# METHODS & TOOLS

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

Summer 2007 (Volume 15 - number 2) www.methodsandtools.com

## What's Right with Agile Approaches

In the recent years, the agile development approaches have been gaining more and more interest and acceptance in software development organisations. The basic value of agile approaches is to put back some emphasis on the importance of people collaboration, for both developers and customers, in software development projects.

This position was taken at a time where the software development world was under the influence of "process-centred" visions that could be exemplified by approaches like the Capability Maturity Model (CMM) or the Rational Unified Process (RUP). These visions provide an enormous amount of valuable material on how to develop software in a well-structured framework. Even if their goal was to provide a global framework that has to be adapted to the specific situation of each project, the natural tendency of people was to perform every activity proposed, often for fear of missing something important. The basis of the agile movement is what I would call a "bottom-up" approach. You have to do just the minimal set of activities to produce quality software that satisfies the customer needs. Agile is also appealing for people who are ready to trade the sometimes false security of planning for the added flexibility of an always changing context. Agile approaches are not the silver bullet of software development. "Constant customer collaboration" is not so easy to realise in the real life. Tom Gilb also argues that agile approaches are soft on quantification and many thinks that "you cannot manage what you cannot measure". Agile ideas could not be considered as new. Prioritising requirements for short iterations could be seen as rebirth of evolutionary development or of the "time boxing" principle of RAD. Finally, agile approaches will have also to face the "mass adoption" barrier. Early adopters of new approaches are often motivated and intelligent people trying to improve their current situation. With this kind of people, it is easier for projects to succeed. It is when the "average" developer starts to use a new approach in the "average" organisation that its value can be fully acknowledged.

There is nothing right or wrong with agile approaches. Each project has its own context: development teams, customer requirements and organisational environment. You have to adopt the process that gives your project the most chances to succeed under the current circumstances, taking valuable tools from every approach.

[illegible]

## Inside

# An Agile Tool Selection Strategy for Web Testing Tools

Lisa Crispin
http://lisa.crispin.home.att.net/

Selecting a test automation tool has always been a daunting task. Let’s face it, just the thought of automating tests can be daunting! The selection of tools available today, especially open source tools, is positively dazzling. In the past several years, “test-infected” developers, not finding what they need in the vendor tool selections, have created their own tools. Fortunately for the rest of us, many are generous enough to share them as open source. Between open source tools and commercial tools, we have an amazing variety from which to choose.

To avoid that deer-in-the-headlights feeling, consider taking an ‘agile’ approach to selecting web testing tools. Plan an automation strategy before you consider the possible tool solutions. Start simple, and make changes based on your evolving situation. Here are some ideas based on experiences I’ve had with different agile (and not so agile!) development teams. Even if your team doesn’t use agile development practices, you’ll get some useful tips.

## An Agile Test Automation Strategy

First of all, your team should consider your testing approach. When I say ‘team’, I’m thinking of everyone involved in developing and delivering the software, which in your case might be a virtual team. When do you write tests?  Who writes them? How should the test results be delivered? Who needs to be able to look at the test results, and what should they be able to learn from them? What kind of tests need to be automated, and when? Do you have other tedious tasks, such as populating test data or looking through version control system output, that you’d love to automate?

Back in 2003, my current team had no test automation at all, and a buggy legacy web-based J2EE application. We desperately needed to automate our regression tests, since the manual regression tests took the whole team a couple of days to complete, and we were delivering new code to production every two weeks. We had decided to start rewriting the system, developing new features in a new architecture, while maintaining the old code, but this would be impossible without a safety net of tests.

We committed to using to test-driven development for a number of reasons, one being that automated unit tests have the highest return on investment of any automated test. We went a step further, and decided to also use ‘customer-facing’ tests and examples to help drive development. We’ve found that one example is worth pages of narrative requirements! We wanted to be able to write high-level, big-picture test cases before development starts, and then write detailed executable test cases concurrent with development so that when coding is finished, all the tests are passing.

Meanwhile, we required some kind of ‘smoke test’ regression suite for the legacy application, to make sure that critical parts kept working. Due to the old code’s architecture, we decided these would have to be done through the GUI. We wanted all of our tests to run during our continuous build process, which was automated using CruiseControl, so we’d have quick feedback of any regression failures.

Quick and easy-to-read notification of whether tests passed or failed was important to us. Ideally, our build would include these results in an email. In the event of a failure, we wanted to be able to quickly drill down to see the cause.

Platform is an obvious consideration. Our build runs on Linux, and our application was running on Linux, Solaris and Windows at the time. Any test tools that, for example, only ran on Windows did not have much appeal.

Based on all these needs, we started searching for tools. Our whole team takes responsibility for quality and testing, so we all needed to agree on our automation approach and tools. Having programmers, testers, database specialists and system administrators collaborate on test automation leverages a variety of skills to help get the best solutions. I highly recommend taking a ‘whole team’ approach to deciding on a test automation strategy, choosing and implementing tools.

