

I presented the problem to the team. After many discussions, we came up with the idea of letting the functional testers create test automation scripts.

The idea may be simple, but we faced a number of challenges in trying to implement it. The main obstacle was that functional testers did not have programming skills or test automation knowledge. To solve this, the test automation suggested that the functional testers would create their test cases using pre-defined words and they will “translate” this into automated scripts. The entire team favorably accepted this approach.

We proceeded to create a Keyword dictionary. The keywords were mutually defined by the automation experts and functional testers. During the first phase, the functional testers wrote their test cases using the keywords, and the automation experts translated them into automation scripts. At a later stage, we developed an automated application that took care of the translation. This enabled the functional testers to create and execute the automated tests and saved even more time.

The keyword dictionary combined with the automated translation application generated a fast return of investment. This allowed us to position test automation as a basis of our testing strategy rather than a luxury.

This was the beginning of Test Language.

Test Automation

Let's review the way Test Automation is commonly approached.

The testing process includes a number of phases test planning, test definition, bug tracking and test closure activities. The actual test execution phase is often a recursive, repetitive and manual. This phase usually described as a tedious and boring and can be carried out automatically.

To automate the execution phase, you need to create test automation scripts that are usually implemented in commercial, test automation tools; such as HP's QTP IBM's Robot, AutomatedQA's Test Complete, MS VS 2010 team system and freeware tools such as Selenium, etc..

Each test automation script represents a test case or complementary steps for manual test cases. Test automation experts create the scripts after the completion of the manual test design.

Contrary what test-tool vendors would have you believe, test automation is expensive. It does not replace the need for manual testing or enable you to "down-size" your testing department.

Automated testing is an addition to your testing process. According to Cem Kaner, "Improving the Maintainability of Automated Test Suites", it can take between 3 to 10 times longer to develop, verify, and document an automated test case than to create and execute a manual test case. This is especially true if you elect to use the "record/playback" feature (contained in most test tools, as your primary automated testing methodology.

Sanity check: Measure the success of your Testing Automation project.

Step 1: Measure how many man hours were invested in test automation in the past 6 months (include Test Automation experts, functional testers, management overhead etc.). Mark this as **A**.

Step 2: Measure how many working hours would have needed in order to manually execute all tests that were executed automatically in the past 6 month. Mark this as **B**.

If A > B than most likely your Test Automation project is a failure candidate.

SpiraTest - Click on ad to reach advertiser web site

Used by over 600 customers worldwide...
SpiraTest® - End The Testing Chaos!
Why spend money on standalone requirements management, bug tracking and testing tools?
SpiraTest® provides a complete Quality Assurance solution that manages your Requirements, Test Cases, Test Sets, Bugs and Issues in one environment with complete traceability from inception to completion.
spiraTest®
Version 2.0 Professional Edition
inflectra
> Learn More | Download Free Trial

Why Do Test Automation Projects Fail?

These are the common reasons for test automation failures:

1. **Process** – Test Automation requires changes in the Test process. The changes apply to the
   a) Test Design approaches - Test Automation requires more specific design.
   b) Test Coverage - Test Automation allows more scenarios to be tested on a specific function.
   c) Test Execution - functions in the application that are tested automatically shouldn't be tested manually.

   Many organizations fail to plan or implement the required changes.
2. **Maintenance** – Automated tests requires both functional maintenance (updating the scripts after a functional change in the AUT) and technical maintenance (i.e. - AUT UI changed from MFC to C#).
3. **Expertise** – In order to write an efficient automated test, test automation experts are required to be a techno-geek version of superman – good test engineers, system experts and great SW developers.

Obviously a different approach is required to make test automation work.

What is Test Language?

Test Language is a dictionary of keywords that helps testers to communicate with each other and with other Subject-Matter experts. The keywords replace the common English or as the basis and create an approach called keyword driven testing (KDT).

KDT can be used to achieve a number of goals:

- **Improve communication** between testers
- **Avoid inconsistency** in test documents
- Serve as the infrastructure for **Test Automation** based on Keyword Driven Testing.

Test Language Structure

The Test Language is based on a dictionary, which is comprised of words (keywords) and parameters.

Test Cases

A Test Case is a sequence of steps that tests the behavior of a given functionality/ feature in an application. Unlike traditional test approaches, Test Language uses pre-defined keywords to describe the steps and expected results

The Keywords

Keywords are the basic functional sub-procedures for the test cases of the application under test. A test case is comprised of at least one keyword.

Types of Keywords

Item Operation (Item) – an action that performs a specific operation on a given GUI component. For example set value "Larry Swartz" in "customer name" control, verify that the

value "3" appears in "result" field. When performing an operation on a GUI item, the following parameters should be specified: Name of GUI item, what operation to perform and the values.

Utility Functions (Function) – a script that executes a certain functional operation that is hard\ non-effective to implement as a Sequence. For example: wait X seconds, retrieve data from DB etc.

Sequence – a set of keywords that produces a business process, such as “create customer”. We recommend collecting frequently used functional processes such as login, addition of new records to the system as a sequence instead of implementing them as items in test cases.

Parameters

In most cases, parameters should be defined for the created keywords. The parameters are additional information required in order to generate the test conditions. For example: failed authentication by passing username with illegal password, number for mathematical calculation, etc.

Examples for sequence parameters:

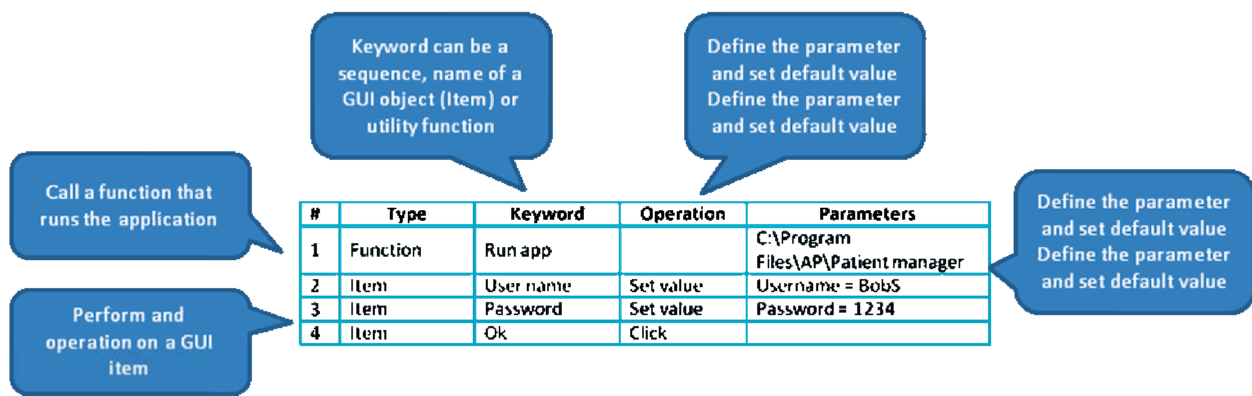
create_customer (FirstName, LastName, BirthDate, email)

Keyword   Parameters

When the user wants to create a new customer, the following syntax is used:
create_customer (Bob,Dylan,1/1/2000,bobdylan@gmail.com)

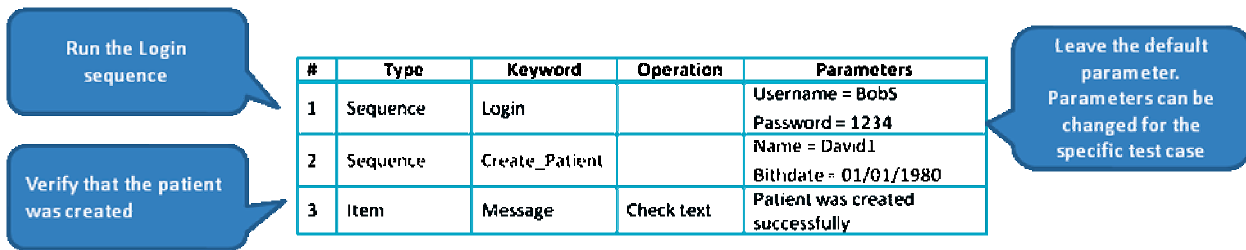
Using default parameters

Some of the keywords may contain dozens of parameters. In order to simplify the test creation, all parameters should contain default values. The tester should be able to change each one of the default parameters according to the context of the test. For example, if the tester would like to create new customer that is older the 100 years, only the birth date will be changed and all other parameters will remain the same. Obviously, a specific change will not affect the default parameters being used for other tests.



#	Type	Keyword	Operation	Parameters
1	Function	Run app		C:\Program Files\AP\Patient manager
2	Item	User name	Set value	Username = BobS
3	Item	Password	Set value	Password = 1234
4	Item	Ok	Click	

Figure 1 - Login sequence



#	Type	Keyword	Operation	Parameters
1	Sequence	Login		Username = BobS Password = 1234
2	Sequence	Create_Patient		Name = David1 Bithdate = 01/01/1980
3	Item	Message	Check text	Patient was created successfully

Figure 2 - Create Patient Test Case

It is extremely important to plan the keywords well and to optimize the amount of keywords by creating multi-function keywords (for example, creating "Change_customer_status" keyword is a better approach than creating 2 special keywords for "activate_customer" and "deactivate_customer")

Keyword Driven Testing (KDT)

KDT is the mechanism used in order to implement test language.

Keyword Driven Testing can be divided into two main layers:

Infrastructure Layer (KDT Engine), a combination of Item Operation, Utility Functions and User Defined Functions (also called Sequence), used as an “engine” that receives inputs (keywords) and performs operations on the application under test.

Logical Layer (KDT Test Case) - used by manual testers and other subject matter experts to build and execute test scripts by using a pre-defined keyword.

KDT Principles

Simplicity of test case implementation and reviewing

Functional testers should create the automated test cases directly in a simple non-code elite language. This eliminates the need for the time consuming two steps process in which testers create manual test cases and then converting them to automated scripts by the test automation team.

Web Performance Testing - Click on ad to reach advertiser web site

Web Performance Testing: Simple, Reliable, Precise, 100% Realistic
Multiple Application Modes From One Test Script:
- Functional Testing with AJAX/Web 2.0 Support
- RIA Monitoring with Easy Reporting Integration
- Multiple Parallel Playbacks for Server Loading

eValid
www.e-valid.com
Test Your Web Applications with Real Browser-Users
Free Evaluation Download

Automation Infrastructure and Test Cases Coexistence

The test automation infrastructure should be separated from the logical layer which is the test cases). This allows testers to create automated test cases at an early stage in the development life-cycle before the application is ready and even before the automation infrastructure is developed.

Plan and control

In order to maximize the benefit from KDT, a plan used to assess the on-going effectiveness of the automation program.

What to Automate

Test Language should focus on testing new features rather than on regression testing, which is the common practice in traditional test automation

In traditional test automation implementations automation scripts cannot be created before the application is ready for testing. As a result, sanity and regression tests are the main areas, which are automated. This approach usually leads to parallel executions (both manually and automatically) of the same tests.

In addition, tests, written for manual execution, are phrased in a way that does not take into account the strengths of automation scripting. The main gap between the two approaches is:

- Detail level: Test Automation requires more details (for example, after log in, the automated script should define what is expected to happened while in many manual scripts, this is left for the tester intuition).
- Coverage: When performing test automation, more test cases can be created. For example: when testing a certain numeric field that get values in the range of 1-10, usually boundary analysis technique will be used to test the following numbers: 1, 2 , 10 and 11 while when running automated tests, more scenarios can be planned.

KDT allows functional testers to plan test automation before the application is ready. This enables organizations to broaden the automated testing spectrum.

We recommend that new tests rather than translation of existing tests should be the focus of automated testing using KDT. All new tests can be planned using the KDT approach. We believe that this approach ease the KDT implementation process and greatly improve automated testing ROI.

Organization Structure and Responsibilities

The organizational structure of KDT based organization is similar to traditional test automation based organizations. It is consisted of Core Test Automation teams and Functional Testers.

However the responsibilities are much different:

Activity	Automation Experts	Functional Testers	Notes
Define Keywords		+	Keywords should be defined by functional testers under the supervision and control of the Test Automation Experts
Develop Utility Functions	+		
Develop Sequences (User defined functions)	+	+	Sequences are developed by functional testers in small scales organizations and for private use (functions that will not be used by other testers).
Develop Test Cases		+	
Execute Test Cases		+	

Figure 3 - KDT Responsibilities Matrix

KDT Process

The key factor of fully achieving the benefit of Keyword Driven approach is by full integration throughout the entire test process.

The following diagram shows the KDT process:

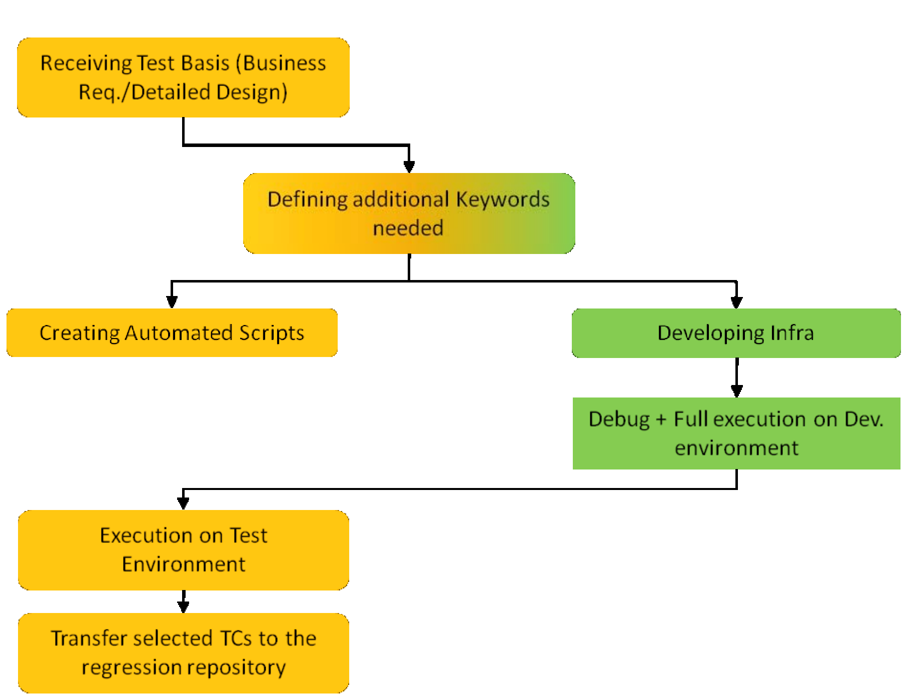


Figure 4 - Keyword Driven Testing Process

A High Volume Software Product Line

Peter Bell, peter @ pbell.com
SystemsForge, http://systemsforge.com/

Software product lines are designed to promote the efficient reuse of artifacts between projects. Domain specific modeling allows for the faster development of custom applications within a particular domain. Each provides powerful techniques for developing applications more efficiently, but together they provide a particularly effective toolkit for the faster development of custom applications. In this article I'll review an internal software product line that I have been involved with developing for a number of years, designed to rapidly build semi-custom web applications.

The article starts off with a review of our experiences and design choices. It then continues with the conclusions we have drawn from our experiences which hopefully will be of some use to anyone considering leveraging domain specific languages and/or software product line engineering on their projects to allow them to deliver applications more quickly and cost effectively. The experience report assumes a certain familiarity with domain specific modeling concepts, but if any of the terms (projectional editing, internal vs. external domain specific languages, etc) don't make sense - don't worry. They are explained along with heuristics for working with them in the second half of the article.

The Problem Space

We have found it useful to break down website development projects into three broad categories. Generally if someone has less than $5,000 to spend on an entire website including design, content and programming, we advise them to use a standard piece of software such as Wordpress for content management or Yahoo! stores for e-commerce. They may want something a little more custom, but they simple don't have the budget to get it built. At the other end of the spectrum, if someone has over $50,000 to spend on a project, (web) domain specific frameworks like Grails, Rails or Django will definitely speed up development, but you don't have to engineer all of the costs out of the project using a software product line based approach. If it takes half a day instead of ten minutes to set up a base project along with a build script and a Continuous Integration server, that's not generally a problem. In the middle, projects that cost from $5,000 to $50,000 need to leverage standard functionality wherever possible, but in a way that allows for rich customization of the parts of the application that the client cares about.

We generally find that any given client will focus on customizing 10-15% of their application functionality. Some clients really care about how their discounting system works. Others are really focused on the details of their registration process or the way that clients can navigate their products or the process of signing up for events. So the goal for these "semi-custom" applications is to be able to "generate everything, customize anything" so we can deliver an acceptable solution based on industry best practices very quickly but can then work with the client to customize the parts of the application that they really care about, extending or adapting the generated functionality.

At SystemsForge, we specialize in developing these semi-custom web applications. We provide a white labeled service for ad agencies. They sell the sites, handle the project management, design the look and feel and create the HTML/CSS. We do the programming to deliver the functionality required. Most of the projects we deliver have programming costs of between $1,200 and $5,000 so we have to deliver them very efficiently.

Around twenty percent of the projects require more custom functionality and have programming costs of between $5,000 and $35,000 so we need to have a process that blends efficiency with extensibility. Recent sample projects include simple content management systems, e-commerce stores for business to business and business to consumer clients, event calendars, newsletter systems, scheduling solutions, a ticketing system for a building contractor and an application for sharing information about sitting Shiva (mourning) for a deceased person. On a superficial level the projects are fairly different, but over the years we have found a number of ways of developing the applications more efficiently by identifying commonalities.

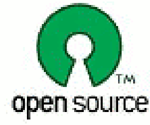
Approaches to the Problem

Originally, we started off in 1996 by hand coding websites. Fairly quickly we noticed a substantial duplication of effort between projects, so for a short period of time we utilized informal cut and paste reuse. Pretty quickly we ran into problems with different versions of various assets and issues with finding a particular piece of functionality for reuse ("which project has the latest calendar month to view display code?").

The next quick and dirty solution was to create a project skeleton. At first we created a simple skeleton to allow us to kick off projects more quickly. We then added the latest and greatest version of our various code snippets to the project and used manual negative variability - manually deleting any files we didn't need for a particular project. Before long, the skeleton ended up looking more like a bloated corpse with far too much functionality that had been poorly designed! We tried briefly to abstract optional functionality into a separate directory, but before long we realized that we needed a more scalable solution for the reuse of code.

Code Quality Management - Click on ad to reach advertiser web site

Code has so much to say !