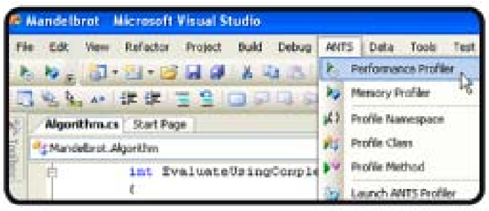
## An Agile Tool Selection Strategy

The whole team approach means asking ourselves, “What skills do we have on our team?” Do any team members have extensive experience with particular test tools or types of test tools? What programming and scripting language competencies exist on the team? How much technical expertise do the testers have? How about the business people who might be reviewing or even helping to write tests? What types of tests are you automating? Unit, integration, functional, security, or do you need to do performance or load testing? How robust do your test scripts need to be, and how much can you spend on maintenance? Are you planning to do data-driven or action keyword type tests where the tests accept a variety of input parameters and have a lot of flexibility? Or are you looking for straightforward, low-maintenance tests? Can you test at a layer below the user interface, or do you have an architecture that makes that difficult? These are all considerations when shopping for a test tool.

Advertisement – Code and Memory Profiler for .NET - Click on ad to reach advertiser web site

Code and memory profiler for .NET

ANTS Profiler™ Pro is a code and memory profiling tool. It is used to pinpoint performance bottlenecks in .NET applications, including ASP.NET web apps and Windows Services. ANTS Profiler allows you to drill down and examine the specific lines of code responsible for performance inefficiencies, so code can be optimized.



Use ANTS Profiler to:
- Identify performance bottlenecks quickly
- Improve the performance of your application
- Optimize memory usage and locate memory leaks

Visit www.red-gate.com/code_profiling.htm for more information and your free trial.

ANTS PROFILER™

redgate®
ingeniously simple tools

With a variety of test needs, consider that you may need a variety of tools. We tried to keep an open mind on what might solve a particular automation problem, and we were willing to experiment. We'd pick a tool to try for a few iterations and see how we liked it. Getting up to speed on tools to the point where you can effectively evaluate them takes time, so be sure to budget plenty of time in your planning.

## Our Tool Selection Process

If your team has specific needs that may not easily be met by a generic test tool, and you have the skill set to accomplish it, consider "growing your own". This way, you get a tool is customized to your needs and integrates well with your application. Many teams have chosen this route, which is why there are so many excellent open source tools available today. If you 'home brew', you still need to consider your test tool requirements. For example, if non-programmers need to specify tests, you'll have to develop an interface to allow that.

In our case, our financial web-based application didn't seem all that unique, and we were a tiny team without a lot of bandwidth for tool development. Our management included funds for test tools in the budget, so we looked for an outside solution.

To illustrate how your tool selection process might work, come back in time with me to late 2003 / early 2004 when we started our search. If we started from scratch in 2007, there would be even more options to consider! A couple of great places to start your search for web testing tools are www.softwareqatest.com/qatweb1.html and www.testingfaqs.org, which list both commercial and open source tools, and www.opensourcetesting.org, which provides information about all kinds of open source software testing tools. Another good resource are members of your local testing/QA user group, and testing-related mailing lists such as http://groups.yahoo.com/group/agile-testing.

We had to address our various needs with tools. Unit testing was a no-brainer. With a Java application, JUnit is the obvious choice. The team got started writing unit tests right away, as they would form the foundation of our regression tests. Next on the priority list was GUI testing.

## Vendor Tools

We desperately needed to get some automated smoke tests going, and thought maybe we could buy a tool that we could run with. As a tester, I have extensive experience with the capture/playback/scripting type of automation tools. Vendor tools are often a safe choice. They're generally designed for non-technical users, who can get started fairly easily. They come with user manuals and installation instructions. Training and technical support are available for a fee.

Commercial tools are often integrated with a suite of complementary tools, which can be an advantage. Many vendors offer functional, performance and load test tools. They offer impressively robust features. However, they tend to be targeted towards test organizations, and aren't very 'programmer-friendly'. They can be difficult to integrate into a continuous build process. They often have proprietary scripting languages, or are limited to one scripting language such as Javascript.

I had previously used Mercury test tools, so QuickTest Pro was one option to consider among many commercial tools. Other vendor tools we could have considered, or could consider if we were looking today, are TestPartner, Rational Functional Tester, SilkTest, BadBoy, TestMaker and (one I have always wanted to try) LISA. We also tried out Seapine's QA Wizard, since we used their defect tracking tool TestTrack. At that time, both of those capture/playback tools used proprietary scripting languages. We didn't want to be limited to only capture/replay, as those scripts can be more work to maintain. The programmers on my team didn't want to have to learn a new tool or new scripting language. That ruled out all the vendor tools we looked at.

## Open Source Tools

We turned to open source tools. Since these were generally written by programmers to satisfy their own requirements, they're likely to be programmer-friendly. But there are issues with open source tools as well. Support, for example. If you have a question about an open source tool, whom do you ask? Most of the open source tools we considered had mailing lists where users and developers shared information and helped each other. I checked each tool's mailing list to see if there was lots of activity on a daily or weekly basis, hoping to see a large and active user base. I also tried to find out if users' issues were addressed by the tool's developer community, and whether new versions were released frequently with enhancements and fixes. Of course, with open source tools you're free to add your own fixes and enhancements, but we knew we wouldn't have the bandwidth to do this at first.

Open source tools have a wide range of learning curves. Some assume programming proficiency. This is fine if everyone using the tool is able to achieve that level of competence. Others are geared to less technical users, such as testers and analysts. Some have user documentation on a par (or even better) with good vendor tools, and others leave the learning more up to the user. Some even have bug tracking systems, and the developers actually fix bugs!

Accessibility
Compatibility
Broken Links
Legal Compliance
Usability
W3C Compliance
Search Engine Friendly
Question
Does your website tick all the boxes?
SortSite is a standards-based testing tool that makes it quick and easy to check websites for quality issues.
Visit www.electrumsolutions.com for further information and your 30 day free trial.
SortSite
www.electrumsolutions.com

Think about the level of support and documentation you will need, and find a tool that provides it. We were looking for a tool that came with a lot of help.

**Our GUI Tool Search**

One example of a tool we researched, but didn't try out, was JWebUnit. Since this tool lets you create scripts with Java, it appealed to the programmers on the team. At the time (2003), it didn't seem to have as many users or as much mailing list activity as other open source test tools. We considered other Java-based tools, such as HtmlUnit, which is widely used. I had used a similar tool, HTTPUnit, before, and had liked it well enough. However, all these Java-based tools were a problem for my severely limited Java coding skills. While we anticipated that the programmers would do a large percentage of automating the customer-facing tests, I needed to write most of the GUI test scripts. I wanted to get a smoke test suite to cover the critical functionality of the legacy system, while the programmers got traction on the unit test side. We needed a programmer-friendly tool, but also a Lisa-friendly tool.

We considered scripting tools such as WATIR and scripting languages such as Ruby. I'd used TCL to write test scripts on a prior team, and I like the flexibility of scripting languages. We didn't have any Ruby experts on the team, and although I was eager to learn it, the time it would take was an obstacle.

We looked for something that required less OO programming proficiency. I'd heard good things about Selenium, but at the time it had some limiting factor such as being difficult to integrate into our build process. Another tool that would have been a strong contender, but either it wasn't available yet or I just didn't know about it, is Jameleon.

After much research, we decided to try Canoo WebTest. This tool, based on HtmlUnit, uses XML to specify tests, and the scripts run via Ant. Since we use CruiseControl for our builds, it was simple to integrate the WebTest scripts with our continuous build. Being used to the features found in commercial tools, WebTest at first seemed a bit simplistic to me. At the time, it didn't support things like if logic, except by including scripts written in Groovy or other scripting languages. It didn't look easy to do data-driven tests with WebTest. However, we liked the idea that the tests would be so simple and straightforward, we wouldn't have to test our test scripts. WebTest seemed a good choice for creating a smoke test suite.

Advertisement – Web Testing & Analysis Suite - Click on ad to reach advertiser web site

Concerned about the Quality of Your Web Application or Website?

eValid™ Web Testing & Analysis Suite is a Complete Quality Solution:

- Functional Testing with AJAX Support
- Real Server Loading Driven by Multiple Browsers
- Automated Regression Testing
- Site Analysis of Website Tree With 3D-SiteMaps

eValid

*Free product evaluation download and info available at www.e-valid.com*

The programmers were comfortable with WebTest, since they were all familiar with XML. If a test failed, they'd be able to understand the script well enough to debug the problem. They could easily update or write new tests. We decided to try it for a few iterations. Since the programmers were busy with learning how to automate unit tests, it was helpful to have a GUI test tool that was easy for me to learn. I implemented it with some help from our system administrator. We soon had two build processes, one running all the unit tests, and the other running the slower WebTest scripts. It took about eight months to complete enough scripts to cover the major functionality of the application. These scripts have caught many regression bugs, and continue to catch them today. The return on our investment has been awesome.

## Extending Our Coverage

Automated GUI test scripts are by far the most fragile and expensive to maintain, although WebTest's features minimize the need for changes. These scripts were fine for smoke tests, to make sure nothing major or obvious in the application was broken, but we didn't want to do detailed functional testing this way. Also, we still needed a tool to support our plan to drive development with customer-facing, executable tests and examples.

Once we had traction both at the unit test and GUI test level and could take a breath, we looked for a tool that filled up the big gap in the middle. We looked at FIT and FitNesse (which is essentially FIT using a wiki for the IDE) since they allow a non-programming user to write test cases in a tabular format and programmers to easily write fixtures to automate them. These tools basically replace the UI. They allow you to send test inputs to the code, operate on them, return actual results and compare them automatically with expected results. The results turn red, green or yellow, and we love color coding. Both tools have a large user base and active mailing lists. We liked FitNesse's wiki component, so we decided to try it.

FitNesse turned out to suit our needs for documenting not only the features we developed, but other information about maintaining the application. We found that it was easy to learn how to define test cases and write the fixtures to automate them. Integrating the test suites into our build process took longer, but was doable. It was easy to write both high level tests to give the big picture, and executable tests to capture the detailed requirements.

Sometimes you get unexpected benefits from tools. We understood that creating FitNesse tests would require work from both a tester, to specify test cases, and a programmer, to automate them. We found that this enforced collaboration enhanced communication within the team. The resulting communication flushed out misunderstandings and wrong assumptions early in the development process. Test tools aren't just for automation!

## Be Open to Experimentation

Several months later, we had good coverage on the GUI side from our smoke test scripts, a safety net of unit tests for new code (plus the benefits of TDD), and used FitNesse tests to help guide development. We still had unfulfilled testing needs.

One difficulty was setting up test data. We had test databases, but creating data to achieve a specific scenario was often a tedious manual task. When we hired a tester with extensive Perl experience who was also interested in learning Ruby, we decided to give WATIR a try. Although we already had a GUI test tool, our new WATIR scripts allowed us to quickly create any combination of data we needed, making it much easier to do serious exploratory testing. No doubt, we could have probably done the same thing with the tools we already had, if we had spent the time researching how to do it. Having someone with skills to get us all up to speed with WATIR scripts made that the path of least resistance, although we had to plan time to learn Ruby. (I enjoyed learning Ruby, and enjoyment is an important factor too!)