More than 600 coding rules available
Checkstyle, PMD, FindBugs

Report Unit Test metrics
Cobertura, Clover, Emma

Project and Portfolio dashboards
Drill down from portfolio to sources

Replay the past and compare versions

Sonar is the
central place to manage
source code quality

Visit the web site : http://sonar.codehaus.org
Powered by SonarSource : http://www.sonarsource.com

The first serious attempt at formal code reuse took a component based approach. We built a framework for basic content management and then added a number of modules for common functionality like e-commerce, email newsletters and event registration. Each module supported all of the functionality required by all of the clients. They used a set of configuration parameters to configure the functionality for any given project. The configuration parameters were edited using a web based "site builder" wizard and stored in a database. This worked well for the first few clients.

As we did more projects we found that each client wanted different customizations. By the time we got to fifty projects using the module based approach - each with their own distinct customization requirements - the configuration options and the underlying code for the components had become unmaintainable.

The next step in the evolution was a rather clumsy concatenation based code generator. You would fill out forms to enter metadata describing the domain and those would drive a generator which would generate the appropriate functionality by concatenating strings to build the runtime scripts. It had the benefit of moving some of the configuration from runtime to generation time, but the concatenating scripts were difficult to maintain because you couldn't easily see the kind of code that the concatenators would generate just by looking at them.

At this point we took some time off to research methodologies for reuse, learning about best practices in code generation, domain specific modeling and software product lines. We were then able to revisit the problem space using a more sophisticated set of approaches to the problem.

A Domain Specific Framework

From the many applications we had built by this time, we had a very good idea of what we wanted our languages to look like for describing any given application. We described most applications in terms of the model (business objects with properties, relationships, validation rules, administrative rules and default values), the controllers (controllers with actions that allowed us to describe lists, detail screens, import and export functions and a number of other common capabilities), the views (which allowed us again to declaratively describe at least a first cut of the functionality required for most screens) and custom data types (a cross cutting concern that allowed us to set sensible default validation rules, generate rich admin interfaces and decorate the output of the object properties with display rules, so we could - for example - say that a field was a USPhoneNumber data type and it would validate that it was 10 numeric digits, allow us to edit it using three text boxes and would display the number as (xxx) xxx-xxxx even though it was stored as a single 10 digit string). We looked carefully at a number of web frameworks available in various languages, but at that time they were not sufficiently mature to solve the range of problems we were looking to handle and the effort to implement a facade between the models we wanted to create in our domain specific languages (DSLs) and the inputs required by the frameworks available looked like it would be greater than just writing an in-house framework to interpret the models the way we wanted.

We also thought for some time about what kind of concrete syntax to use for the DSLs. Rails was just starting to become popular so we looked seriously at internal DSLs within our host language (at the time ColdFusion - the first commercially successful dynamically typed scripting language available on the JVM - providing some of the benefits that languages like Groovy and JRuby now offer - but in 2002). However, we were concerned about maintainability and what would happen if we wanted to evolve the grammar of our DSLs so we opted for external DSLs. We looked briefly at leveraging tooling like openArchitectureWare's Xtext (for

textual DSLs) or MetaCase MetaEdit+ (primarily for visual DSLs). We also considered writing a custom parser using ANTLR, or even creating a forms-based interface which would persist the models to a database which was what we had done in the past. In the end we decided that the quickest win for us would be to start by using XML as a concrete syntax for the DSLs as it was very easy to parse, transform and emit and we could easily create a more human readable representation of the model for viewing by business users and could eventually add a forms based interface for generating the XML if we needed to make the DSLs end user writable. In short, XML was a simple starting point, a great transport mechanism and made it easy for us to implement the various projections of the language that we needed for different stakeholders.

The domain specific framework was a great start. It used runtime interpretation of XML files to effectively "pretend" that a bunch of class files existed for describing the domain objects, a service based interface to the model and the controller functionality. They were also used by dynamic runtime templates to create a first cut of most of the view screens along with a passive generation option for screens that needed extensive customization. It allowed for easy run-time extension by just dropping in real class files with actual methods to overload or support those that were described by the XML files, and despite doing so much at runtime the system was still very performant and easy to work with. In addition, the entire framework was realized in under 6,000 lines of code, so it was very easy to maintain.

Introducing the Software Product Line

The problem with the domain specific framework was that if we wanted to build common functionality like registration or a simple product catalog or shopping cart, it still felt very repetitive. The good news was that we could describe the functionality very concisely using the DSLs. The bad news was that even with concise languages it took a while to describe a rich, complex e-commerce system with products, product categories, product attributes, discounts, gift certificates and the many other domain objects required to fully describe such a system. We really needed a more efficient way to reuse our models while still having the ability to customize them. Here was where the software product line came in.

While the DSLs gave us unbounded variability, allowing us to describe any collection of domain objects, controllers and views, we also needed some way to implement bounded variability - a feature model allowing us to quickly and easily configure a first cut of the application which we could then customize from there. In the end we implemented this using a website with a forms-based interface. The system allowed a non-technical user to enter the name of a new project, to select a set of common features such as an events calendar, a product catalog, a shopping cart and a email newsletter system, and then for each to select common sub-features using a simple feature tree.

The feature tree supported essential features, optional features and alternate features (with both single and multiple select capabilities). Each feature then added one or more essential or optional statements to the model. So, for example, adding a "product catalog" feature would add a product domain object with an essential title property (there had to be a title property in the generated application). Adding that feature also added some optional statements such as properties for the product object that might be useful but would not always be required. For example, short description, detailed description and price would be optional properties (unless you added a "cart" feature in which case price would become an essential property).

Core content management
Versioning
Auditing/rollback
Email Newsletter
List size:
< 500 subscribers
< 1,000 subscribers
< 5,000 subscribers
< 15,000 subscribers
< 50,000 subscribers
E-commerce
Goods types:
Digital downloads
Tangible goods
Shipping Methods:
Fed-Ex
UPS
USPS
Custom

Figure 1: A simplified example of a feature modeling screen

The next step then was to configure the optional properties, deciding which of them to include or exclude for this particular project.

Product Properties:
Title
Price
Weight
Short Description
Description
Cost
Shipping premium

Figure 2: A simplified example of a configuration screen

The configuration information was then used to passively generate a first cut of all of the necessary XML files to describe the solution selected. We could then go in and manually edit the XML files to make changes like adding completely custom properties to the domain objects or changing the flow of the controller actions. At first we envisioned support for active regeneration of the XML files - storing the changes to the files separately and applying conditional model transforms to allow us to do things like automatically add new domain concepts or properties to applications after they had been developed and deployed. Apart from the technical challenges of the problem, we fairly quickly realized that our clients did not want their applications to change how they worked as part of an automated upgrade cycle so we decided to treat the software product line piece as simply a jump start for creating the models. This still left us with the flexibility to upgrade the implementation of our models, so at least for the simpler applications we could make substantial changes to the implementation framework without having to rework the applications manually.

Latest Focus

Recently, we have decided to make some substantial changes to the software product line. The overall structure of the system is working well, but details of the implementation are starting to become problematic. Firstly we're refactoring to use a third party web framework - Grails. While it doesn't map perfectly to all of our DSLs, we have now reached the point where Grails is sufficiently mature and the complexity of implementing some of the domain concepts we would like to express in our own system is sufficiently high that it makes more sense to implement a thin productivity layer using our software product line than to try to maintain a deeper stack in-house.

We have also continued to research DSL evolution. One of the big unsolved problems is that as your understanding of the domain grows, you often want to refactor the grammar of your DSLs. Sometimes those changes will not be backwards compatible so eventually you need to have an approach to handling DSL evolution - whether it be versioning or automated model migrations.

Finally we have also been investigating various test frameworks for describing acceptance tests for the applications so we can automatically generate acceptance tests from our requirements. We have been working on the DSLs we use to express our requirements (stories and scenarios) and the goal is to be able to generate acceptance tests directly from the scenarios in a way that is similar to what you see with the Behavior Driven Design frameworks like rspec, GSpec and JBehave.

What We're Learnt

In this section of the article I want to review some of the important concepts for working with domain specific modeling and software product lines and our experiences when working with them.

Internal vs. External DSLs

In the last few years there has been a lot of discussion about internal DSLs in languages like Ruby and Groovy - and more recently in languages like Scala, Boo and Clojure. Of course, this is nothing new. Most Lisp programs are built up using DSLs defined using Macros, but the concept of language oriented programming has been gaining wider attention recently.

Internal DSLs are domain specific languages which are implemented within an existing general purpose programming language (gppl). You add domain concepts to your program and your

DSL models are part of your application code and are parsed directly by the gppl parser. Internal DSLs are relatively easy to implement. Often you'll create a fluent interface using a builder pattern on top of your domain model API, so you can make statements like "new meeting from: 4.pm, to:6.pm" which are easier for business users to read and validate than traditional API calls. Internal DSLs also provide access to native language constructs for mathematical operations and control flow constructs, so if you do find a need to express conditionals, loops or mathematical operations, you don't have to write a parser to support that - it's available for free from the host language.

External DSLs require a little more work to implement. You might create a simple DSL that is implemented using XML, or put a little more effort into creating a custom textual DSL that is parsed either using a parser generated by ANTLR or perhaps using XText and XPand - now part of the Eclipse Modeling Framework. You could also create a visual external DSL using tools like those provided by the Eclipse Modeling Framework or MetaCase's MetaEdit+. External DSLs give you more control over your models. Users cannot just add any arbitrary constructs from a gppl. This is particularly useful for DSL evolution as with external DSLs it can be practical to apply model transformations to your existing model to upgrade existing models to work with a new grammar. That isn't usually practical with internal DSLs.

End User Readable vs. Writable

A lot of discussion concerning DSLs is often about the practicalities of end user programming and creating DSLs that business users can write. What is often forgotten is that depending on the use case you can often get most of the benefits of DSLs by creating languages that end users can read and validate - even if they can't write them.

For example, most internal DSLs would be difficult for end users to write. They have to follow specific syntactic conventions and often the error messages thrown are not very helpful in identifying what really went wrong. In our case we did end up creating end user writable DSLs using a form based interface (which along with visual modeling tools are often the easiest input mechanisms for casual modelers), but it's worth considering whether you can get sufficient value just by creating DSLs that end users can read so they can validate that the program states what they were trying to express about their domain. Often end user readability provides most of the benefits of a DSL for much less effort than creating a truly end user writable solution.

Editing Options

There are lots of ways of editing DSLs. You could use an XML concrete syntax and some kind of XML editor, a visual modeling tool, a forms based UI, a simple text editor or using XText you could even generate a plugin for Eclipse to allow your modelers to use an IDE. Of course, with an internal DSL, your modelers will probably just use their IDE or text editor of choice for their regular general purpose programming language editing.

It's important to think carefully about the kind of people who have to be able to write the models and the support they'll need in terms of editor features, semantic checking and meaningful error messages. Creating an editor that is usable by domain experts is usually more complex than creating one that a programmer could use. One thing to consider is the possibility of pairing domain experts with developers for writing DSL models. That way the programmer can help with the structuring of the models and deal with less sophisticated tooling while allowing the domain expert to focus on describing the business rules and validating that the models properly describe what they were trying to communicate.

Projectional Editing

Recently there has been a fair bit of work in the field of projectional editors. In 2005, Martin Fowler coined the term "language workbench" to describe an IDE designed specifically to support language oriented programming for the development of DSLs. Both Intentional Software and JetBrains now offer editors that try to fulfill this promise. I would argue that there is still work to be done before it's clear whether this will become a mainstream technology, but I would thoroughly recommend downloading and trying out JetBrains MPS ("Meta Programming System") to get an idea of the potential of this approach.

At a more pragmatic and immediate level, it's worth remembering that you are not limited to just a single concrete syntax for displaying and editing your languages. It would be quite possible to support a fluent interface to allow developers to quickly implement calls against your framework/domain model, to provide a forms based interface to allow casual domain experts to add their own rules and to provide some kind of graphical representation of your models to allow business stakeholders to get a high level view of all of the models within your system.

Bounded vs. Unbounded DSLs

Another important distinction is between bounded and unbounded DSLs. Unbounded DSLs are used to deal with solutions where there is a theoretically infinite solution space. A good example would be a language for describing domain objects and their properties. While there are a relatively small number of domain objects that come up on a regular basis, such a language is capable of describing an unlimited number of potential models.

Bounded DSLs are designed for configuration of systems with a limited number of possible configuration states. If you find that you need a bounded configuration DSL, make sure to research existing solutions for feature modeling. There are well proven patterns for describing configuration DSLs and it is worth reviewing them to help with the design of your specific configuration DSL(s).

Runtime Interpretation vs. Code Generation

While we started off by thinking that we were in the code generation business, we realized pretty quickly that actually we wee in the domain specific modeling business. There is nothing wrong with generating code. It is one of the many ways of implementing DSLs and has strengths and weaknesses. There are a number of benefits to code generation. From an intellectual property perspective it allows you to deliver to clients a specific solution for their needs rather than giving them a copy of an entire system for delivering a whole class of possible solutions. It also makes the runtime code easier for less experienced programmers to understand as the generated code will often be simpler than a runtime framework that will interpret your models. If you have a proven performance issue with a runtime framework, pre-compiling your models into code can also improve performance if that really is the bottleneck.

The main thing to ensure with code generation is that wherever possible you use active code generation so that you can regenerate the code as your models change without losing any custom code. Generally this is done by putting any custom code into different files than the generated code - often using some combination of subclassing, AOP, event driven programming or (where your language supports it) include files, partial classes and/or mixins. If you have to mix generated and custom code within a single file, protected blocks with some kind of unique multi-character delimiter between editable and generated code is a possible solution, but it's generally better to keep the generated and hand written code in separate files wherever possible.

Avoiding the Customization Cliff

Another important issue to look out for is what Krzysztof Czarnecki termed the "Customization Cliff". The ideal solution should make it very easy to do simple things, and just a little harder to do more complex things. All too often you will see a software product line which supports easy point and click selection of features, but where making a change to a feature requires writing C++ code! You want to have lots of small steps each of which give you a little more control rather than just dropping off the edge of a cliff when you hit the limitations of the expressiveness of your DSLs.

DSL evolution

Finally, think about how your DSLs may evolve over time. Generally there are four stages that a DSL will go through. To start with you might try to avoid having to change the DSL. You simply do what you can with the expressiveness that it provides and then write custom code as necessary for any special cases.

After a while, the custom code for the special cases gets repetitive and the pressure grows to allow your DSL to become more expressive. You decide to allow changes to the grammar, but only in a way that will preserve backward compatibility. That works for so long, but eventually you often find yourself needing to make a change that will not be backwards compatible. A common approach to that solution is to start to version your language, with all models having a version and automatically being run against the appropriate version of your DSL.

That works for a while, but eventually you run into issues with old models needing new language features or with the overhead of having to maintain multiple versions of your DSLs. If you have a large number of models, at this point it's worth looking at the possibility of applying model transformations to automatically upgrade old models to work with the new version of your language. Not all grammatical changes to a language allow for automated transformations, but at very least you should be able to automate some of the model transformations and provide an efficient UI for human intervention in the case of models that need to be upgraded by hand. Model transformation is typically easier to apply to external DSLs, but if you find yourself with a lot of internal DSL statements and need to apply model transforms, consider the possibility of writing a script that will walk the in-memory runtime model and generate a set of models in a new external DSL that fully describes the current state of the model. In that way you can refactor to external DSLs and then apply your model transformations to them more easily.

Conclusion

This problem domain has ended up being a really interesting space for examining best practices for developing software more efficiently. On the one hand, the very constrained price points we deal with mean that we have to get every last bit of efficiency out of our software product line. On the other hand, continued investment in the software product line over a number of years has allowed us to investigate a wide range of approaches and to get some real world experience of the relative benefits of different approaches for different classes of problems.

Better Requirements Definition Management is Better for Business

Genefa Murphy, genefa.murphy @ hp.com
HP Software and Solutions, www.hp.com/blogs/requirementsmanagement

Why focus on requirements definition management in the application lifecycle?

Increasingly, smart businesses are looking much closer at requirements definition (RD) and requirements management (RM) (sometimes grouped together under the Gartner-coined phrase, requirements definition management (RDM)) to streamline the entire application lifecycle. Why? Because systematic and effective RDM captures software defects earlier in the lifecycle, and it reduces the overall likelihood that defects will be introduced. That's important. How important? According to one study, **the cost to fix a defect after delivery is more than 100 times the cost to fix it in the requirements and design phase**. [1] No business wants to be hit with that bill. Now to add to this the growing interest in agile development techniques as a way to deliver higher quality applications and we have an interesting recipe for success.

What exactly is RDM?

Before we can apply solid RDM practices, we need to understand what RD and RM are. Requirements definition and requirements management are the terms used to describe the process of eliciting, documenting, analyzing, prioritizing, and agreeing on requirements, and then controlling, managing, and overseeing changes and risk. In its most effective use, RDM is a continuous process that occurs throughout the application lifecycle. Now, how does this change when it comes to agile? Well in my opinion the basics shouldn't change – the only difference is that in agile this process is a lot more iterative, probably has more stakeholders involved and the currency we will probably be speaking in is user stories and tasks and not requirements per se.

So keeping this in mind what is the difference between a requirement and a user story? In my mind its all about structuring of the requirement as essentially both a requirement and a user story do the same thing – they describe a specific condition or capability needed by a user to solve a problem or achieve an objective. The difference between traditional requirements and user stories is that in traditional waterfall environments, requirements are normally broken down first into two blocks functional and non-functional and then into a deeper layer of granularity, layering business requirement with design specs etc normally in a hierarchical manner. Conversely in agile these specifications are usually first wrote as "user stories" and then the non-functional elements of the user story are captured as "fit criterion". At the heart of the user story approach are the usages of short and concise sentences which use the everyday language of the business user to describe the functionality that should be implemented. If the further detail is required the user can either tie several user stories together to form an epic or super story.

Given this what information should go where?

In the functional requirements or the body of the user story the aim should be to articulate the system-wide elements such as the main product features particular to the business (say order processing for a shipping business). Coupled with this in the non-functional requirements or "fit criterion" the user should aim to describe what elements should be included in order to enhance the user experience with the software.

Common non-functional requirements include:

Usability: This includes looking at capturing and stating requirements based around user interface issues—things such as accessibility, interface aesthetics, and consistency within the user interface.

Reliability: This defines requirements such as availability levels, computation accuracy, and recoverability of the system from shutdown or failure.

Performance: This involves things such as throughput of information or computation through the system or application, response time (which also relates to usability), recovery time, and startup time.

Supportability: This specifies a number of requirements such as testability, adaptability, maintainability, compatibility, configurability, installability, scalability, localizability, and the like.

Both functional and non-functional requirements vary tremendously with the type of application or product being developed. As we have already discussed they can take the form of traditional requirements specifications or more agile like user stories. Similarly, they can be defined and captured in simple text format or visualized more elaborately with pictures, business process flows and even simulations. It all depends on your organization's development method.

To visualize or not to visualize

Whether a requirement is textual or needs to be visualized is a product of the requirement itself. How do you know? Here are some questions to ask:

- Will visualization facilitate more agile and rapid requirements definition or will it add time to the development process unnecessarily?
- Will requirements complexity (multiple sub-requirements, for example) result spider-web in diagramming that over-complicates definition?
- Will requirements be visualized by type? For instance, visualizing the user story and textualizing functional requirements associated with the user story.
- Will requirements be visualized by priority or risk? High-priority or high-risk requirements may be visualized while low-level requirements are not.

MKS Enterprise Agile - Click on ad to reach advertiser web site

Visualization is a useful tool to help business communicate better with IT. Answering these questions can give you an indication of the level of visualization requirements your project demands. Nevertheless, you may also find that visualization is not required.

Once you have requirements, what do you do with them?

It is not enough to get your requirements and or user stories on paper or in a tool, distributed to your team, and then hope that they are met. Once requirements are defined and captured, it is essential that they are managed actively and correctly. This is even more important in agile as each iteration is a pressure cooker where user stories and requirements need to be defined, estimated, re-estimated, developed and tested in a short period of time. Here are some basics tenants to follow for successful RM:

Prioritize requirements and user stories so resources are assigned to address the highest-priority, highest-risk requirements. This helps prevent a lesser requirement (with perhaps a more vocal stakeholder) taking precedent over other more important requirements.

Baseline requirements to set measurable success standards/thresholds for all requirements so that all stakeholders are on the same page. Achieve sign-off agreement between business analysts and between business and IT.

Track changes to compare current baseline requirements against pre-defined thresholds. Business analysts can flag requirements exceeding change thresholds, assign risk, and take appropriate action - for example doing this may show that a user story is too large as is and needs to become an epic with multiple user stories under it.

Defining, capturing, and managing requirements properly in this way can add tremendous value to business, IT, and to the ultimate success of the software application or service developed. Improper RDM, or merely choosing to place less importance on it, can result in a cascade of negative effects that not only affect software in the development and testing stages, but the business as a whole. For example, according to Gartner, 40 percent of unplanned downtime is caused by application failures, costing an average of $100k per hour for mission-critical applications. [2] A large portion of those failures are due to poor requirements introduced during the development phase.

The problems of poor RDM

The unfortunate results of poorly defined and managed requirements have been known and even quantified for some time. Defects are most often introduced early and found later in the application lifecycle. This is, in part, because business analysts have historically worked in an ad hoc way using word processing and spreadsheet software. Those were the only options available until recent RDM-specific software tools helped them work more systematically.

Working ad hoc with tools not designed specifically for the job naturally leads to problems. The National Institute of Standards and Technology (NIST) estimates that 70 percent of software defects are introduced in the requirements phase [3]. The later defects are found in the application lifecycle, the more expensive they are to fix. This issue is compounded in agile because if each iteration is bogged down with defects from the start how are the teams meant to move forward and isn't one of the aims of agile to produce higher quality software?

So what is the real cost of weak RDM to business?

According to voke Inc.'s recent Business Analyst Survey, a software development project can cost anywhere between $1 million and $20 million. [4] An average project typically has the following profile:

- People per project: 75
- Project duration: 17.2 months
- Cost per project: USD$3.2 million

So, based on an average project spend of USD$3 million and factoring in the estimated effect of poor RDM, an average project could end up costing as much as USD$5.87 million - nearly double its original cost. Furthermore, if we estimate that this happens in 5 out of 10 major projects per year, that is an estimated total yearly cost of almost USD$30 million. That kind of money does not merely affect IT and software development. That level of unneeded expenditure affects every part of the business and the overall bottom line.

Superior RDM, superior project results.

The pillars of good RDM: Business analysts and software quality managers

As we've stated, poor RDM introduces a wide range of negative effects to business and IT. While defining and managing quality requirements and user stories early in the software development lifecycle helps verify that the final software or application product delivers what the business needs and customers want - effectively and efficiently, it also mitigates release date extensions and costly fixes later that may negatively impact resources and business-critical functions.

Applying effective management of requirements used to be the realm of the business analyst. However, experience has shown that requirements are more accurately defined when a joint effort between business analysts and software quality managers exists. With agile this even more true as in agile all stakeholders should be involved in the development of the user story so that it is properly estimated and defined so it can be auctioned by DEV and QA. Here is a quick look at the general responsibilities of the different stakeholders involved in the life of a requirement / user story:

Business analysts must properly:

- Write a succinct user story in the business language
- Define unambiguous requirements
- Establish the business value of each requirement
- Quantify the risk associated with each requirement
- Understand the dependencies of each requirement

Software quality managers have critical questions to answer in relation to the requirements business analysts define:

- Are the requirements verifiable when implemented?
- Are the requirements realistic, and how can they be implemented and tested?
- Where do I assign testing resources for increased

Developers should:

- Be involved in the definition and estimation of the requirement to ensure realistic times are allocated to the user story for development purposes
- Make sure they have enough detail in the user story to start developing

All stakeholders manage requirements changes during the entire application lifecycle. Together they must:

- Determine how a requirements change affects other requirements
- Figure out which tests are affected by requirements changes
- Decipher if changes introduce new risks and the level of those risks
- Calculate how changes affect development schedule and release

When business analysts work in unison with other SDLC stakeholders using strong requirements management practices, they can reduce changes, defects and delays and potential disruptions in the application lifecycle can be smoothed out.

Agile 2010 Conference - Click on ad to reach advertiser web site

ORLANDO, FLORIDA
AUGUST 9-13, 2010

Presented by

Jim Newkirk, Chair
agile2010.agilealliance.org

Register now at http://agile2010.agilealliance.org/register.html

Everyone in the application lifecycle wins with strong RDM

Positive results from strong RDM are not limited to business analyst’s. Concrete requirements definition and management help deliver a product that is more in line with business needs, budgets, and schedule. It does this by helping everyone in the application lifecycle stay on task and add more value to the overall project:

- Business analysts can show stakeholders why certain requirements take precedence over others
- Designers know exactly which application characteristics, such as performance, scalability and usability, to emphasize and why
- Coders understand the level of resources they should devote to developing specific functionalities
- Software analysts can assign testing resources more efficiently based on requirements importance
- Test planners can determine the level of test effort necessary for each requirement. Plus, they can determine if reducing or increasing test efforts is justified given the business value of each requirement.

If more focus on RDM is encouraged and implemented from the top down, not only are projects more likely to meet intended business metrics, but also all the players involved have a better understanding of what is required of them, what they need to accomplish, and why. Which ultimately leads to a team that is armed with the information it needs to develop software that successfully lines up with business and user expectations.

So what is the industry waiting for? It is time to get a handle on the intricacies of requirements definition management and implement them.

Understanding the phases of RDM

Successful RDM implementation is a phased approach. For business to really get the most out of employing stronger requirements definition management practices, it is important to understand and utilize the specific phases of each. With a working knowledge of each of these phases, it is quite easy to implement more effective RDM throughout the application lifecycle.

Requirements development phases

Elicitation is the stage where business analysts gather the requirements (sometimes called trawling) and deliver them to the product backlog for assessment.. Requirements elicitation practices typically include JAD (joint application development) techniques such as interviews, questionnaires, user observation, workshops, brainstorming, use cases, role-playing, and prototyping and in agile especially this process will involve many contributors.

Elaboration stage adds more depth to the meaning of each requirement or user story maybe breaking it down from epic to user story to task or to technical details. Methods used by the business analyst in this stage include developing use cases in rich text, creating flow diagrams, class models, GUI mock-ups, and assigning business rules. The elaboration phase can also help to address known risk factors and establish/validate the system architecture.

Validation verifies that requirements are complete (and testable). The requirements document is checked for ambiguity, conflicts and errors, and so on and developers and testers should check the "fit criterion" associated with the user story. Techniques used at this stage include discussions, simulations, and face-to-face meetings.

Acceptance is the final stage in the requirements definition lifecycle and occurs only when the requirements have been verified and accepted by all the stakeholders. It is here that the business analysts create a baseline of the requirements and they are allocated to releases and iterations so that technical development can start and test planning begin.

Requirements management phases

Requirements prioritization phase gives hierarchy to a project's requirements set. Requirements should be prioritized based on customer need and business risk.

Establishing each functionality's relative importance enables the greatest product value to be delivered at the lowest cost. Collaboration between the business and IT is key here. IT does not always know which requirements are most important to the business, and the business cannot always judge the cost and technical difficulty associated with each requirement. This phase also adds objectivity to the application lifecycle. Too often individuals believe their requirement is most important due to a context-skewed perception of that requirement.

Traceability deals with the association that exists between requirements and other entities in the lifecycle. These relationships can exist on many levels in the context of requirements (user stories)

- Requirement to requirement
- Requirement to business process
- Request to test
- Requirement to defect

Overall, traceability between requirements and other project artifacts allows a business analyst to manage scope creep and verify all requirements have been met.

Change is a requirement alteration, addition, or deletion. A change can occur at most anytime and for a multitude of reasons. Some of the more common ones are listed here:

- **Missed functionality** : A stakeholder working with an existing system could simply realize that it is missing a feature - a new user story will likely be added to the backlog
- **New defects found**: A bug, or more importantly the need to address the bug, should also be considered a requirement.
- **Incomplete understanding**: The business or IT realizes they do not understand their actual need fully. It is common to show a stakeholder your working system to date only to have them realize what they asked for really is not what they want after all. This is why active stakeholder participation and short iterations are vital.
- **Politics:** The political landscape within an organization is always dynamic. When the balance of power shifts amongst stakeholders, so may priorities. This often motivates changes to requirements.
- **Marketplace changes**: For instance, a competitor releases a similar product which implements features your product does not.

- **Legislation changes**: New legislation may require new features, or changes to existing features, in your software.

The main role of requirements management is to control and manage the impact of changes to the defined operational need so that all stakeholders in the lifecycle have visibility into what alterations have been made.

Status tracking is made possible through traceability. Having the link between requirements and other assets allows the business to get a better idea of the true quality of the application and its ability to go live. Similarly, comparing the number of requirement changes to baseline, and tracking how many defects are associated with requirements, shows the business analyst how accurate requirements were in the first place. Too many changes or defects are an indication that the requirement may be inaccurate - the sooner this is identified the better.

The phases of requirements definition and management may appear tedious. However, the fallout of omitting or glossing over them is infinitely more expensive to business and IT when defects and delays appear. That is why businesses are starting to take a look at applying RDM more systematically and thoroughly.

A different view of the application lifecycle

Why wouldn’t business focus on RDM throughout the application lifecycle?

Forward-thinking organizations are already applying RDM across all phases of the application lifecycle. But even more could. Why aren’t they?

First, it is quite common in the industry to look at the application lifecycle management (ALM) process through only a technology and development lens. This is seemingly where all of the action is. Second, there are resource investments required to implement strong RDM practices. However, those investments are nowhere near the costly consequences of weak RDM. From my point of view the recipe to success means defining the right requirements to work on and selecting the right technology to achieve the businesses goals within the right timeframes.

However, it doesn’t need to be difficult. With software solutions now on the market to help stakeholders manage requirements and many solutions providing hooks into other SDLC tools in particular aimed at bringing the business analyst, the tester and the developer closer together, it has never been easier to implement better RDM practices.

References

[1] B. Boehm and V. Basil, “Software Defect Reduction Top 10 List,” IEEE Computer, January 2001

[2] Gartner, From Concept to Production, Software Changes and Configuration Management, April 2008 Management,

[3] NIST 2002 RTI Project 7007.011

[4] voke Inc.’s recent Business Analyst Survey

The Core Protocols, an Experience Report

Yves Hanoulle, www.hanoulle.be

Somewhere in a bar at a European Agile conference.

"Hello Yves."

Yves turned his head to the voice, but did not answer back.

She continued, "I played your leadership game a few years ago."

Yves who, had run the session at a dozen agile conferences, frowned in an effort to remember her name.

"At that moment you were about to leave for a McCarthy BootCamp. You were both excited and afraid of doing so."

Yves kept silent for a second.

"Will you spend some time with me and share your experience?" she asked firmly.

Yves replied, "Hi Allison, glad to see you back. I'm delighted you remember that. Yes, I will!"

She smiled.

After a little pause, Yves continued, "The week after we talked, I went to a Core Protocols BootCamp in the US. Before I share my experiences of that week, and of using the Core with teams in the five years since, let me start with a little history of the Core. Where you in the session yesterday about high performing teams?"

Allison: "yes, as a CEO I want to know how I can turn my company into a high performance company."

"High performing teams have been a big buzzword in IT over the last 20 years," he reminded her. "Although everybody talks about this, there is limited experience in how to create such high performance, results-oriented, successful teams. Luckily for us, Jim & Michele McCarthy created an experiential workshop in 1996 to learn about it. Over the last 14 years, they have recorded communication patterns (they call these the Core Protocols), that help create high performance teams."

Allison turned silent for a second, but she kept looking at Yves. Interest had sparked in her eyes. Something else, not so nice, also. "That is quite a strong statement," she merely said.

"Yes, it is," Yves answered. "And I had my doubts when I first read about these patterns," he quickly added.

"So is it the silver bullet they claim it to be?" She might have been cynical, but there was no trace of it in her voice.

"Mmm," Yves mumbled. "I don't believe in silver bullets, but I do believe in tools and techniques that I can use to help teams. When I first read about these patterns, although I liked them, I found all sorts of excuses not to use them," he remembered. While she looked away, as if she was now in some other universe, Allison said, "That reminds me of the people I encounter who are afraid to use or start using agile." As this was exactly what he was trying to say, Yves knew they were connected. He continued, "When I realized that, I said to myself it was time to get outside of my comfort zone. As some of these patterns were so different from what I was used to, I needed to have a safe place to experience them. I was too scared to try them out in a

team, or even talk about them." She smiled again as she teased him, "So that's why you did not want to answer my question back then."

She was right and he knew it. "Yes. That's why I'm really happy you have asked me this question again. Let me see, where should I start?"

"Yesterday, Jeffrey said that teams needed a shared vision. Do you agree with him?" she tried to link what she already knew and this new stuff. "

Great question," Yves felt the adrenaline, and he continued, "Indeed, a lot of management books talk about having a shared vision for a team. Often people (like Jeffrey) make the mistake of thinking of such a vision as a *vision* or *mission statement*."

Surprised by his answer, Allison asked, "You don't think the exercise where Jeffrey told us to create a vision statement helps to focus a team?"

Yves remembered his resistance for the exercise, "That's correct, for me a shared vision is about a state, not a statement. And having management creating a statement for a team is even worse."

"Shared vision is a state, not a statement," she tasted the sentence in her head. Yes, it felt right. Yves continued, "What I learned from BootCamp is that a true shared vision is the state where a team all thinks in a direction, sharing a picture of a desired future."

"People thinking the same; don't you want diversity in your team?" she asked.

"No, no, thinking in the same direction is not thinking the same," he replied quickly. "Let me tell you a story to clarify this." He knew this was hard to understand. "When I organized a BootCamp in Europe, I arranged an interview with the trainers and a business magazine. As I walked into the interview room, I told Paul and Vickie (the trainers) that my most scary idea would be to ask Benny (the reporter) to join us on Friday when the team had to deliver their BootCamp results. Halfway through the interview, I got a phone call from the team (that was out on an adventure), 'Yves, will you ask the reporter if he wants to join us on Friday...' That is an example of being in a state of shared vision. This particular shared vision was created by working together over the course of three days."

"Three days, gee, they must have been lucky," Allison thought skeptically.

As if he knew what she was thinking Yves added, "I have seen examples of a shared vision state in all the BootCamps in which I have participated."

"Creating a shared vision state in a repeatable way?" Allison could hardly believe what she was hearing. "Wow, that sounds powerful. Tell me how this is done."

Yves, who knew how unrealistic this might sound to people who did not have the same experience, felt it was time to get to more practical examples. "Patience my dear Watson. Before I can do that, we have to look at the basics. The behavior patterns (called Protocols) of the Core."

"Then, let's find some seats," said Allison. "I'm tired of standing up with my drink in my hand, and something tells me I want to sit down for what you've got to say next!" After her initial talk with Yves, she had tried to read 'Software for your head' and although she had given up, she somehow felt she would like the Core.

"As a coach, I give feedback to teams and individuals all the time, and as I currently see the **Perfection Game** as the most powerful way to give feedback, the Perfection Game is the pattern I use most."

Allison, who had been reading his blog, said, "Yes, I've noticed you use it a lot."

Yves, surprised that she knew the Perfection Game, saw that as an opportunity to do a little experiment. "Let's try it out right now. Will you perfect this conversation?"

This unexpected shift made Allison catch her breath. She looked down. She had read his posts, even printed out the Perfection Game, but she had never actually done one. And now this expert was asking her to perfect this conversation. She hadn't even had time to think about the conversation. Hey, they were only talking for five minutes.

Yves waited, he knew it was quite a challenge for someone who had never done this. Especially because he had not repeated the pattern of the game with her. He had done that on purpose, so he would be able to see what parts she understood and what parts were still unclear.

She looked up. She came to the conference to get out of her comfort zone. Yves seemed like a nice guy, and he seemed to be determined to have her do this...

"I'm giving this conversation a 6 out of 10. What I like about it is:

-you are taking time to talk to me,

-you seem to have become even more passionate about the Core as I saw before,

-you are making links between the Core and the agile stuff I know,"(Allison felt it was going well)

"To make it a 10:

-we should take time to talk about all the core protocols,

-I want this conversation to be fun,

-I want to see links with Scrum,

-I don't like the military name BootCamp."

Yves interrupted her, "Protocol check." Allison facial expression showed she never heard of that before.

Without missing a beat Yves continued, "In a Perfection Game, you are only allowed to say what you want to have, not what you don't want."

"Damn, I knew that." Allison's mind raced fast, "OK, let me rephrase that last sentence. I want to understand why it's called a BootCamp."

Yves felt happy. "Cool, you seem to get that one. Any questions?"

That remark annoyed Allison. "Aren't you going to reply to my remarks?"