Another item missing from our toolbox was a load test tool. When we needed a way to do load testing a couple of years down the road, we started doing some research. I asked the members of the agile-testing Yahoogroup for load test tool suggestions. Recommendations included Httperf, Autobench, OpenWebLoad, Apache Bench, Apache JMeter and The Grinder. After considering factors such as scripting language, learning curve, result content and formatting, recommendations from other users, and compatibility with our application, we budgeted time for trial runs of both JMeter and The Grinder. We had the best results with JMeter, a Java desktop application, and found that a tool called Badboy helped us create the JMeter scripts. We liked the statistics it generated, and how they were displayed. We were productive with JMeter quickly and were happy with our choice. As with all our tools, though, we're always open to new developments on the tool front.

Even after you have test automation in place, there is so much development of new and existing tools that it pays to keep up with what's new. Monitor mailing lists such as http://groups.yahoo.com/group/agile-testing, attend local user group meetings, read online and print publications (such as this one!) and attend conferences when possible to catch the latest tool buzz. Recently a couple of our team members saw demos of Selenium, and the agile-testing mailing list had posts from happy Selenium users. It has some potential advantages for us, so when we can budget time for it, we'll check it out. We don't intend to throw any existing tools away, since they're still working for us. But if there's a better way to automate particular types of tests, we're open to trying it.

Your needs might change and evolve, as ours did. As team members come and go, your team's skill set might change. When our WATIR expert move on, my Java programmer teammates bought Ruby books and started learning more about it. However, now there is a Java-based option, WATIJ, that might be more appropriate to our needs (there is also a .NET version, WATIN). Part of Selenium's appeal is the ability to write the scripts in Java (which the programmers like) or Ruby (which I like). More and more tools, especially the open source ones, support multiple languages and provide more flexibility.

Tools aren't just for automating regression tests or helping with exploratory testing, either. Scripting languages such as Ruby can automate all kinds of tedious tasks that teams need to do. Get ideas from books such as _Everyday Scripting with Ruby_ by Brian Marick.

Advertisement – Load Test Your Web Application - Click on ad to reach advertiser web site

**Load Test Your Web Applications**

● ● ●

NeoLoad is a simple to use, cost-effective stress tool that measures the performance of your web applications, identifying potentially damaging issues before they occur.

**>> Discover our solution** **>> Download now**

NEOTYS
www.neotys.com

Sometimes tools can be combined for even more advantages. Groovy scripts can be integrated into WebTest scripts to provide more flexibility. Selenium tests can be run from FitNesse. Those are just a couple of examples. Don’t limit your thinking to an individual tool’s features.

## Successful Tool Implementation

What have our test tools done for our web application development? Just a few months after adopting JUnit, Canoo WebTest and FitNesse, we had a suite of regression tests that gave us a useful safety net, and our defect rate was going down dramatically. Today, three years later, we have increased our test numbers by a factor of ten or more. Our regression suites, running many times per day in two continuous builds, catch bugs at least a few times a week. Our rate of defects introduced during development is down by half. We have time and tools to do robust exploratory testing and load testing. Most importantly, but harder to measure, using our tests to drive development has resulted in features that delighted our customers.

My team put plenty of time into the research and tool adoption I’ve described here. Test automation is a big investment, but carefully done, it returns many times what you put into it. You need enough resources to first define what you need, then investigate your options, then to try out tools. Our team takes two weeks, twice a year, to devote to tasks such as researching new tools, and refactoring tests and code. This may seem like a luxury, but our management knows it helps us keep our technical debt to a minimum, so that we can improve our future productivity.

Depending what people have which skill sets, it may pay to pair people up to research or try out a test tool. A programmer and a tester could team up to try out a scripting language. A system administrator might help determine if a tool can be integrated into the team’s build process. Have brown bag sessions to brainstorm ideas, or start a book club to get ideas from publications.

Remember the ‘whole team’ approach. The team should come to a consensus on what tools to build, to try or to adopt. If a tool isn’t producing expected benefits, the team should decide whether to try a new approach or a different tool. More experienced members of the team can coach their less experienced coworkers to help everyone get up to speed on using the new tool. You may need to involve experts from outside your team to help you succeed with the new tool. People outside your team who need to use the tool, for example, to specify tests themselves, will need your help.

## What If Your Team Isn’t Interested?!

“This sounds very nice,” you say, “but my team is so overloaded and busy, nobody has time to think about tools, and they don’t think it’s important to automate tests.” Not everyone has like-thinking team members. Or maybe you’re a tester on an isolated QA team that isn’t getting much support from programmers or other groups.

Don’t despair, just get creative. I once worked in a chaotic company developing a retail internet application. Despite encouragement from the development manager, the programmers automated few unit tests. The company owned licenses for a vendor GUI test tool. We hired a tester with expertise in that tool, who could automate some regression tests and teach others how to use the tool. But the tool couldn’t address all our automation needs.

The web application was written in TCL, so there were several TCL developers. On various mailing lists, I ran across several people using TCL effectively for test automation. I decided to teach myself enough TCL to create test scripts for areas we couldn’t cover with the vendor tool. While the developers weren’t interested in test automation, they were happy to help me with my

TCL coding problems. This shows how it pays to leverage the expertise around you. Although far from an ideal situation, we automated enough regression tests to free up our time for useful exploratory testing.

People, not tools, make projects successful. It doesn't matter what tools you use or develop, as long as they help you towards your goals. Collaborating to choose and use the right test tools for your team's situation helps allow all team members to do their best work. That's the bottom line for test automation.

## References

- www.testing.com
- www.opensourcetesting.org
- www.softwareqatest.com/qatweb1.html
- www.testingfaqs.org/
- www.io.com/%7wazmo/papers/homebrew_test_automation_200409.pdf
- fit.c2.com
- www.fitnesse.org
- webtest.canoo.com
- wtr.rubyforge.org
- www.openqa.org/selenium
- jameleon.sourceforge.net
- Jakarta.apache.org/jmeter/usermanual/index.html
- _Everyday Scripting with Ruby_, Brian Marick, Pragmatic Bookshelf, 2007

# What's Wrong With Agile Methods
# Some Principles and Values to Encourage Quantification

Tom Gilb, http://www.gilb.com/
Lindsey Brodie, Middlesex University

## Abstract

Current agile methods could benefit from using a more quantified approach across the entire implementation process (that is, throughout development, production and delivery). The main benefits of adopting such an approach include improved communication of the requirements and, better support for feedback and progress tracking.

This article first discusses the benefits of quantification, then outlines a proposed approach (Planguage) and, finally describes an example of its successful use (a case study of the 'Confirmit' product within a Norwegian organization, 'FIRM').

## Introduction

Agile Software Methods (Agile Alliance 2006) have insufficient focus on quantified performance levels (that is, metrics stating the required qualities, resource savings and workload capacities) of the software being developed. Specifically, there is often no quantification of the main reasons why a project was funded (that is, metrics stating the required business benefits, such as business advancement, better quality of service and financial savings). This means projects cannot directly control the delivery of benefits to users and stakeholders. In turn, a consequence of this is that projects cannot really control the corresponding costs of getting the main benefits. In other words, if you don't *estimate* quantified requirements, then you won't be able to get a realistic *budget* for achieving them. See Figure 1 for a scientist's (Lord Kelvin's) opinion on the need for numerical data!

> *"In physical science the first essential step in the direction of learning any subject is to find principles of numerical reckoning and practicable methods for measuring some quality connected with it. I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the state of Science, whatever the matter may be."*
>
> Lord Kelvin, 1893

Figure 1. A statement made by Lord Kelvin on the importance of measurement. From http://zapatopi.net/kelvin/quotes.html

Further, quantification must be utilized throughout the duration of an agile project, not just to state requirements but, to drive design, assess feedback and, track progress. To spell this last point out, *quantification of the requirements* (what do we want to control?) is only a first step in getting control. The next steps, based on this quantification, are *design estimation* (how good do we think our solutions are?) and *measurement of the delivered results* (how good were the solutions in practice?). The key issue here is the active use of quantified data (requirements, design estimates and feedback) to drive the project design and planning.

One radical conclusion to draw, from this lack of quantification, is that current conventional agile methods are not really suitable for development of industrial products. The rationale for this being that industry is not simply interested in delivered 'functionality' alone; they probably already have necessary business functions at some level. Projects must produce competitive products, which means projects must deliver specific performance levels (including qualities and savings). To address this situation, it is essential that the *explicit* notion of quantification be added to agile concepts.

See Figure 2 for a list of the benefits to agile development of using quantification.

**Benefits of the Use of Quantification in Agile Development**

- Simplify requirements (if the top few requirements are quantified, there is less need for copious documentation as the developers are focused on a clearer, simpler 'message');
- Communicate quality goals much better to all parties (that is, users, customers, project management, developers, testers, and lawyers);
- Contract for results. Pay for results only (not effort expended). Reward teams for results achieved. This is possible as success is now measurable;
- Motivate technical people to focus on real business results;
- Evaluate solutions/designs/architectures against the quantified quality requirements;
- Measure evolutionary project progress towards quality goals and get early & continuous improved estimates for time to completion;
- Collect numeric historical data about designs, processes, organizational structures for future use. Use the data to obtain an understanding of your process efficiency, to bid for funding for improvements and to benchmark against similar organizations!

Figure 2. What can we do better in agile development (or 'at all'), if we quantify requirements

**Defining Quality**

The main focus for discussion in this article will be the quality characteristics, because that is where most people have problems with quantification. A long held opinion of one of the authors of this article (Tom Gilb) is that all qualities are capable of being expressed quantitatively (see Figure 3).

The Principle Of 'Quality Quantification'
*All qualities can be expressed quantitatively, 'qualitative' does not mean unmeasurable.*
Tom Gilb

Figure 3. Tom Gilb's opinion that all qualities can be expressed numerically

A Planguage definition of 'quality' is given in Figure 4. Planguage is a planning language and a set of methods developed by Tom Gilb over the last three decades (Gilb 2005). This next part of the article will outline the Planguage approach to specifying and using quantitative requirements to drive design and determine project progress.

**Definition of Quality**

Quality is characterized by these traits:

- A quality describes 'how well' a function is done. Qualities each describe the partial effectiveness of a function (as do all other performance attributes).
- Relevant qualities are either valued to some degree by some stakeholders of the system - or they are not relevant. Stakeholders generally value more quality, especially if the increase is free, or lower cost, than the stakeholder-perceived value of the increase.
- Quality attributes can be articulated independently of the particular means (the designs and architectures) used for reaching a specific quality level, even though achievement of all quality levels depend on the particular designs used to achieve quality.
- A particular quality can potentially be a described in terms of a complex concept, consisting of multiple elementary quality concepts, for example, 'Love is a many-splendored thing!'
- Quality is *variable* (along a definable scale of measure: as are all scalar attributes).
- Quality levels are capable of being specified *quantitatively* (as are *all* scalar attributes).
- Quality levels can be *measured* in practice.
- Quality levels can be *traded off* to some degree; with other system attributes valued more by stakeholders.
- Quality can never be perfect (no fault and no cost) in the real world. There are some valued levels of a particular quality that may be outside the state of the art at a defined future time and circumstance. When quality levels increase towards perfection, the resources needed to support those levels tend towards infinity.