Yves realized that he had gone too fast. "No, these are your ideas. Even if I don't agree with something, they are still true for you. They help me when I create a new version of what I asked you to perfect. In this case, I will in the next parts of the conversation remember what you said and use some of your ideas."

Now Allison really was not happy. "Some, not all of them?"

He tried to ignore her mood; and replied monotone, "No, not all of them. When I don't like an idea, or don't know how to apply that, I leave it out." He paused.

That seemed to have calmed her down a bit. "Please say more."

Yves felt an example might do the trick. "For example your idea of Scrum. I prefer not to link Scrum and the Core protocols."

Allison started to mirror Yves' teasing style. "Why not - they don't work together?"

Yves was clearly amused by her remark. "Haha, no they do. I see them as different tools in my toolbox. When I explain one, I leave the other out as I think it will confuse my audience."

Allison felt both happy and sad. Yves seemed to have an answer to everything. She said (as calmly as possible), "OK, that's fair."

For Yves that was a sign to push her further. "Will you tell me, why did you rate it a 6?"

"Isn't that obvious, I had 4 ideas: 10 minus 4 is 6," she said, surprised by his question.

"That was what I was afraid of," Yves quickly replied.

"What did I do wrong?" Allison's voice sounded upset.

Yves ignored her emotion and told her, "The number you subtract, should be about the value that your ideas have to you."

"I can't do that!" Allison's voice now sounded shrill.

Yves: "Why not?"

You could hardly hear her voice when she said, "That number is too high."

Not surprised by her answer he asked, "So you are telling me, you can't give me a low number?"

A little louder she said, "Yes."

Clearly amused, Yves asked again, "Why not?"

"Why not? Is he laughing with me? I avoid humiliating him and he laughs with me." Allison was really confused now. "I liked the conversation so far, giving a low number would give the wrong impression."

"Aha!" Yves slapped his hand on the table, "That is another misconception about the Perfection Game. The Perfection Game is not an evaluation. It's a feedback tool. A feedback tool to communicate ideas."

Surprised she said, "So you won't feel bad when I give you a low number?"

Yves smiled. "That's correct. Please tell me what your ideas are worth to you."

I hope he means it, she thought as she said, "Then I give this conversation a 2 out of 10."

Yves' smile doubled. "Wow, that's nice."

Allison turned away, "you see, you feel hurt."

"No!" Yves said firmly, "I said: that is nice."

Allison replied, "Yeah right, we both know that was sarcasm."

Yves refuted, "No it was not. I'm really happy you gave me ideas that for you are worth 8 out of 10."

Allison looked at him as if he was from Mars. "Boy, you really are strange."

Yves agreed, "Thank you. Let me go over the format."

"As you noticed, there are three major parts in a Perfection Game:

-a score from 1 to 10,

-things you like; what you liked about the thing you are perfecting,

-improvements."

Allison: "Why is perfect a goal?"

Yves: "Great question. It's not. Although the name seems to insinuate we strive for perfection, the game is more for finding idea's to make things better. I would have preferred the greatness game."

Allison wondered, "That score: how do you deal with that with people that have no experience with this?"

"You mean what to do with people who still feel they are evaluated as a person?" Yves asked.

"Yes," she nodded.

Yves: "As a coach I usually start with a score of 7 out of 10. That is, I am aiming to give improvements worth 3 out of 10. When I'm listing my improvements and I feel my improvements are not worth 3 out of 10, I adjust my score. At XP days Benelux, the Perfection Game is used in its original form, to improve the proposed sessions for the conference. A lot of speakers appreciate this kind of feedback. I have seen the Perfection Game being adopted as a kind of evaluation. That does not work. The Perfection Game was intended for improvement and not for evaluation. Mixing these two does not work. That is also why the third part is improvements, and not things we don't like. It's very easy to come up with a long list of things we don't like about everything. To come up with a small list of improvements is a different thing." Yves paused for a moment to make her think about that last message. Then he added a last remark, "That is why for me, using the Perfection Game is a serious engagement."

While she ordered some more drinks, Allison asked about emotions. "I spend eight hours a day at the office. That is at least the same amount of time I spend with my family. Some people claim we should leave our emotions outside the office and behave professionally. I can't do this. And when I try, I lose all of my creativity."

Yves could not agree more. "You are right, that is impossible to do. For a long time I have been puzzled about using my emotions in a smart way inside the corporate world. And then I learned about **check in**, and a new world opened. Let me now start with an example:

I'm checking in:

I'm glad you asked me to talk about the Core protocols,

I'm mad, sad, afraid I might be in my bed late again,

I'm sad I miss my family,

I'm mad I forgot to call my kids today,

I'm in."

Yves turned to her and asked, "So, how do you feel when you hear something like this?" Yves waited a minute or two to let her reflect, then he continued, "Let me tell you about my first reaction. My first reaction when I learned about this was 'Wow, that is powerful'. My second, 'This won't work with me for …'."

Allison nodded. "Yes, it won't work in -"

Yves interrupted her. "My team, my company, my country. Give me any reason [1] why it won't work in your team, your company, your country: I had them all. Let me tell you about a chat I had with Michele. At that time I was already convinced it would be a good way to communicate with people, it might even work with children, but not yet with my children; I considered them too young. Michele McCarthy told me to try it, and then we could talk again. (For her it had no sense, that I would say, 'No, it does not work', without trying it.) I'm not sure what I thought at that moment, not even sure if I wanted to try at that moment. The very next day, Joppe, three years old, came home with a self made card. On that card were listed the four basic feelings

(mad, glad, sad, afraid). Joppe had learned something like check in at school. That day, I realized that I was blocking myself from using a powerful tool with my kids. Since that moment, I check in with my kids, when I put them in bed."

Allison moved by the story, wondered how it would have been to check in with her father. "Tell me about the format of the pattern."

"The first part, 'I'm checking in'," Yves explained, "that sentence asks for attention, telling you I'm going to disclose my feelings, please pay attention. An alternative that is used in a team setting is 'Let's check in'. Here you check in, but you also ask everyone else present to do the same."

Allison asked, "Why are only four feelings allowed? I have more then four feelings!"

"Great question," Yves answered. "You are correct; only four feelings are allowed. More complex feelings are actually a combination of these four. Yes, it is harder when you are limited to only four feelings. You won't hear me say the Core Protocols are easy. They are simple, not easy."

Allison tried if she could fit her complex emotions into these four basic emotions.

"That last sentence, 'I'm in', tells the people that I'm finished." Yves added, "The response to this is 'Welcome'."

"'Welcome'? That makes me think of AA or therapeutic groups I saw in movies," she complained.

Yves looked at her and said, "Get over that. When you do this, you will realize that it feels really nice to be welcome in a group. I guess that is why they do that in these groups."

Allison wondered, "Isn't it scary to do this at work?"

Yves honestly told her, "Yes, it is scary to check in at work, especially as a coach with a new team and yet the only advice I can give, is to try it."

Allison - still not convinced: "The format seems rather rigid. Do I have to use it like that?"

Yves reminded her about a rule that works great for all agile methodologies and tools." I advise people to start using these patterns as described, then when you are good at it, you can be more flexible. When things get tough (as they always do), you should be stricter again." You could feel Allison was already more convinced as she asked, "When do you check in?"

"That's easy," Yves replied, "Every time you feel the need. In a well running team, people check in multiple times a day."

"If there is a check in, is there also a check out?" Allison wondered.

Much to her surprise, Yves said, "Yes, there is. My partner introduced me to check out, 13 years ago. When we had a discussion that became too heated, she left my house to go for a walk. I was very frustrated with that at the time. It took me a few years to understand I was not frustrated that she left the conversation; I was frustrated that we did not restart the conversation when things cooled down." Yves was now really in a flow. "A check out goes like this: Whenever you feel you are not capable of adding something valuable, you say 'I'm checking out' and you leave the room. There could be all kinds of reasons for this: like being ill, or being too angry..."

Allison added, "This would be good for those companies that focus on being present and not enough on being productive." Yves agreed. "I prefer that everybody being present is a person I can count on, over counting on everyone being present. I like the visual management of being able to count on the people that are present." After nipping from his drink, Yves continued. "I use check out personally as a kind of personal time out (you know the kind of time-out I ask my

kids to take when they behave badly). I know coaches who use this when they feel that their body is not OK (migraine etc).

In those companies where being present is more important than working, I see people going to cigarette breaks, or the healthier long toilet visits to have a check out. It's much nicer if people can do this officially.

Allison said, "But what if something has to be done?"

Yves replied, "Well, if people are able to do that, they will. If not, forcing them to stay won't help. (It will actually make things worse as the people not checking out will limit the result of the people that would be able to do the work.) On top of that, a five-minute break can make people 35 times more productive than if they would not take that break. It is a good practice when you check out - that you mention when you intend to come back (if you know). When you check out, you should leave the room immediately.

"Like the law of two feet in Open Space technology?"

"Actually, I think its law of two feet and check out are completely the same. During one of my BootCamps, I needed to finish a report about another training. I did not want to miss anything so I wrote the paper in my team room. That was a big mistake, I should have checked out. I was not productive for my team, and I gave them the message that integrity was not important.

Allison looked astonished. "I'm surprised to hear that, I saw you leave multiple sessions this conference."

Yves: "Yes, that was something I learned at that BootCamp, I have been much more productive since then."

Allison wanted to get moving. "What other patterns are there to learn?"

Yves: "Let's talk about **ask for help**. In the daily standup of a Scrum, we answer three questions. The last one is 'are you blocked'. I tell the teams I'm coaching, it's really 'where do you need help?'"

Allison: "Does it really matter? Most teams don't even bother about that last question." Yves: "I'm not surprised that most team members don't answer this question. In school we learn to do everything on our own. (Working together in school is called cheating.) In the IT industry, asking for help is even worse; I think this comes because we feel we are paid because we are smart, intelligent. Somehow, we think that being smart means not asking for help. Well, I've got news for you. That is not true. Asking for help is really a sign of strength."

Allison: "That could be true, but I have been working with a guy that was a total idiot and he kept asking for help, and did not learn anything."

Yves agreed. "I guess that 'I do not want to ask for help, as I don't want to look like this guy' is another reason why people don't ask for help. Guess what, that guy is a real exception, a lot more people make the opposite mistake. The typical situation being a male driver preferring not to ask for help and continuing to drive even when he does not know where he is. Rachel Davies told me recently she would also drive for 10 minutes before asking for help."

"So it's not exclusive behavior for you men?" Allison said with a wink. "Did she also say why?"

Yves answered, "Yes, for her it was not so much that she thinks she will look foolish asking for help. However she does not want to bother people if she can work it out herself. She also added that sometimes it takes a while to realize she got lost. Just as it might take people two or three days to ask for help in their team. Recently Dave Nicolette blogged about a similar experience with some very mature agile coaches that forgot to ask for help during the first certified developer Scrum course."

Allison: "Ok, then when should I ask for help?"

Yves laughed. "Whenever you feel blocked."

Allison: "What if I don't know where I need help with?"

Yves: "Especially then you should ask for help. It feels scary, but the most powerful instances of help I have received, are those when I had no idea I needed help but still asked for it."

Allison: "The previous patterns have a strange format. What about ask for help?"

Yves: "Use 'will you' instead of 'can you'."

Allison interrupts him. "The managers trick."

Yves reacts surprised. "Please say more."

Allison continues. "My first manager never asked 'will you do this'; he always asked 'can you do this', because he knew people would quickly say 'yes'. Of course they can, even when they don't want to."

Yves added to that, "Yes, but that does not show much respect for people. For me, respect for people is one of the core agile values."

Allison wonders, "What if I don't want to help?"

"Then you say 'no'." Yves adds, "An ask for help is only valid, if the other party has the right to say 'no'. Without a possible 'no', a 'yes' has no value." Yves proceeds. "When you know that people will ask for help when they need it, it also removes the urge to rescue people. When I was young, I had a hard time asking for help. I learned that behavior because in my environment people rescued other people before they could ask for help. Sometimes agile coaches think that rescuing is the same as helping. It's not. And it teaches people not to ask for help. So basically you make them more powerless. During my first years as a parent, it was hard for me to know the difference between helping a helpless child and teaching them to ask for help. (Which is helping them in the long run.)"

While Allison - who had a ten and a seven year old - grabs her notebook, to jot down a few sentences about this, Yves moves on to the next pattern. "Imagine the next situation, you are in a meeting and after I said something, someone else takes over and says something like:

> *What Yves says, has a lot of value, remember last year when we released supergizmo 364, customer babelstone had similar situation as what Yves describes. And on top of that, we should not forget that our company goal was to optimize the ROI of our department. And by consequence deploying our system that does what Yves described, we do precisely that. So I propose that we consider that option."*

Allison gazes at Yves, confused by the unclear conversation. Yves, amused by her reaction, clarifies, "Let me translate this:

> *BlahBlahBlah, I would like you to hear my voice, as for me it is important that you think I am important. And blahblahblah.... I agree with everything that was said."*

Yves looked at Allison and asked, "Is this a pattern (or better an anti-pattern) you recognize?"

She nodded.

Yves continued. "Imagine you have a meeting with 10 people. Five of these people each spend five minutes talking like this (usually in a group, once a person talks like this, a lot more people adopt the same behavior, so it's probably more then five)."

Allison calculated, "That is 25 minutes for 10 people."

That is what Yves wanted her to realize. "Now imagine that instead of this, you quickly make decisions. Would you not like this?" He did not wait for her to answer his rhetorical question. "**Decider** offers this possibility. Let me give you an example of a Decider:

> *Christophe: I propose that we publish our French customer documentation to our website* at 12:00.
>
> *1, 2,3*
>
> *Everyone shows his or her hand*
>
> *Christophe: thumbs up*
>
> *Gino: thumbs up*
>
> *Sylvia: flat hand*
>
> *Rachel: thumbs up*
>
> *Alistair: thumbs up*
>
> *Linda: thumbs down*
>
> *Christophe looks at the group and focuses on Linda: Linda what would it take to get you in?*
>
> *Linda: If want to be sure we have enough time to remove the spelling mistakes Bernard found this morning.*
>
> *Christophe does an eye-check with everyone in the team and says: I'll change my proposal to 12:30. Proposal accepted.*

Allison, will you tell me what happened?"

She accepts the challenge. "Christophe made a proposal. I'm not sure - was this after a long conversation?"

"It could be, but in general, everyone makes proposals as soon as possible."

Allison asks for more clarification. "So then he counts to three. Why did he do that?"

Yves explains, "The reason we do this, is to make sure that everyone votes at the same time. It might sound silly but if you don't, people look at each other and will vote what other members vote."

Allison continued. "The voting had three outcomes, will you explain them?

Yves: "Yes, I will. Thumbs up: I support your idea and I will do everything to make this a success. Flat hand: I don't have enough information to support your idea. Thumbs down: I can't support the idea like it exists now. It needs some changes. If the proposer thinks that the change is too different from his proposal, he says the proposal is dead (not accepted). The same is true if you have too many flat hands (you need enough people that support the idea).

Allison, puzzled, asked, "When I have a better idea, what should I vote?"

Yves smiled. "When you think you have a better idea, you vote flat hand and after the first proposal is accepted, then you propose your better idea. This has..."

Allison interrupts him, "I want to be sure I understand, I propose my better idea after the first proposal is accepted, not when it's implemented?"

Yves agreed, "exactly! This has the advantage that proposals get accepted and we create a momentum. Even if your new proposal is not accepted, at least we have the first proposal that is accepted." This creates a bias towards action. If you vote thumbs down and you propose your own idea, we might end up with pingpong proposals and nothing gets accepted.

Working in this way, means that a team does not block itself from taking decisions even if they think they can come up with a better idea later." It's ok to have better ideas later. We will do a new decider and we'll switch to the new idea (when people find it better). In 'lean' we want to defer decisions to the last responsible moment. The problem I see with this is that for most teams it's hard to find that last responsible moment. On top of that, I see teams block them selves from taking decisions and actions by doing this. So in reality they take no decision, which is worse and which is also waste, waste of time. Using decider to quickly take actions with the possibility to re-decide later (when we have more information) is for me in sync with take decision at the last responsible moment."

Allison: "What if the decision you have to take is a difficult or expensive one to change once it's made?" Yves whistles, "That is a great question."

Yves turned to Mary Poppendieck who was listening into the conversation, "will you answer this one Mary?"

Mary, "Yes, I will. The goal in Software Development is first of all to make all decision changeable. (Decoupled architecture, test harnesses etc.) But when a decision is permanent, it cannot be easily changed, and when that decision is also critical to success, you do not want to make it early, instead you want the team members to decide when they will decide. That means the teams comes to an agreement on when the last responsible moment is, and then when that moment comes, they make the critical, difficult-to-change decision."

Yves relaxed. "I could not have said that better. Remember that first meeting with these same 10 people, imagine that I would have proposed my idea and that we had voted on this. We would have won 25 minutes for 10 people. That is 250 minutes."

Allison: "That is half a working day."

Yves: "Yes, Imagine winning half a day in every meeting. How do you feel about that?"

Allison: "That would make me feel a lot better about meetings. What about aligning people in meeting? And getting to know each other in these meetings."

Yves: "I agree alignment is important, for that you should use the alignment protocol. Actually having your team taking decisions helps them much more for understanding the purpose of the team. For getting to know each other, the earlier check in is a lot better. I had teams take five or six decisions in less then 10 minutes. And then they go off and continue making software. Imagine what this does to your productivity. (And moral, as most software developers hate unproductive meetings.)"

Allison asks, "Meetings are the cornerstone in my company, can you give some more advice about optimizing our meetings?"

Yves: "I'm about to have a meeting with Christophe and Bénédicte to organize the next BootCamp. If you want, you can observe the meeting."

The next passages are the notes that Allison took while observing the meeting.

Christophe starts the meeting:

Christophe: Let's check in. I'm glad that we have this meeting to decide about a next European BootCamp; I'm sad we do this over the phone, I'm glad I hear your voice. I'm in

Bénédicte, Yves: Welcome.

Yves: I'm glad we talk, I'm glad to meet Bénédicte, I'm afraid to have such an important meeting over the phone, I'm glad to know that the core will help us, I'm glad Christophe keeps pushing me for a next BootCamp. I'm glad, sad, afraid my priorities were elsewhere before. I'm in

Christophe, Bénedicte: Welcome

Bénédicte: I'm glad I finally meet you, I'm glad I see things moving. I'm in

Yves, Christophe: Welcome

Christophe: We disclose what we want. I want to know when the next European BootCamp will take place.