(Gilb 2005)

Figure 4. Planguage definition of 'quality'

**Quantifying Requirements**

Planguage enables capture of quantitative data (metrics) for performance and resource requirements. A scalar requirement, that is, either a performance or resource requirement, is specified by identifying a relevant scale of measure and stating the current and required levels on that scale. See Figure 5, which is an example of a performance requirement specification. Notice the parameters used to specify the levels on the scale that is, Past, Goal. And Fail.

Tag:
Scale:
Meter:
Past:
Goal:
Fail:

Figure 5. Planguage parameters used to specify a performance requirement

## Evaluating Designs

Impact Estimation (IE) is the Planguage method for evaluating designs. See Table 1, which shows an example of a simple IE table. The key idea of an IE table is to put the potential design ideas against the quantified requirements and estimate the impact of each design on each of the requirements. If the current level of a requirement is known (its baseline, 0%), and the target level is known (its goal or budget depending on whether a performance requirement (an objective) or a resource requirement respectively, 100%), then the percentage impact of the design in moving towards the performance/resource target can be calculated. Because the values are converted into percentages, then simple arithmetic is possible to calculate the cumulative effect of a design idea (sum of performance and sum of cost) and the performance to cost ratio (see Table 1). You can also sum across the designs (assuming the designs are capable of being implemented together and that their impacts don't cancel each other out) to see how much design you have that is addressing an individual requirement.

Table 1 also shows how you can take into account any uncertainties in your estimates. An additional feature, not shown here, is to assess the credibility of each estimate by assigning a credibility factor between 0.0 and 1.0. Each estimate can then be multiplied by its credibility factor to moderate it.

While such simple arithmetic does not represent the complete picture, it does give a convenient means of quickly identifying the most promising design ideas. Simply filling in an IE table gives a much better understanding of the strengths and weaknesses of the various designs with respect to meeting all the requirements.

Advertisement – Visual Use Case - Click on ad to reach advertiser web site

Document Requirements precisely with "Structured Pseudo-Code"

...and then with a single click...

...convert it into a "Flow Chart" that everyone can understand!

Visual Use Case™ 2006

Write better Use Cases in less time. Multiply Productivity, Improve Software.

Click here to download your instant free trial of Visual Use Case.
http://www.TechnoSolutions.com

| Design Ideas-><br><br>Requirements:<br>Goals and Budgets | Idea 1<br>Impact<br>Estimates | Idea 2<br>Impact<br>Estimates | Sum for<br>Requirement/<br>(Sum of<br>Percentage<br>Impacts) | Sum of<br>Percentage<br>Uncertainty<br>Values | Safety<br>Deviation |
|---|---|---|---|---|---|
| Reliability<br>300 <-> 3000 hours MTBF | 1650hr<br>±0 | 840hr<br>±240 | 92% | ±9% | -108% |
| | 61%±0 | 31%±9% | | | |
| Usability<br>20 <-> 10 minutes | 1min.<br>±4 | 6 min.<br>±9 | 70% | ±130% | -130% |
| | 10%±40% | 60%±90% | | | |
| Sum of Performance | 71% | 91% | | | |
| Capital<br>0 <-> 1 million US$ | 500K<br>±200K | 100K<br>±200K | 60% | ±40% | -10% |
| | 50%±20 | 10%±20 | | | |
| Maintenance<br>1.1M <-> 100K/year US$ | 0 K$/Y<br>±180K | 1 M$/Y<br>±720K | 100% | ±90% | -50% |
| | 0%± 18% | 100%±72% | | | |
| Sum of Cost | 50% | 110% | | | |
| Performance to Cost Ratio | 1.42<br>(71/50) | 0.83<br>(91/110) | | | |

Table 1. An example of a simple IE table (Gilb 2005)

Table 1 simply shows estimates for potential design ideas. However, you can also input the actual measurements (feedback) from implementing the design ideas. There are two benefits to this: you learn how good your estimates where for the design ideas implemented, and you learn how much progress you have made towards your target levels. You can then use all the IE table data as a basis to decide what to implement next.

## Evolutionary Delivery

The final Planguage method we will discuss is Evolutionary Project Management (Evo). Evo demands include the following:

- that a system is developed in a series of small increments (each increment typically taking between 2% and 5% of the total project timescale to develop);
- that each increment is delivered for real use (maybe as Beta or Field trial) by real ‘users’ (any stakeholder) as early as possible (to obtain business benefits, and feedback, as soon as possible).
- that the feedback from implementing the Evo steps is used to decide on the contents of the next Evo step;
- that the highest value Evo steps are delivered earliest, to maximize the business benefit.

Note that ‘*delivery*’ of requirements is the key consideration. Each delivery is done within an Evo step. It may, or may not, include the building or creation of the increment (Some Evo steps may simply be further roll-out of *existing* software);

Development of necessary components will occur incrementally, and will be continuing in parallel while Evo steps are being delivered to stakeholders. Most development will only start when the decision has been taken to deliver it as the next Evo step. However, there probably will be some increments that have longer lead-times for development, and so their development will need to start early in anticipation of their future use. A project manager should always aim to 'buffer' his developers in case of any development problems by having in reserve some components (readied in the 'Backroom') ready for delivery. The 'Frontroom' being the term for the interface between developers and stakeholders – for implementation of the steps.