Yves: I want to understand how many people are really interested in a next European BootCamp.

Bénédicte: I want to know if my idea for a place for a BootCamp is a good idea.

Christophe: Alignment check, where are you compared to your goal: I'm at 7 out of 10.

Yves: I'm at 6 out of 10

Bénédicte: I'm at 2 out of 10

Christophe: Bénédicte, as you are the lowest, you start.

Bénédicte: I understood that almost all European BootCamp took place in Koningsteen Belgium.

Yves: Yes.

Bénédicte: Is there an option to do that in another place?

Yves: Yes, there is. We keep working with Koningsteen as it is a great environment where we know we can do everything we want. We selected Belgium as it is rather central in Europe. Will you tell me what other place you had in mind?

Bénédicte: Yes, I was thinking about Domain de Soulignac en France.

Yves: Will you tell me some more about the Domain?

Bénédicte: Yes. At the heart of the Limousin, ...

Yves: That looks like a great place. I would love to try this location.

Christophe: Alignment check; I'm at 5

Bénédicte: I was at 10. I have a new want: I now want the same as Christophe wants. I'm at 5.

Yves: Ok, let' see when we can do this. Do you want English and French speaking trainers?

Bénédicte: Yes, some of the people I know can speak English but they would prefer a French speaking trainer.

Yves: Then the first available option is September. What week in September can we do this?

Bénédicte: I would prefer the second week of September.

Christophe, Yves: Yes, that is possible.

Yves: Cool, we do the next BootCamp from the 5 September 2010 till the 10th.

Alignment check: Bénédicte: 10, Christophe 10, Yves 6.

Bénédicte: I have what I want, I'm checking out.

Christophe: Yves, you wanted to know how many people where interested.

Yves: Yes.

Christophe: Well, we have three people from my company, plus me, we have Bénédicte and five of her friends. How many people is the minimum you need for a BootCamp?

Yves: Seven is the minimum number to learn about group dynamics in one week. I have seen it work with less people, but that where people that were already a tight team before.

Christophe: We have nine people, that should be enough?

Yves: Yes. And I know some more to contact. Looks like we have a new BootCamp ...

Alignment check: Yves 10, Christophe 10

Yves: I have what I want, I'm checking out.

Christophe: Yes, so do I, thank you.

Yves: "You heard a meeting where people used the **meeting protocol**. Will you tell me what you saw?"

Allison: "Christophe started the meeting, by doing a group checkin. Then everyone stated what she wanted out of the meeting. With a score from 1 till 10. Nobody talked about time constrained. Is that the moment to talk about?"

Yves: "Yes, after the check in, that is the right moment. It is one of the improvements important if you do the meeting over phone or chat. People tend to not do it when face to face as all these meetings are on our calendars."

Allison interrupts, "so why aren't you all at 0 at the beginning?"

Yves, "oh , good observation, I was not at 0 because I had been talking to other people, so I already had an idea how many people where interested, does this answers your questions?"

Allison, "yes it does. Then you started by the person with the lowest score. Why did you ask everyone what he wants? If it is my meeting, I want to set an agenda!"

Yves: "That is true and we want to make sure we address the most important issues concerning this topic. If everyone has to organize his own meeting to address an issue, we now have 10 meetings. (Did I mention already that we schedule our meetings for at least an hour?) The meeting protocol offers us a way to have 10 productive meetings instead of one. And yes people are smart enough to only bring topics related to the goal of the meeting."

Allison: "What if I have no goal for this meeting?"

Yves: "Then why are you in this meeting?"

Allison: "Well, I don't know if my input is needed to solve an issue in this meeting."

Yves: "With the meeting protocol, you will know at the beginning if you are needed or not. And if not, you leave."

Allison: "Yes, I noticed that Bénédicte left halfway the meeting, once she had what she wanted."

Yves: "You can stay if you think someone needs your input in the meeting. (If you always stay and nobody actually needs you, your goal probably is: 'I want to feel needed by the team'. That goal will never be reached.)"

Allison remarks, "haha, this conversation is as funny as I hoped it would be. Earlier you talked about **Personal Alignment** will you tell me what that is?"

Yves closed his eyes, "No, I will not do that now. I'm too tired and I need some sleep. Why don't we get together tomorrow and continue our conversation?"

Allison: "I propose we see each other at 10 am here in the bar, 1,2,3".

Yves showed thumbs up and said, "That is a deal."

Allison laughs while she said, "Protocol check. You are not supposed to talk during a decider."

The conversation of the next day will appear in the next issue of Methods and Tools.

Allison and Jeffrey are fictive persons. This conversation is based on talks that Yves had at agile conferences the last five years. The meeting between Yves, Christophe, Bénédicte took place in 1st quarter 2010 over the phone. Although the transcript was edited, this was how this 30 minutes conversation went.

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

This article was written with the help from Jim & Michele McCarthy, Els Ryssen, Paul Reeves, Christopher Thibaut, Adam Feuer, Ralph Miarka, Mary Poppendieck, Gino Marckx, Alistair Cockburn, Philip Almey, Lilian Nijboer, Esther Derby. A big kudo's to Emmanuel Gaillot who initiated the conversational style. Another thank you to google docs that made it possible for this international team to have +3000 revisions.

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

[1] Feel free to mail me your reasons why check in or any other protocol could not work.

eValid

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

eValid, is a testing tool suite built into an IE browser. eValid performs every function needed for detailed Web Site static and dynamic testing, regression testing, QA/Validation, page timing and tuning, transaction monitoring, realistic and scalable server loading

Web Site: http://www.e-valid.com
Version Tested: eValid V9 Build 301, tested during May-June 2010 on Windows XP Home Edition SP3
System Requirements: eValid V9 relies on certain properties of the IE DLLs that are only available in IE 5.50 and later versions until IE 9.0. Similarly, certain features of the technology require use of Windows operating system features that are only present in Windows 2000/SP4, Windows XP, Windows Vista, Windows 7 and Windows Server 2008. At launch eValid will provide an advisory notice (popup) in case the minimum required operating capabilities are not present.
License & Pricing: Commercial
Prices are visible http://www.e-valid.com/Products/bundle.pricelist.9.html
Support: User mailing list, forum and blog

Installation

The installation runs smoothly through an InstallShield process. After registering the license key(s) (Help/Manage License), you can start working with the product.

Documentation

The documentation (user manuals, tutorials, movies, demos, etc) is comprehensive and fully accessible from the tool. To have a first glance at the tool features, I will recommend looking at these online resources:
Tutorials: http://www.e-valid.com/Products/Training.9/index.html
Videos: http://www.e-valid.com/Products/Documentation.9/Movies/inventory.html

Configuration

eValid has a "Settings" panel that allows modifying general options and those dedicated to a specific feature (record, playback, site analysis). The user manual gives a detailed explanation of each configuration field. The default configuration allows working well, but there are many parameters that could change your assessment of the results, especially in the site analysis area.

Features

Architecturally, eValid is an "overload" of the IE browser. It adds functions to the browser by incorporating the entire browser as a support engine.

* Functional Testing

You can create a script simply recording your actions on the web page and adding validation action that are proposed by the tool. By default the settings are for "real time" recording, which means that the "wait" time between two actions will be automatically recorded by the tool. This can be changed in the settings. An existing script can be modified, inserting new actions at a

particular point. The recorded script can be also manually edited, which allow to remove certain items, modified input, waiting times and validations parameters. Validations can be made at the document level (title, byte size, last modified date), on a page element (text, image, table cell) or on a page area. In fact the tool can validate and synchronize any DOM (Document Object Model) property anywhere on the current page.

```
Script File - [eValid MT Subscription.evs]
File  Edit  Insert
 1 #
 2 # Recording by eValid™ V9 Build #301, Build Date: (Apr 19 2010)
 3 # Copyright © 2010 by Software Research, Inc.
 4 # Recording made on: Microsoft Windows XP Service Pack 3 (IE 8.0.6001.18702), HostName "Martinig"
 5 # Recording started at: <2010/06/10 15:33:20>
 6 #
 7
 8 ProjectID "Project"
 9 GroupID "Group"
10 TestID "eValid MT Subscription"
11 LogID "AUTO"
12
13
14 ScreenSize 1280 1024
15 FontSize 1
16 DeleteCache
17 InitLink "http://www.methodsandtools.com/"
18 ValidateSelectedImage 0 88 "http://www.methodsandtools.com/images/subscr" \
19         "ibe.gif" -1 80 25 "" ""
20 Wait 12266
21 FollowLink 0 87 "" "http://www.methodsandtools.com/forms/submt.php" ""
22 ValidateDocumentTitle 0 "Methods & Tools - Subscription Form" ""
23 Wait 15312
24 InputValue 0 68 "TEXT" "" "Name" "Name" "" ""
25 Wait 7203
26 InputValue 0 73 "TEXT" "" "Email" "email@methodsandtools.com" "" ""
27 Wait 7469
28 InputValue 0 78 "SELECT-ONE" "" "Country" "Switzerland" "178" ""
29 Wait 10672
30 InputValue 0 291 "TEXT" "" "Copies" "10" "" ""
31 Wait 3516
32 InputValue 0 315 "SELECT-ONE" "" "Size" "51 to 100 people" "2" ""
33 Wait 4375
34 InputValue 0 332 "CHECKBOX" "" "Mini" "ON" "TRUE" ""
35 Wait 3265
36 InputValue 0 338 "CHECKBOX" "" "Web" "ON" "TRUE" ""
37 Wait 2110
38 InputValue 0 334 "CHECKBOX" "" "Mainframes" "ON" "TRUE" ""
39 Wait 2781
40 SubmitClick 0 349 "" "Submit" "Submit" "" NAV
41 Wait 11719
42 ValidateSelectedText 0 56 0 "Your subscription request has been sent to " \
43         "our administrative team and will be processed as soon as possible" ""
44 # Recording stopped at:  <2010/06/10 15:34:47>
Script: C:\Program Files\Software Research\eValid\Program\Project\Group\eValid MT Subscription.evs    Ln 20
```

Figure 1. Functional test script created through navigation with element validation

"Point and click" is the first and simplest way to create functional tests with eValid, but the tool support "structural/algorithmic" testing and full programmatic testing. Thus eValid can automatically synchronize with AJAX applications using the built-in DOM interrogation capability. You can also parameterize scripts, manage control flow as they execute. Data from external files can be fed into scripts.

*** Load Testing**

The load testing function in eValid uses the same script repository than the functional testing part. You can therefore either reuse an existing script or create a dedicated item for load testing. You can specify the number of times that you want to run the script and the delay between each run. This playback multiplier allows increasing or decreasing the speed or repetition. The functional scripts and this additional information allow you to create load-testing scripts. The estimated number of users is dependent of these parameters. A script repeated 5 times that is complete in 20% of the normal time is the equivalent of a 25 users load. The load script runs each functional script in a separate eValid sub-browser without cache to assure that each page is

actually downloaded. Information used for the test (like user names) can be changed for each test. You can obtain 100 Browser Users per desktop session and eValid is able to run over 1,000 using multiple desktop sessions on a single machine.

My test scenario runs twice a test that just loads the home page, then twice a test that goes to a poll page, displays results and verifies the presence of a text on the result page, then again once the load home page test.

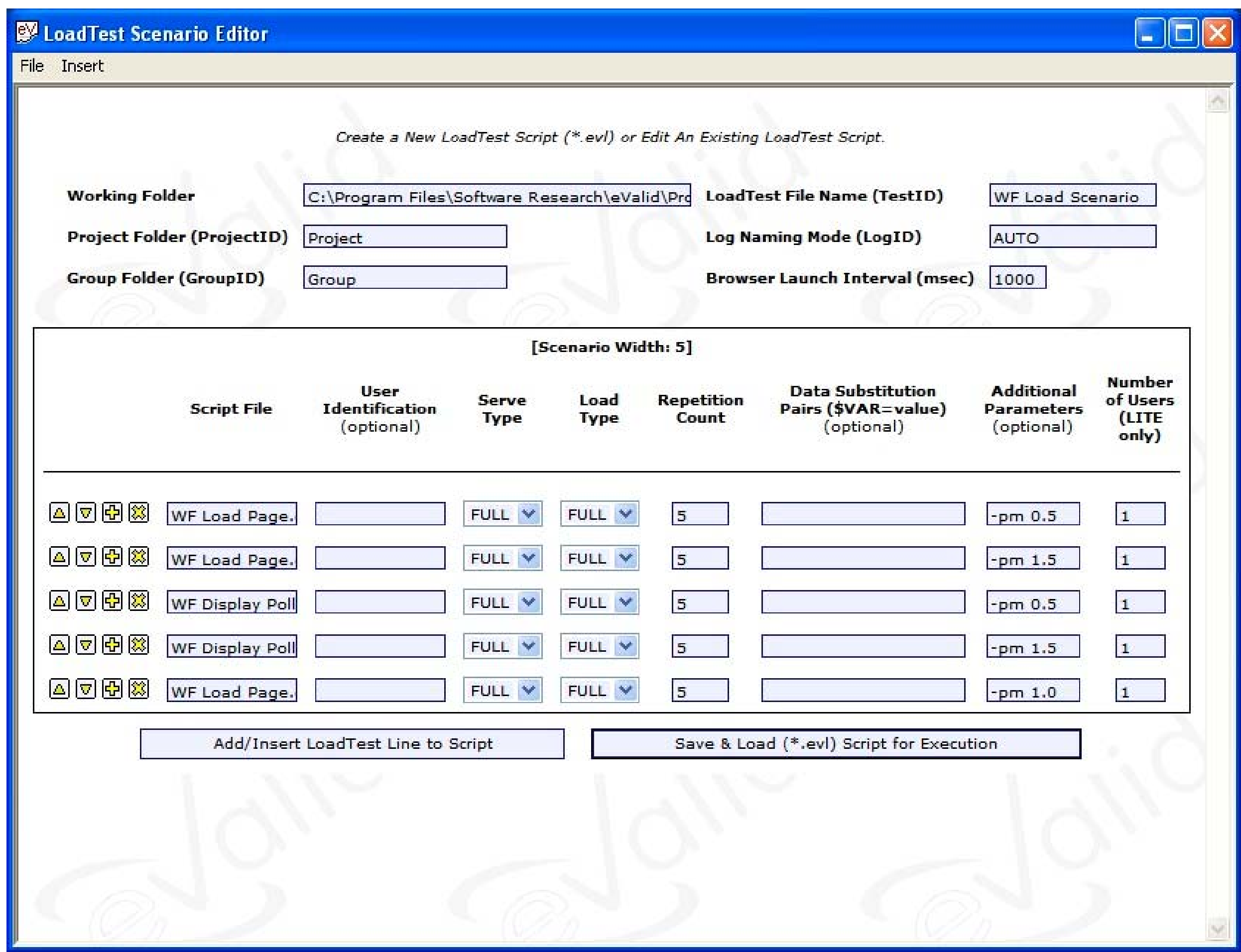
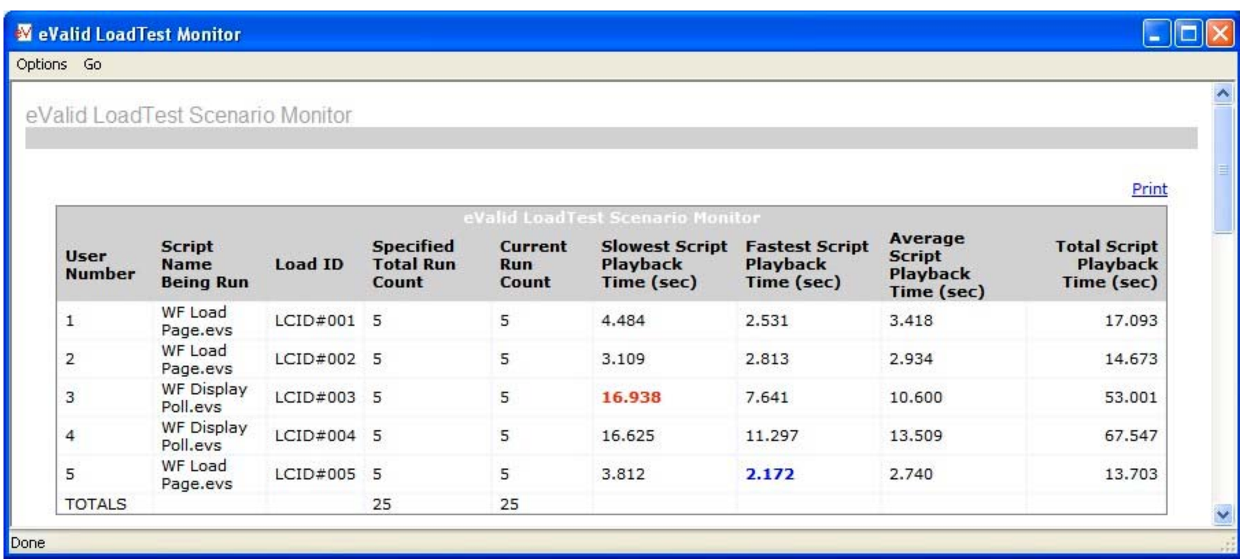


Figure 2. Load testing scenario



eValid LoadTest Monitor

Options Go

eValid LoadTest Scenario Monitor

Print

eValid LoadTest Scenario Monitor

User Number	Script Name Being Run	Load ID	Specified Total Run Count	Current Run Count	Slowest Script Playback Time (sec)	Fastest Script Playback Time (sec)	Average Script Playback Time (sec)	Total Script Playback Time (sec)
1	WF Load Page.evs	LCID#001	5	5	4.484	2.531	3.418	17.093
2	WF Load Page.evs	LCID#002	5	5	3.109	2.813	2.934	14.673
3	WF Display Poll.evs	LCID#003	5	5	16.938	7.641	10.600	53.001
4	WF Display Poll.evs	LCID#004	5	5	16.625	11.297	13.509	67.547
5	WF Load Page.evs	LCID#005	5	5	3.812	2.172	2.740	13.703
TOTALS			25	25				