## Planguage approach to Change

It is important to note that the quantified requirements, designs and implementation plans are not 'frozen,' they must be subject to negotiated change, over time. As Beck points out, "Everything in software changes. The requirements change.  The design changes. The business changes. The technology changes. The team changes. … The problem isn't change, per se, … the problem, rather, is the inability to cope with change when it comes" (Beck 2000).

Planguage's means of dealing with change is as follows:

- performance and resource requirements are quantified to allow rapid communication of any changes in levels;
- IE tables allows dynamic reprioritization of design ideas and helps track progress towards targets;
- Evo enables all types of change to be catered for 'in-flight', as soon as possible. There is regular monitoring of what the best next Evo step to take.

## Description of the Planguage process

To summarize and show how the methods (for quantifying requirements, evaluating designs and evolutionary delivery) described earlier in this article fit together, here is a description of the Planguage process for a project:

1. Gather from all the key stakeholders the top few (5 to 20) most critical goals that the project needs to deliver. Give each goal a reference name (a tag).
2. For each goal, define a scale of measure and a 'final' goal level. For example:

   *Reliable:*

   *Scale: Mean Time Before Failure,*

   *Goal:  1 month.*
3. Define approximately 4 budgets for your most limited resources (for example, time, people, money, and equipment).
4. Write up these plans for the goals and budgets (*Try to ensure this is kept to only one page*).
5. Negotiate with the key stakeholders to formally agree the goals and budgets.
6. Draw up a list of design ideas: Ensure that you decompose the design ideas down into the smallest increments that can be delivered (these are potential Evo steps). Use Impact Estimation (IE) to evaluate your design ideas contributions towards meeting the requirements. Look for small increments with large business value. Note any dependencies, and draw up an initial rough Evo plan, which sequences the Evo steps. In practice, decisions about what to deliver for the next Evo step will be made in the light of feedback (that is when the results from the deliveries of the previous Evo steps are known). Plan to deliver

some value (that is, progress towards the required goals) in *weekly* (or shorter) increments (Evo steps). Aim to deliver highest possible value as soon as possible.

7. Deliver the project in Evo steps.
   - Report to project sponsors after each Evo step (weekly, or shorter) with your best available estimates or measures, for each performance goal and each resource budget. O*n a single page,* summarize the *progress to date* towards achieving the goals and the costs incurred.
   - Discuss with your project sponsors and stakeholders what design ideas you should deliver in the next Evo step. This should be done in the light of what has been achieved to date and what is left to do. Maximizing the business benefit should be the main aim.
8. When all goals are reached: 'Claim success and move on.' Free remaining resources for more profitable ventures

**Ten Planguage Values for an Agile Project**

*Simplicity*

1. Focus on real stakeholder values.

*Communication*

2. Communicate stakeholder values quantitatively.
3. Estimate expected results and costs for weekly steps.

*Feedback*

4. Generate useful results, weekly, to stakeholders, in their environment.
5. Measure all critical aspects of the attempt to generate incremental results.
6. Analyze deviation from initial estimates.

*Courage*

7. Change plans to reflect weekly learning.
8. Immediately implement valued stakeholder needs, next week.

*Don't wait, don't study ('analysis paralysis') and, don't make excuses. Just Do It!*

9. Tell stakeholders exactly what you will deliver next week.
10. Use any design, strategy, method, process that works quantitatively well - to get your results. Be a *systems engineer,* not a just programmer. Do not be limited by your craft background, in serving your paymasters

Figure 6. Planguage's ten values for an agile project based around Beck's Four Values for XP (Beck 2000 Page 29)

**Planguage Project Management Policy**

- The project manager, and the project, will be judged exclusively on the relationship of progress towards achieving the goals versus the amounts of the budgets used.
- The project team will do anything legal and ethical to deliver the goal levels within the budgets.
- The team will be paid and rewarded for benefits delivered in relation to cost.
- The team will find their own work process and their own design.
- As experience dictates, the team will be free to suggest to the project sponsors (stakeholders) adjustments to ‘more realistic levels’ of the goals and budgets.

Figure 7. Planguage policy for project management

## Case Study of The ‘Confirmit’ Product

Tom Gilb and his son, Kai taught the Planguage methods to the FIRM (Future Information Research Management, in Norway) organization. Subsequently, FIRM used these methods in the development of their Confirmit product. The results were impressive, so much so that they decided to write up about their experiences (Johansen 2004, Johansen and Gilb 2005). In this section, some of the details from this Confirmit product development project are presented.

## Use of Planguage Methods

First, the quantified requirements were specified, including the target levels. Next, a list of design ideas (solutions) was drawn up (see Figure 8 for an example of an initial design idea specification).

Recoding:
Type: Design Idea [Confirmit 8.5].
Description: Make it possible to recode a marketing variable, on the fly, from Reportal.
Estimated effort: 4 team days.