Done

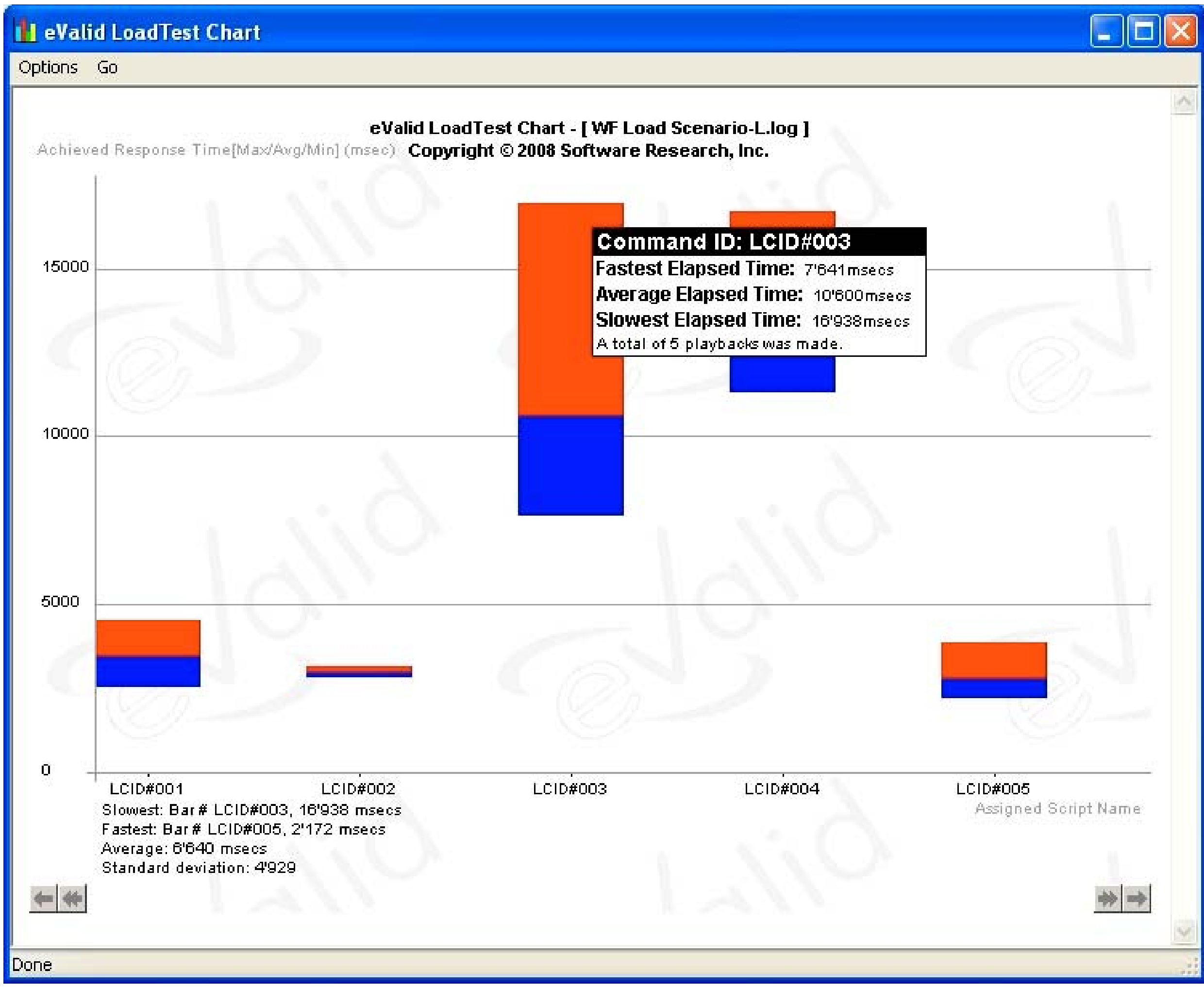


Figure 3. Load testing results

*** Web Site Analysis**

You can launch the web site analysis either in automatic or controlled (interactive) mode. Many setting screens that allow you to control the analysis (depth of pages, links to follow or exclude, etc), manage the degree of analysis and your own definition of what is an old, large or small page. It took 30 minutes to analyze automatically the complete Methods & Tools web site with the default analysis settings. For a simple web site like waterfallalliance.org, it takes 20 seconds. A 3D-SiteMap shows inter-relationships of all scanned pages. Reports are generated in real time.

*** Additional Functions**

The eValid suite contains a dedicated test suite management engine. This system organizes a suite of eValid tests into groups and complete projects when you locate test scripts in the same folder. It launches and analyzes eValid test results and provides a compact PASS/FAIL report summary. You can use eValid as an AJAX monitoring agent with its synchronization capabilities.

eValid offers also a test data generator. All test results are usually stored in text files, but you can also use a MySQL access to store them. The tool proposes also a batch interface (command line operation) and an available C++ interfacer.

Conclusion

eValid is a software testing tool suite that allows testing both the functionality and the performance of a web site. You can also do a site analysis. The tool is quite intuitive and the training level needed for a first usage not important, watching the short movies available that explain the basic functions. This allows people without technical experience to build and run their own tests. It is nevertheless a powerful tool that goes far beyond a simple "point and click" technology and you can spend time adjusting the settings and the scripts to your technical needs for more sophisticated testing, especially for AJAX applications.

Hudson – Your Escape from "Integration Hell"

Dr. Simon Wiest, Dr. Wiest Software Engineering, http://www.simonwiest.de

Hudson is a popular web-based continuous integration server, written in Java. Hudson is used in 17.000+ server installations worldwide in small, medium and large companies alike, including eBay, Hewlett-Packard, MySQL, JBoss, Xerox, Yahoo, LinkedIn, or Goldman-Sachs. It allows you to automate your software build chain, e.g. monitoring changes in version control systems, triggering new builds, testing artifacts, sending notifications, deploying to production servers, and much more. Hudson is liked for its ease of use and broad extensibility via 250+ plugins.

Web Site: http://hudson-ci.org (see http://hudson.glassfish.org for a live installation)
Version Tested: Hudson V1.358 (Mai 2010)
License & Pricing: Open Source (MIT License)
Support: User and developer mailing lists

Overview

"*Continuous Integration (CI) is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.*" (Martin Fowler, http://martinfowler.com/articles/continuousIntegration.html)

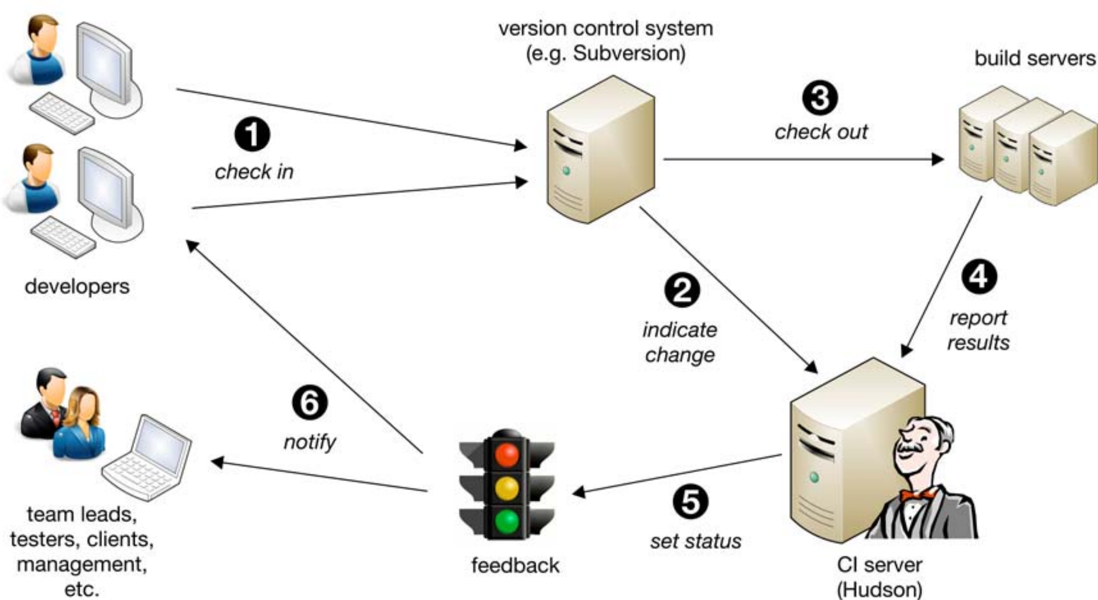


Figure 1: A typical continuous integration cycle

Hudson is a CI server that allows you to implement that approach (Figure 1): Developers will check in new code into your version control system (1). Hudson polls this system regularly for changes, e.g. every minute (2). If changes are indicated, the new code is checked out to build severs (3) and a new build of your project is executed. The results are reported back to Hudson (4) that will set the new status of the project (5): stable, unstable or failed. Finally notifications are sent to developers and other stakeholders (6), which will either fix any problems found or proceed implementing new features. This complete cycle typically does not exceed 15 minutes.

Installation

Download the latest release from http://hudson-ci.org/latest/hudson.war. Start Hudson in the command line using the following command: `java -jar hudson.war`. Finally, open http://localhost:8080 in your web browser. You should now see the Hudson Dashboard (like Figure 2, but for obvious reasons without any projects yet). You are now ready to run. If port 8080 is already used, you can choose another port by starting Hudson with a command like: `java -jar hudson.war --httpPort=8888`. Hudson can also be deployed in application servers like Tomcat, WebSphere or JBoss.

The Hudson project publishes new releases roughly every week. Hudson will inform you of updates available on its Administration page (Hudson→Manage Hudson). Plugins are installed via the built-in Plugin Manager (Hudson→Manage Hudson→Manage Plugins).

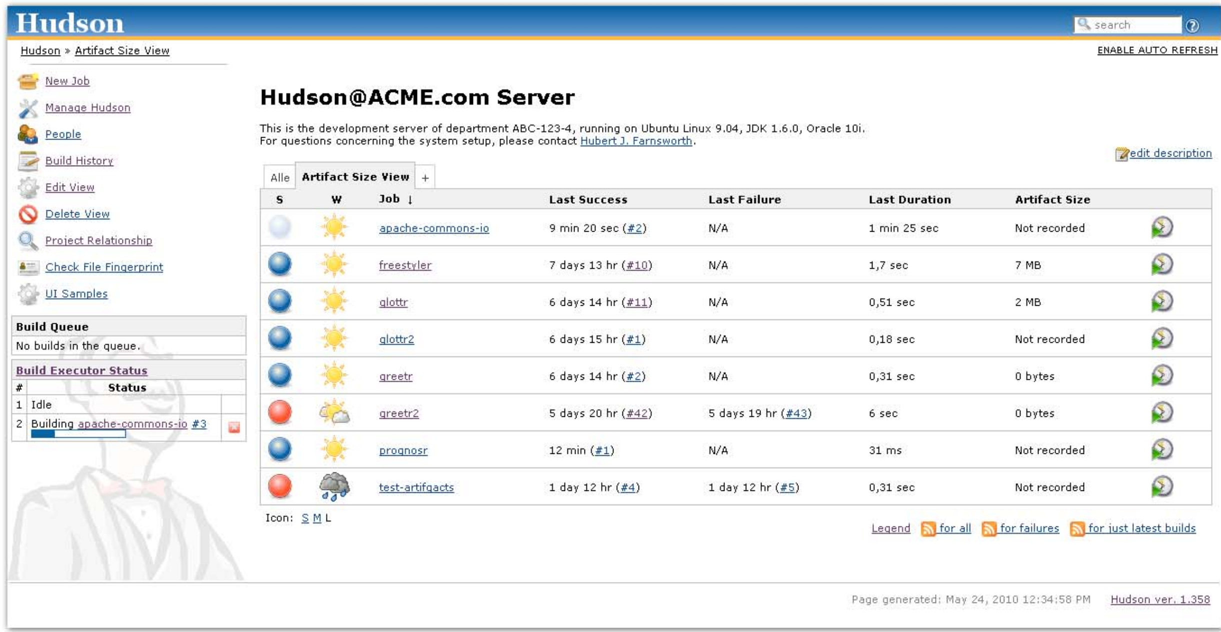


Figure 2: A typical Hudson Dashboard

Documentation

Very useful documentation is placed right into the application as context-sensitive help. There's also a wiki on the project website and two active mailing lists for users and developers (some 18,000 postings in 2009). Two books on Hudson are currently in progress, "CI with Hudson" by J.F. Smart and "CI mit Hudson" (in German) by S. Wiest.

Distributed Architecture

Building and testing software is a rather resource intensive process. *Continuously* building software obviously requires even more computing power. Hudson can distribute your builds to a grid of build servers, the "slave nodes". This speeds builds up, but also enables you for example to test software on multiple slave nodes running different operating systems, databases, application servers, web browsers, service packs, etc. Slave nodes can be run on "real hardware" under your desktop, in your data center or on virtualized systems, e.g. on a VMWare host or in the Amazon EC2 cloud. Hudson will take care of monitoring the nodes, distributing the work and collecting the results, once the builds have been completed on the slave nodes.

Plugins

As software build chains always differ from team to team, a good CI server must demonstrate "infrastructure chameleon" capabilities. Hudson is extensible by 250+ plugins, offering support for alternative build tools (e.g. msbuild, gradle, rake), version control systems (e.g. CVS, Git, Perforce, Mercurial), notification channels (e.g. rich email, jabber, twitter, eXtreme feedback devices), virtualization and cloud computing (VMWare, Amazon EC2), user directories (e.g. LDAP, ActiveDirectory), issue trackers (e.g. JIRA, Bugzilla, Mantis), and much more. At current pace, the Hudson community releases 1–2 new plugins and 3–5 updates of existing ones per week. The source code of these plugins is available to serve as a starting point for own customizations. In fact, many Hudson "heavy users" came for a shrink-wrapped, ready-to-use product, but stayed for a highly customizable build automation platform.

Building a "Test Drive" Project

You understand Hudson best by seeing it in live action. Thus, for a simple five-step test drive of Hudson, you could build any Open Source project that hosts its source code publicly accessible. We will use Apache Commons IO in the following example:

1. First go to Hudson→Manage Hudson→Configure System: In Section "JDK" add a new Java Development Kit. In section "Maven" add a new Maven installation. Click on "Save" at the bottom of the page.
2. Then go to Hudson→New Job enter "apache-commons-io" as job name and choose "Build a maven2 project". Click on "OK" to proceed with the job configuration page.
3. On the job configuration page, in section "Source Code Management", choose "Subversion" and enter "http://svn.apache.org/repos/asf/commons/proper/io/trunk" in field "Repository URL". Click on "Save" at the bottom of the page.
4. Click on the "Build Now" icon in the left sidebar. A new build will start, checking out the source code, then compiling and testing it. You can follow the progress by peeking at the live output of the build: Click on the progress bar on the left hand side to open the console view.
5. Once the build has been finished, return to the Hudson dashboard page by clicking on "Hudson" in the top left corner. If the build succeeded, you will see a blue ball icon. If any problem occurred, the ball will be yellow or even red. Note how you can browse the test results interactively by clicking on the project name and then on "Latest Test Results".

You have now configured your Hudson server, added your first job and built it. In a production instance, builds would be triggered automatically (by changes in the version control system or by a timer) - but the basic scheme would be the same.

Conclusion

Continuous Integration is a software development practice that most teams cannot think of living without, once it has been established. With Hudson, the necessary tooling is fast to install, easy to maintain and scalable for future extensions.

A final note: While Hudson is implemented in Java and supports mainstream Java build chains out of the box (CVS/Subversion, Ant/Maven, JUnit/TestNG), it is not restricted to the Java world: Many companies use Hudson to automate their software development efforts in C/C++, C#, PHP, Groovy, Scala, and many other languages.

FitNesse: A Tester's Perspective

Lisa Crispin, http://lisacrispin.com

FitNesse is a test framework that allows testers, developers and customers to collaborate to create test cases on a wiki. Teams can take examples of desired software behavior and turn them into automated tests, integrated with narrative requirements documentation. FitNesse runs its own wiki web server. It's possible to test applications written in Java, .Net, Ruby, Python, C and PHP using FitNesse.

My team has used FitNesse since 2004. This review is based on our experiences. I hope it will help you decide whether you'd like to try out FitNesse for your own test automation.

Web Site: http://www.fitnesse.org
Version tested: 20100308
License & Pricing: Open Source
Support: Yahoo group mailing list

Installation

Ease of installation is one of the strengths of FitNesse. You can download FitNesse via http://fitnesse.org. Installation is simple. For a Java installation, type java –jar fitnesse.jar. This will put all the necessary files in place. To start the server, type java –jar fitnesse.jar –p:xxxx where xxxx is your desired port number. Now you can access FitNesse via your browser with http://<hostname>:xxxx/FrontPage where hostname is your localhost or the server where you just installed FitNesse and xxxx is the port number you used. Yes, it really is that easy to get it installed.

You can write and execute FitNesse tests via the browser. Tests and suites may also be run from the command line with a RESTful syntax, or from a JUnit test.

Configuration

FitNesse comes with its own version control and test history functionality, so it can be used stand-alone. Everyone on the team can use a single FitNesse server, or each team member can import tests into their local environment, work on them, and export changes and new tests back to the main server.

You may also choose to integrate FitNesse with your existing version control, so that tests can be tagged along with the code that they test. Each team member can have their own independent FitNesse environment, and check new and updated tests into the source code control system. My team checks our tests into Subversion, and we run our FitNesse regression suites from our Hudson version control so that we get regular, speedy feedback and take advantage of Hudson's test reporting features.

FitNesse tests may be run on top of either Fit, using additional fixtures in FitLibrary, or Slim. Both currently come with FitNesse, although Fit may not be bundled with FitNesse in future versions and you'll have to download it separately. Each has its own set of test fixtures built in, with a lot of similarity between the two sets of fixtures. For example, the ScriptTable in Slim and the DoFixture in FitLibrary can both be used to create flow tests that are more readable to the business. You can use either or both, and your team will build on these built-in fixtures to automate your tests with your production code.

Documentation

The http://fitnesse.org site has an up-to-date user guide and many tutorials and screencasts to help you get started with FitNesse. There are two excellent books to help your team design effective FitNesse tests: *FIT for Developing Software* by Ward Cunningham and Rick Mugridge, and *Test-Driven .NET Development with FitNesse* by Gojko Adzic. Other websites, such as http://gojko.net and Brett Schuchert http://schuchert.wikispaces.com/FitNesse have helpful tutorials, videos and screencasts.

FitNesse has an active developer and user community, which is vital for an open source tool. If you have questions or issues, you can search the archives on the Yahoo mailing list http://tech.groups.yahoo.com/group/fitnesse/, and post your questions if you don't find an answer. You can rely on getting help from the community.

Learning Curve

Learning to write test cases in the various formats such as ColumnFixture is fairly easy. Learning how to design them well, for efficiency and ease of maintenance, takes more time. If you don't have object-oriented design skills, pair with a programmer or other team member who does. Apply good code design practices such as 'Don't Repeat Yourself' to your FitNesse tests.

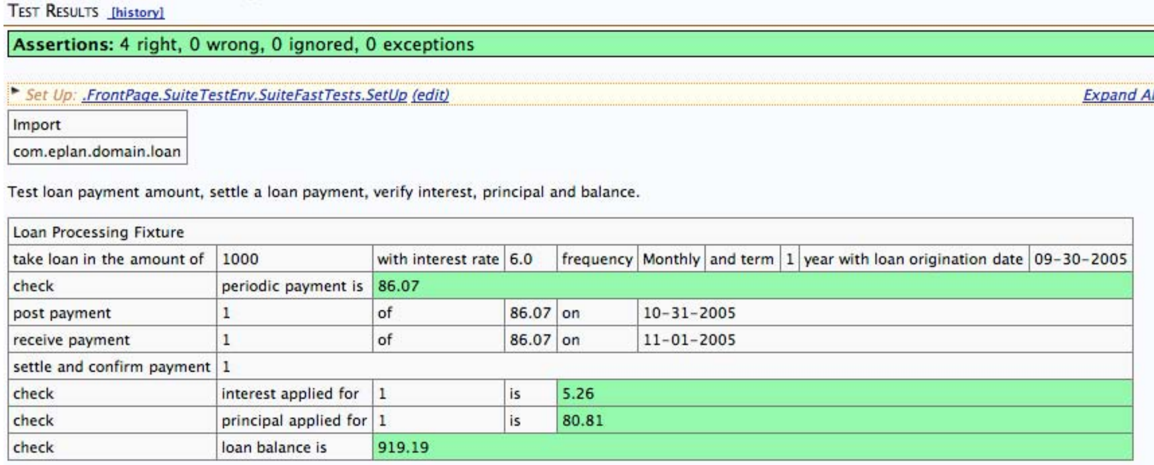
If you're not a programmer, you'll also have to collaborate with a programmer who can write the necessary fixtures which take the test inputs, pass them to the production code, and send the results back to FitNesse for comparison with expected results. Writing fixtures is fairly straightforward for programmers, as they will do this in the same language in which they write production code. For our team, it usually takes two to four hours to write a new FitNesse fixture. Occasionally, it can be tricky to come up with a fixture for a complex testing scenario.

Examples

There's a wide variety of formats for FitNesse tests, both tabular and scenario style. Our tests mainly extend the ColumnFixture. Here's a simple example of from the FitNesse user guide.

eg.Division		
numerator	denominator	quotient?
10	2	5.0
12.6	3	4.2
22	7	~=3.14
9	3	<5
11	2	4<_<6
100	4	33

Sometimes a tabular format doesn't reflect the business example well, so there are other options. Here is an example from our own tests, using the DoFixture. The test takes out a loan with a given amount, interest rate, payment frequency, term and start date. A payment is received and processed, then the test checks the interest and principal applied and the remaining balance. This mimics the actual business flow. Slim scenario tables are similar to the Fit DoFixture.



Test Results [history]

Assertions: 4 right, 0 wrong, 0 ignored, 0 exceptions

Set Up: .FrontPage.SuiteTestEnv.SuiteFastTests.SetUp (edit) Expand Al

Import
com.eplan.domain.loan

Test loan payment amount, settle a loan payment, verify interest, principal and balance.

Loan Processing Fixture									
take loan in the amount of	1000	with interest rate	6.0	frequency	Monthly	and term	1	year with loan origination date	09-30-2005
check	periodic payment is	86.07							
post payment	1	of	86.07	on	10-31-2005				
receive payment	1	of	86.07	on	11-01-2005				
settle and confirm payment	1								
check	interest applied for	1	is	5.26					
check	principal applied for	1	is	80.81					
check	loan balance is	919.19							

Well-Designed Tests

FitNesse provides features that enable you to design automated tests for ease of maintenance. The !include feature is one way you can extract duplicated test code and reuse it in several tests. Variables can also be used so that when you need to make a change, you can do it in one location, and not have to change multiple test pages. Here's an example.

We can define a variable whose contents are a test table:

```
!define LoanSetup (!|Loan Processing Fixture|
|take loan in the amount of|${loanAmount}| with interest
rate|${interestRate}|frequency|${frequency}| and term | ${term} | year with
loan origination date | ${date}|
|check| periodic payment is|${periodicPaymentAmount}|
)
```

We can use this variable in multiple tests, substituting the desired values for loan amount, interest rate and so on for each test. For example:

```
!define loanAmount (1000.00)
!define interestRate (6.0)
!define frequency (Monthly)
!define term (1)
!define date (09-31-2005)
!define periodicPaymentAmount (86.07)

${LoanSetup}
```

When we run the test, it is as readable as the test that doesn't use variables:

variable defined: loanAmount=1000.00
variable defined: interestRate=6.0
variable defined: frequency=Monthly
variable defined: term=1
variable defined: date=09-31-2005
variable defined: periodicPaymentAmount=86.07

Loan Processing Fixture									
take loan in the amount of	1000.00	with interest rate	6.0	frequency	Monthly	and term	1	year with loan origination date	09-31-2005
check	periodic payment is	86.07							

The Scenario Tables in Slim provide similar functionality to this technique.

Test automation projects often fail because tests are not designed for maintainability, and the overhead to keep the tests up to date becomes overwhelming. FitNesse's design-friendly features help prevent this problem.

Add-Ons

FitNesse provides a framework for using other test tools, such as GUI drivers. This allows greater flexibility, while taking advantage of the ease of writing tests in the Wiki and FitNesse's excellent result reporting capabilities. For example, you can drive Selenium (http://seleniumhq.org/) GUI tests from FitNesse test tables. A tool called Selenesse (http://github.com/marisaseal/selenesse) provides a bridge that makes this easy for both Java and .Net environments. SWAT (http://ulti-swat.wikispaces.com/) is another example of a GUI test driver that integrates well with FitNesse. Many teams build their own custom tools using FitNesse or Slim as a base.

Drawbacks

Open source tools are only as good as their developer and user communities. With any open source tool, there may not be a reliable schedule of releases with new features. FitNesse has generally enjoyed frequent releases and improvements. My team suffered some from a backward compatibility issued in a recent release. Overall, however, useful new features arrive regularly.

We sometimes run into issues where we set up FitNesse in a new environment, without any error message or clue as to what is wrong. For example, the tests often hang with no feedback if the classpath as defined in the !path variable is invalid. We've worked around these issues by trial and error, and assume it has something to do with the architecture, but it can be frustrating.

We had a hard time integrating FitNesse into our Hudson continuous integration and build process, but the user and development community helped us find a solution. There's now a Hudson plug-in for FitNesse. You may want to make sure FitNesse will work and play well with your CI before committing to using it long-term.

FitNesse is unique (as far as I know) in using a Wiki for creating and maintaining test cases, and the Wiki can also be used as a knowledgebase and repository for story and theme requirements. The fact that anyone can contribute to the wiki is both a plus and a minus. It's a great collaboration tool, but it can easily go out of control. My team has had trouble keeping our wiki organized well enough so that we can easily find information and tests when we need it. You'll need to budget time and perhaps have a technical writer help organize and maintain the FitNesse pages and hierarchy.

It can also be tedious to create and maintain test cases on the wiki, using the wiki markup. FitNesse provides the ability to import and export test cases to and from an Excel spreadsheet, which helps. However, an IDE plugin so that tests could be edited via Eclipse or other IDEs would be a big improvement.

Benefits

Our team found early on that writing FitNesse tests forced testers, programmers and customers collaborate more. We expected it to provide a great safety net of regression tests, which it does, but we were surprised to find the greatest benefit was the increased communication during the development of a story or feature. In the process of turning business examples into FitNesse tests, we discovered disconnects in our understanding of what a story should deliver. When this happens, we can immediately get testers, developers and customers together to clarify the desired system behavior. Writing code test-first meant that we didn't have many "bugs" in the traditional sense, but the development team often missed or misunderstood requirements. The

process of writing test cases in FitNesse helped us get quicker feedback and deliver the right business value.

Our FitNesse tests make superb documentation. They include both narrative about the functionality they're testing, and executable tests. They have to keep passing, so unlike the written documentation many teams have, we're forced to keep them up to date. When someone from customer support comes over to ask what results should come out from a given set of inputs in production, we don't get into a philosophical discussion over how the code works, or scratch our heads trying to remember. We can prove how the code works with a passing FitNesse test. Maybe this isn't how the business really wants the code to work, and in that case, they can write a story to change it. This saves lots of time, and makes us look smart.

The 'Drawbacks' section in this review is longer than the 'Benefits' section, but for our team, the benefits far outweigh the drawbacks. As I wrote this article, one of our FitNesse suites caught a regression bug 20 minutes after a code change was checked in. We'd have found the regression manually at some point, but it may have taken up to 24 hours. The fast feedback helps us maintain a sustainable pace and ensure we can deliver high quality code to production anytime.

Advertisement - Testing Television - Click on ad to reach advertiser web site

Testing Television

http://www.testingtv.com/

Testing Television is a directory of videos, interviews and tutorials focused on all software testing and software quality assurance related activities: unit, functional, performance & load testing, code analysis, test driven development (TDD), continuous integration, etc..

VoodooMock: Mock Objects Framework for C++

Shlomo Matichin

VoodooMock is a mock object framework that fully automates the repetitive tasks associated with writing mock objects for C++ unit tests. Similar to other frameworks, VoodooMock provides an expectation storage and verification engine. But unlike other frameworks, VoodooMock also automates the generation of stubs, and rerouting calls to the stubs, without requiring any macros or unnecessary interfaces. VoodooMock is available under the GPL, and is successfully deployed in at least three production projects.

Web Site: http://sourceforge.net/projects/voodoo-mock
Version Tested: 0.3
System Requirements: Python 2.5 or higher
License and Pricing: GPL
Support: Through the SourceForge project homepage
Installation: just unzip
Documentation: contained inside the package
Configuration: works out of the box

Introduction

When writing unit tests based on mock objects, there are two main implementation details to address: Interception, or how method calls are forwarded to the mock object, and the Expectation engine, or how to store and verify expected events.

Other frameworks do not handle interception. The test writer usually implements interception by using an abstract class inheritance ("interface class") and a factory (even when there is only one class that inherits the interface). The factory is replaced at the unit test with a stub that returns a stub object instead. The stub object forwards the calls to an expectation engine. All this code is written manually. It also has a demoralizing effect of testing everything through as much existing interfaces as possible, to avoid this type of repetition.

VoodooMock addresses interception differently. A pre-build phase of the test suite build system, the VoodooMock parser generates a parallel directory hierarchy of the project headers, each replaced with an automatically generated stub for all the classes and functions of the original header, the voodoo header. The voodoo header contains ifdef directives to allow including the original header instead. So by just defining some macros before the original code is included in the test, the tester can select which classes are to be replaced by stubs. The stubs in turn, contain full implementation for forwarding all access to the expectation engine.

A drawback for this approach, is that the build system must enable this pre-build support, and compile each test-suite into a different binary file (the stub and real class cannot coexist in the same binary, having the same name and namespace). A test harness that runs all these separate binaries is also recommended. The VoodooMock project at sourceforge contains a sample harness written in python, and a Jamrules file ('Jam' is a 'Make' like language), as a reference.

However, this approach eliminates the need for virtual interfaces, or any other real code modifications (e.g., no macros are needed). One project using VoodooMock was a kernel mode driver, where unnecessary virtual interfaces were unwelcome.

VoodooMock's expectation engine uses Scenario objects, which contain a half textual half object oriented description of what all the stubs in the test are expected to verify, and in what order. A tester will spend most of his time writing these scenarios.

VoodooMock is called voodoo, because it uses bad practices, hacks, and black magic to implement all of its features (i.e., globals, god objects, void pointers, and abusing the header inclusion order). Testing code is allowed to contain hacks to make the test work, but VoodooMock's approach isolates the hacks into the generated code, and out of the test code. Generation of the stub code also means that once a hack is used, it is consistent across all stubs in the test suite. A lot of hacks were designed in order to allow full support of inheritance and templates. For example: an inheriting class can be tested as a different unit than its ancestor, which is stubbed.

A Short Tutorial

The following example provides a quick tutorial of using VoodooMock. The example contains two classes: Number, which is just an object wrapper for unsigned, and Sum, which performs addition on two Number s. The contents of the file Number.h:

```
#ifndef __NUMBER_H__
#define __NUMBER_H__
class Number { public:
Number( unsigned value ) : _value( value ) {} unsigned value()
const { return _value; } private: unsigned _value; };
#endif
```

And the content of Sum.h:

```
#ifndef __SUM_H__
#define __SUM_H__
#include <Number.h>

class Sum { public:
Sum( const Number & first , const Number & second ) :
_result( first.value() + second.value() )
{
}
unsigned result() const { return _result; } private: unsigned
_result; };
#endif
```

A test for the unit that contains only the class Sum (stubbing the Number class), can be:

```
#include <stdio.h>
#define VOODOO_EXPECT_Number_h
#include "Sum.h"

using namespace VoodooCommon::Expect; class TestFailed {};
int main()
{
Scenario scenario;
scenario <<
new CallReturnValue< unsigned >( "Fake Number 1::value" , 100 )
<<
```

```
new CallReturnValue< unsigned >( "Fake Number 2::value" , 200 );
FakeND_Number number1( "Fake Number 1" );
FakeND_Number number2( "Fake Number 2" );
if ( Sum( number1 , number2 ).result() != 300 )
throw TestFailed();
scenario.assertFinished();
printf( "OK!\n" );
return 0; }
```

Lets examine the code in detail. First, the following two lines:

```
#define VOODOO_EXPECT_include_Number_h
#include "Sum.h"
```

The first lines tells the preprocessor, that if Number.h gets included, use the stub code, instead of including the real Number.h. The macro name is derived from the header file name, by the VoodooMock stub tree generator. The second line includes Sum.h, which actually includes the stub Sum.h first, and since VOODOO EXPECT include Sum.h is not defined in this test, the stub simply includes the original Sum.h. Sum.h includes Number.h, and this is when the macro defined in the first line makes a difference. Note: the macro name is created from the relative path of the header file from the project root. In this case, the folder ‘include’ is the prefix to the header file name. The test starts with the following lines:

```
Scenario scenario;
scenario <<
new CallReturnValue< unsigned >( "Fake Number 1::value" , 100 )
<<
new CallReturnValue< unsigned >( "Fake Number 2::value" , 200 );
```

This snippet creates a scenario object, and fills it with the following two events: First, the method ‘value’ of a stub object with the name ‘Fake Number 1’ will be called, with no parameters. It will return an unsigned, with the value 100. Then, the method ‘value’ of a stub object with name ‘Fake Number 2’ will be called without parameters. This time it will return 200. The next section explains in detail about the scenario object and what it can store. The code continues:

```
FakeND_Number number1( "Fake Number 1" );
FakeND_Number number2( "Fake Number 2" );
```

The above snippet creates two Number objects, bypassing the construction phase, and naming them in the process. More about the ‘FakeND ’ prefix in the next section. Note that the objects are created after the story already references them. The example continues:

```
if ( Sum( number1 , number2 ).result() != 300 )
throw TestFailed(); // But C++ test suites usually write this as
// TS_ASSERT_EQUALS( Sum( number1, number2 ).result(), 300 );
```

This snippets actually calls Sum, first constructing it over the two stub objects, and then calling the ‘result’ method, and finally asserts the result. The constructor Sum::Sum first calls the ‘value’ method of the object ‘number1’, which has the textual name ‘Fake Number 1’, that matches the first event in the only alive scenario object. Therefore the return value is fetched from that event, and the scenario object advances to the second event. Sum::Sum continues to call the method ‘value’ of the second object, matching the second event, and completing the scenario.

The last line of interest:

```
scenario.assertFinished();
```

This line verifies that the scenario object ‘scenario’ has finished matching all of its events. VoodooMock configuration file defines how this assertion is implemented (which is usually just TS ASSERT).

Summary: this example shows how to:

- Create stub objects without using the constructor.
- Create a scenario recording, run it, and verify it was complete.
- Assert the unit gave a correct result under that scenario.

Features Of The Expectation Engine

This section describes the different capabilities of the VoodooMock expectation engine, without giving any source code examples. An expectation is one of the following “events”:

- Construction or Destruction of a stub class
- Call to a stub object method
- Call to a global function

The expectation engine is responsible for storing, running and verifying “recordings” of expectations. Each event has a textual representation (e.g. “Construction of std::list<unsigned>”). When an event occurs (e.g., a stub class is constructed), the engine searches for matching expectation in the alive Scenario and Always objects (see next subsection). In order to match, the expectation textual representation must match the event textual representation, plus the parameters must also pass their verification (stored in the expectation - see the ‘Parameters’ subsection).

The features of the expectation engine are described in the following subsections.

Scenarios

Scenario objects store expectations, meant to run one by one and only once (e.g. first create the file, write to it, then close it). Before a Scenario object is destroyed, one of the following methods must be called: ‘assertFinished’ will assert all the expectations were matched, or ‘assertNotFinished’ otherwise.

Several Scenario objects can coexist, representing independent “recordings” of expectations. If both Scenario objects currently point to two expectations matching the current event (in both textual name, and parameters verification), the Scenario object constructed last takes precedence. Only the return value from that Scenario object will be used, and only it will advance to the next expectation. This behavior is useful in test driven development: first write a “good” scenario, and when writing the “bad” cases, use the “good” scenario, but only “override” the failing event.

‘Always’ objects are lists of unordered expectations, that can be called zero or more times (e.g., gettimeofday and logging). Several Always objects can coexist, together with Scenario objects. Scenario objects always takes precedence over Always objects. When in conflict, the last

Always object created take precedence over other Always objects. If the expectation conflict is inside a single Always object, the expectation last added to the object takes precedence. This behavior provides a similar facility for “good” and “bad” scenarios: create a single Always object for the “good” test, and fail each call one by one in other tests.

‘ExpectationList ’ is a container of expectations, that is not connected to the expectation engine (i.e., expectations stored in it will not be matched). It allows creating “subscenarios” to be added into a Scenario or an Always as a group (used for dividing scenario building into functions).

All three containers support simple editing. Editing a scenario helps writing readable tests: most of the tests will only describe the “difference” from one of the major testing scenarios. The editing primitives include:

- Iterating containers
- Deleting or replacing expectations
- Insert ExpectationList s

Expectations

Expectations are what the tester spends most of his writing time. Each expectation matches an event the tested unit must or can do. VoodooMock is shipped with the following expectations:

- Construction expectation: ‘scenario << Construction<class>(”Name to give new object”);’
- Destruction expectation: ‘scenario << Desturction(”Name of object being destroyed”);’
- Call to a global function: ‘scenario << CallReturnVoid(”Namespace1::Namespace2::function”);’
- Call to a stub object method: ‘scenario << CallReturnVoid(”Stub objects name::method name”);’

Once a call matches an expectation, the expectation contains instruction as to what to provide as a return value. Storing the return value in the expectation also raises the question of scope for the return value stored. VoodooMock provides several calls expectations that only differ between them in the way the return value is stored. A call expectation looks like this: ‘CallReturn[ReturnValueScope](”Stub object::method”, value)’. Simple examples include:

- ‘CallReturnVoid’ is for calls returning void.
- ‘CallReturnValue’ copies the return value once when creating the expectation, and a second time when the call return. This works best for immediate values (e.g., integers).
- ‘CallReturnReference’ stores a reference to the provided return value, only copies it when the call return. This works best for functions like ‘gettimeofday’ returning a global variable representing “now”.
- ‘CallReturnAuto’ receives a pointer, copies what it references as the return value, but also deletes the pointer when the scenario is destroyed.
- ‘CallReturnCallback’ fetches the return value from a given functor.

Implementing additional call expectations is a matter of inheriting from the Expectation hierarchy.

Each expectation can also contain one or more “hooks”; functors to run after they are matched. The hook notation is similar to the parameters notation (see below).

When no expectation matches an event (e.g., the unit called a stub method which were not supposed to be called at this point), VoodooMock fails the test suite with a message containing the next expectation of every Scenario object, by line number created, and index inside the Scenario.

Parameters

Except for destruction, each expectation also describes the parameters it expects, and how to verify their correctness. For example:

```
const Serializable * secondObject; scenario <<
new Construction< DataFile >( "The Data File" ) <<
new EqualsValue< const std::string >( "C:\\\\data.dat" ) <<
new Ignore< int >() <<
new CallReturnVoid( "The Data File::serialize" ) << new Named<
const Serializable >( "First Object" ) << new Destruction( "The
Data File" );
```

Each parameter verifier is strongly typed. It will not match a different type. Overloading is supported by expecting different parameter types. Note: when using default values for parameters, the default values must also be verified. ‘EqualsValue’ verifies the parameter is equal to the value expected. Like return values, the issue of the scope for the stored value is addressed by providing multiple such verifiers (e.g., ‘EqualsReference’, ‘EqualsAuto’ and so on). The ‘Ignore’ verifier just checks the type of the parameter, but does not access it. The ‘Named’ verifier assumes the type provided is a stub class and verifies the name of the object. There are more parameters verifiers, and new ones are easily implemented.

Naming

Object names help writing readable tests. For example, a stub file object called ‘Log File’ and a second one called ‘Data File’. Each stub class exports a method ‘voodooInstanceName’ as a getter for the object name.

Whenever a stub object is copy constructed, the new object will be named “Copy of [first objects name]”. This is useful for covering exactly which object are copied, and how many times. In case where the tester wishes to ignore wither the object was copied or not, VoodooMock provides a similar version for every expectation type that ignores the ”Copy of” prefix (or prefixes). For example: ‘CallOrCopyOfReturnVoid(”Data File::write”)’ will also match the event “Call to Copy of Data File::write”, and “Call to Copy of Copy of Data File::write”.

Almost every unit, apart from creating objects and receiving them through calls, also accept them as arguments to the interface it exports. VoodooMock provides a feature to create stub objects inside the test suite without calling their normal constructor, and without passing through the expectation engine (and therefore no expectation for their construction is required). It does so by creating another class alongside every stub class, prefixed by ‘Fake ’ (e.g., for a class called ‘Connection’, another class called ‘Fake Connection’ will be generated). When constructing such an object, the test code provides the textual name for the object.

A ‘Fake ’ object can be passed to the unit whereever the original class is used (implemented by derivation, and without adding new members, to allow copy construction).

When a ‘Fake’ object is destroyed, the destruction event still goes through the expectation engine, and is therefore appropriate when handing over the ownership of an object to the unit, since it’s destruction is part of the unit’s behavior. If the unit does not own the object (e.g., receives it by reference), a second prefix, ‘FakeND ’ is more useful: the destruction event also does not invoke the expectation engine.

Externals

Stubbing OS provided global function presents two new problems: First, the OpenCCore parser (the C++ parser used by VoodooMock) can not handle OS level header files (which usually contain compiler specific macros). But even worse, the include file hierarchy is never trivial (e.g., ‘windows.h’ includes other files which actually define the system calls).

VoodooMock has a feature specifically designed for this case: provide an “alternative” header to parse, and generate stubs according to. The fake header file then always includes the original, and implements the stubs, in another names-pace, named ‘External’. For example, the alternative header file for ‘windows.h’ might look like this:

```
HANDLE WINAPI CreateFileA(
__in LPCTSTR lpFileName,
__in DWORD dwDesiredAccess,
__in DWORD dwShareMode,
__in_opt LPSECURITY_ATTRIBUTES lpSecurityAttributes,
__in DWORD dwCreationDisposition,
__in DWORD dwFlagsAndAttributes,
__in_opt HANDLE hTemplateFile
 );
 BOOL WINAPI CloseHandle(
__in HANDLE hObject
);
```

And the real code must be modified, from using ‘CloseHandle(...)’, to ‘External::CloseHandle(...)’. Now the only thing that remains is to define ‘External’ as a macro that expands to an empty string when compiling the real product.

Programmable Mock Objects

VoodooMock has a second implementation to every stubbed class, that allows complete interception of all events into methods. This implementation is faster, and more suitable for crunching tests. For more information, refer to the VoodooMock documentation.

Methodology

VoodooMock was design with the eXtreme Programming methodology in mind. This section details how one XP team used it effectively.

Selecting a unit size is an art by itself. Experiment with different unit sizes, since there is no single answer to the unit size issue. Making each class a unit provides excellent coverage with ease, but will force testing wrapper and adapter classes, and will not cover integration between classes.

When writing a test suite, use a setup and teardown routines to build and destroy the 'Fake ' dependencies, and to create the "normal" Always object and it's expectations.

Scenarios tend to grow quite rapidly. Split them into smaller subscenarios by using functions (either return an ExpectationList from a function, or append to an existing scenario). Practicing this correctly creates readable tests.

Carefully consider if an expectation should be inside a Scenario object, or an Always object. As a rule of thumb, global getters and locking should reside inside an Always, and the core unit behavior should be a Scenario. When in doubt, prefer to use Always. Consider the following case: the unit writes a string to a file. Should the test check only the file content, or how many times it might have been overridden or appended? The answer of course is "it depends on the requirements". But consider that the first alternative will not require changing the test when refactoring to use some other formatting engine, which might decide to split the write to several smaller ones.

Finally, reuse scenarios. Make the good case the first test, then only override it (more than one way) at the failure point, to cover failures. Split stories into simple smaller stories and reuse these "building blocks" to create your good cases. Same as in software development.

Set Up

VoodooMock uses a binary python module that wraps the Open C Core parser. The official release includes this module in binary form for Python 2.6 under windows. Other platforms will require building the module. The VoodooMock distribution includes the sources for rebuilding it, and makefiles for GCC and MinGW (inside the archive named wrapocccore.tbz2). Warning: The OpenCCore parser still has many bugs.

An example VoodooMock generator command line invocation:

```
tools\\voodoo\\multi.py --input=cpp --output=voodoo
--exclude=cpp\\client\\Core.h --ignore="\\bIN\\b"
--ignore="\\bOUT\\b" --ignore="\\bINOUT\\b"
--ignore="\\bOPTIONAL\\b" --only-if-new
```

The '–input' argument points to the first directory to scan for header files. The '– output' argument points to a directory where the header files mirror tree will be created. '–exclude' excludes a file from being mirrored. This is useful when the original code contains include magic, or when the OpenCCore parser crashes on a specific file. '–exclude' might be repeated as many times as necessary. '–only-if-new' will only regenerate stub header files if the modification time stamp on the original header file is newer (similar to make). A must when using a system that detects header file dependencies. '–ignore' is a list of regular expressions to erase from the code before parsing it. Since the OpenCCore parser parses the files before preprocessing, any macros or annotations must be removed before a valid C++ syntax remains.

The last thing to verify is that the directory 'voodoo' precedes the directory 'cpp' in the include path, when building the tests.

Conclusion

A good unit test is one which reflects the requirements, and is readable. A bad unit test just reflects the code itself, and is implemented in a 400-lines long test case functions.

VoodooMock's influence on unit testing mentality is similar to the influence script languages have on software development: It solves the little problems so the coder can focus on progressing rapidly. However, without the commitment to produce high quality code, script languages sometimes just help the coder accumulate a big pile of crap, quite rapidly as well. Using VoodooMock without this commitment for high quality tests, has the same effects. Its common to see a "mirroring" test, where each line in the scenario just rewrites a line of code.

As a personal advice: VoodooMock is only one tool, out of many I have written over the years to help me test code more efficiently. I have found it rewarding to keep developing my working environments and coding tools. Note that while VoodooMock is a complete package by itself, I warmly recommend working on extending and adapting it to your needs. My must have customization list includes: stack trace on expectation verification failure (redefine VOODOO TS ASSERT in VoodooConfiguration.h), a shortcut for running a single test and rerun all tests on each build.

Jazoon Conference Reports

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

Methods & Tools is the sponsor of a large number of software development conferences, but I cannot find the time and budget to visit them. This year I managed to find some time spend some time at Jazoon, a major Java event located in Zurich, Switzerland. I will present below some of the interesting things that I heard.

97 Things Every Programmer Should Know by Kevlin Henney

Nothing beats experience in software development and usually you learn the important things the hard way. This presentation is based on an O'Reilly book with contribution from 73 different people and edited by Kevlin Henney http://curbralan.com. He presented for us 17 of them in his lively style.

1) Do a lot of deliberate practices
Kevlin added: "to make the task, not to complete the task". He compared this with a cello player that will exercise for hours just to perform rarely on stage. The importance is not to realise things but to be confident when and how to use coding techniques.

2) Learn to estimate
This part deals with the difference between estimate, target and commitment. It is contributed by Giovanni Asproni, a Methods & Tools author. For a more detailed discussion, look at his article in Methods and Tools "Fingers in the air: a Gentle Introduction to Software Estimation" http://www.methodsandtools.com/archive/archive.php?id=79

3) Know your next commit
It is a good thing to have a big picture of your current project, as in "I am working on this user story for my customer", but developers should also be able to focus on the current task, as in "I am refactoring the CreateAccount code".

4) Comment only what the code cannot say
If a program is incorrect, documentation matters little.

5) Code in the language of the domain
Use in your code names meaningful for domain, like AccountNumber or CreateAccount for a bank.

6) Prefer domain specific types to primitive types
DistanceInMeters is more meaningful than integer

7) Resist the temptation of the singleton pattern
You are never sure that there will always be only one instance of an object.

8) Don't repeat yourself
Duplication is waste. Repetition in process calls for automation. Repetition in logic calls for abstraction.

9) Beware of the share
On contrast with the previous item, sharing code libraries creates dependencies for code that could evolve separately and that are just temporally coinciding.

10) The road to performance is littered with dirty code bones
Sometimes you just get better performance with less and cleaner code

11) The longevity of interim solutions
Try to avoid them... or take time to clean them after implementation.

12) The Boy Scout rule
You should leave the world just a little better than you find it and improve in little increments.

13) Two wrongs can make a right and are difficult to fix
Sometimes a bug is "corrected" by another bug further in the code. When you try to fix the second, the first one is revealed, but difficult to detect. It could be easier to leave things as they are.

14) Read code
We love to write code, but when it comes to reading it, we usually shy away. Reading code help us improve our writing abilities and increase our knowledge of programming styles.

15) Write test for people
Your tests are targeted at the person that is trying to understand your code. Tests are here to document the code and tell what it means to be right for the code.

16) Don't be cute with your test data
Because somebody else could see it ;o)

17) Ubuntu coding for your friends
This is not a reference to the Linux version, but means rather than your code might be used by your friends. If your code is bad, then their code will be bad.

The full list of 97 things is available on http://tr.im/97tepsk

Total Cost of Ownership and Return on Investment by Ken Schwaber

Ken Schwaber talked about the lack of technical practices in Scrum projects that lead to technical debt. However, the end of the talk made me thinks that this topic could have been chosen only to promote its Scrum.org certification courses.

Ken Schwaber started by saying that we often consider only the costs of the development phase, but not the costs of the total life cycle of the software. Therefore, we ignore the impact of short-term decisions on long term return on investment (ROI). The ROI can be measured with several aspects like the valuable functionality included or the absence of low value features.

He discussed after this the meaning of quality for the developer
* I can understand the software
* I can change the code without side effect
* I can check if things work after a change

This lead to an exercise where he asked the audience to agree on a definition of "done", using the context of multiple teams developing common software. He then asked if the agreed definition included:
* performance testing
* stability testing

* integration with other teams
* user acceptance testing
* code reviews

His point was that you should have a clear definition of done that include everything needed to deliver software to the user, otherwise you will have the same "death march" at the end of agile projects during the "stabilization" phase than in waterfall projects. He remembered an early embarrassing personal experience where his definition of "done" as a developer was different from the definition from the user and he had to explain what was needed to achieve it. When you have a gap between the team sprint results and what you really need to be "done", you will create at every sprint an increasing backlog of work that will have to be cleared before delivery of the application.

He gave the example of the number of defects for the same team with two different options. In the first release, integration testing was not included in sprints. The number of defects strongly grows at the end of the project. For the second release, they included integration testing in every sprint activity. The number of defects was kept under control and the software was completed before the deadline.

"Not having enough time" is the excuse often presented by teams for not including needed activities in their sprints. However, these are things that will have to be done later at a higher cost. Additionally, as you are accumulating bad code, you create a technical debt that lowers your velocity. Ken Schwaber defines Scrum as a light framework without technical practices. This was done intentionally because people should know better what they needed to do. To his surprise, the holes were not filled well. He cited a survey conducted by Jeff Sutherland finding that more than 50% of "Scrum" teams were not using the iterative/incremental practice. This is what Martin Fowler calls "Flaccid Scrum": http://martinfowler.com/bliki/FlaccidScrum.html

Ken Schwaber used the last 10 minutes of his (reduced) speech to present, I will rather say advertise, the new certifications (Professional Scrum Developer) proposed by Scrum.org. He mentioned several courses offered by a local partner. This sadly makes me think that money could have been a major reason for him to split from the Scrum Alliance.

Java SE and SDK 7 +

Danny Coward presented the first keynote of the Jazoon conference. He mentioned that we were close to the 15 anniversary of Java that was officially announced May 23 1995. On the historical side, he also showed us a nice video of James Gosling demonstrating in 1992 a prototype running on what will become Java. The interface looks very close to what you get now on an iPhone. You can watch it on http://www.youtube.com/watch?v=Ahg8OBYixL0

He discussed the current evolution of Java towards its version 7. There has been a lot of work on the modularity side, removing as much as possible the dependencies between the Java modules. Parallelism has also been improved and will now for instance be used by most of the garbage collector activity. More than 100 languages are now running on the JVM. The Da Vinci project (http://openjdk.java.net/projects/mlvm/) goal is to improve the efficiency of other languages on the JVM. Finally, some small additions are made to the syntax to simplify the language. JavaFX release 1.3 was also announced in May, the fourth release in the last 18 months. Performance has been improved and new UI components have been added. For those who want to see it in action, Danny pointed us towards the web site of the last Vancouver Winter Olympics, more precisely the geographical view of meals. http://www.vancouver2010.com/olympic-medals/geo-view/

Java and Flex in Enterprise

The next two sessions that I attended were both about Java and Flex. In the first presentation, Adobe Evangelist James Ward (http://www.jamesward.com/) did first a small introduction to the Flex technology. Then he showed how it was easy to connect a Flex front end with a Java back end using different protocols (http, web services) and benchmarked their relative speed. All his demos are available on http://www.jamesward.com/demos/. You can check the insurance one for a small taste of what a nice Flex interface can look like.

The second presenter, Florian Müller (http://www.richability.com/), put a different perspective on the marriage between these two technologies with his experience on a large project that was involving both of them. The development of the application can be very easy, but the maintenance is more difficult as functions can be performed either at the client (flex) or server (level). Architecture rules have to be defined very well and synchronization between the server and the client is not easy to maintain.

Flex 4 solves part of these problems, but the proposed life cycle (Photoshop - Catalyst - Flex) does not work well when you have go backwards and redo some things. In the tools that you can use to link Java and Flex, he recommends Granite

Value of Objects

Kevlin Henney presented a lively talk about the concepts of object and value in software development. Kevlin is a lively and wise presenter, so don't miss this guy if you have the chance to be at a conference where he speaks. His talk was a kind of philosophical musing even if he was always firmly rooted in programming.

It is difficult for me to summarize all the ideas that were discussed, but here are some nice quotes excerpted from this presentation (I hope to be precise, but all misunderstanding are my own responsibility):
* As a profession software development do a mess of words
* If you are using java.util.date, you must be tired of life
* Typing is not the bottleneck in software development
* Every time a mock returns a mock, a fairy dies
* Don't confuse the ideas of the programming language with the essence of what you are trying to express.

Maven 3.0

Matthew McCullough http://www.ambientideas.com/ made a very pump up presentation of the improvements that will be available with Maven 3, currently in its final beta testing period. The new version will be faster and have a smaller footprint, due to a codebase reduced by 30%. The software has been transformed in a code library that allows a better integration with third party tools. An important improvement is the possibility to define your configuration in different languages (ruby, groovy, scala, clojure) with polyglot maven. http://polyglot.sonatype.org/. He also presented the last version of m2eclipse http://m2eclipse.sonatype.org/ that he likes for its ability to visualize dependencies and to provide tools for an easier refactoring.

Slides of all the presentations of Jazoon are available on the conference web site http://jazoon.com/. Videos will be published late on the excellent Parleys web site: http://parleys.com/

Load Tester 4: The easiest, fastest, most complete load testing software available. Spot performance problems in your operating system or application server quickly, and cut your load testing time by up to 80%. Our automated record and configuration process means no complicated hand-coding, and no scripting languages to learn.

http://www.webperformance.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
The content of this publication cannot be reproduced without prior written consent of the publisher
Copyright © 2010, Martinig & Associates