Figure 8. A brief specification of the design idea, 'Recoding'

The impacts of the design ideas on the requirements were then estimated. The most promising design ideas were included in an Evo plan, which was presented using an Impact Estimation (IE) table (see Tables 2 and 3, which show the part of the IE table applying to Evo Step 9. Note these tables also include the actual results after implementation of step 9). The design ideas were evaluated with respect to 'value for clients' versus 'cost of implementation'. The ones with the highest value-to-cost ratio were chosen for implementation in the early Evo steps. Note that value can sometimes be defined by *risk removal* (that is, implementing a technically challenging solution early can be considered high value if implementation means that the risk is likely to be subsequently better understood). The aim was to deliver improvements to real external stakeholders (customers, users), or at least to internal stakeholders (for example, delivering to internal support people, who use the system daily and so can act as 'clients').

| | EVO STEP 9: DESIGN IDEA: 'Recoding' | | | |
|---|---|---|---|---|
| | Estimated Scale Level | Estimated % Impact | Actual Scale Level | Actual % Impact |
| REQUIREMENTS | | | | |
| Objectives | | | | |
| Usability.Productivity<br>65 <-> 25 minutes<br><br>Past: 65 minutes.<br>Tolerable: 35 minutes.<br>Goal: 25 minutes. | 65 – 20 =<br>45 minutes | 50% | 65 - 38 =<br>27 minutes | 95% |
| Resources | | | | |
| Development Cost<br>0 <-> 110 days | 4 days | 3.64% | 4 days | 3.64% |

Table 2. A simplified version of part of the IE table shown in Table 3.

It only shows the objective, 'Productivity' and the resource, 'Development Cost' for Evo Step 9, 'Recoding' of the Marketing Research (MR) project. The aim in this table is to show some extra data, and some detail of the IE calculations. Notice the separation of the requirement definitions for the objectives and the resources. The Planguage keyed icon '<->' means 'from baseline to target level'. On implementation, Evo Step 9 alone moved the Productivity level to 27 minutes, or 95% of the way to the target level

The IE table was used as a tool for controlling the qualities: estimated figures and actual measurements were input into it. Each next Evo step was then decided, based on the results achieved *after* implementation and delivery of the subsequent step. Note, the results were *not* actually measured with statistical accuracy by doing a scientifically correct large-scale survey (although FIRM are currently considering doing this). The impacts described for Confirmit 8.0

(the 'Past' levels) are based on internal usability tests, productivity tests, performance tests carried out at Microsoft Windows ISV laboratory in Redmond USA, and from direct customer feedback.

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

Table 3. Details of the real IE table, which was simplified in Table 2.

The two requirements expanded in Table 1 are highlighted in bold. The 112.5 % improvement result represents a 20 minutes level achieved after the initial 4 day stint (which landed at 27 minutes, 95%) . A few extra hours were used to move from 27 to 20 minutes, rather than use the next weekly cycle.

## The Results Achieved

Due to the adoption of Evo methods there were focused improvements in the product quality levels. See Table 4, which gives some highlights of the 25 final quality levels achieved for Confirmit 8.5. See also Table 5, which gives an overview of the improvements by function (that is, product component) for Confirmit 9.0. No negative impacts are hidden. The targets were largely all achieved on time.

| DESCRIPTION OF REQUIREMENT / WORK TASK | PAST | CURRENT STATUS |
|---|---|---|
| Usability.Productivity: Time for the system to generate a survey | 7200 sec | 15 sec |
| Usability.Productivity: Time to set up a typical specified Market Research (MR) report | 65 min | 20 min |
| Usability.Productivity: Time to grant a set of End-users access to a Report set and distribute report login info. | 80 min | 5 min |
| Usability.Intuitiveness: The time in minutes it takes a medium experienced programmer to define a complete and correct data transfer definition with Confirmit Web Services without any user documentation or any other aid | 15 min | 5 min |
| Workload Capacity.Runtime.Concurrency: Maximum number of simultaneous respondents executing a survey with a click rate of 20 seconds and a response time < 500 milliseconds, given a defined [Survey-Complexity] and a defined [Server Configuration, Typical]. | 250 users | 6000 users |

Table 4. Improvements to product quality levels in Confirmit 8.5

| FUNCTION | PRODUCT QUALITY | DEFINITION (quantification) | CUSTOMER VALUE |
|---|---|---|---|
| Authoring | Intuitiveness | Probability that an inexperienced user can intuitively figure out how to set up a defined Simple Survey correctly. | Probability increased by 175% (30% to 80%) |
| Authoring | Productivity | Time in minutes for a defined advanced user, with full knowledge of Confirmit 9.0 functionality, to set up a defined advanced survey correctly. | Time reduced by 38% |
| Reportal | Performance | Number of responses a database can contain if the generation of a defined table should be run in 5 seconds. | Number of responses increased by 1400% |
| Survey Engine | Productivity | Time in minutes to test a defined survey and identify 4 inserted script errors, starting from when the questionnaire is finished to the time testing is complete and ready for production. (Defined Survey: Complex Survey, 60 questions, comprehensive JScripting.) | Time reduced by 83% and error tracking increased by 25% |
| Panel Management | Performance | Maximum number of panelists that the system can support without exceeding a defined time for the defined task, with all components of the panel system performing acceptably. | Number of panelists increased by 1500% |
| Panel Management | Scalability | Ability to accomplish a bulk-update of X panelists within a timeframe of Z seconds. | Number of panelists increased by 700% |
| Panel Management | Intuitiveness | Probability that a defined inexperienced user can intuitively figure out how to do a defined set of tasks correctly. | Probability increased by 130% |

Table 5. Some detailed results by function (product component) for Confirmit 9.0

The customers responded very favorably (see Figure 9).

*"I just wanted to let you know how appreciative we are of the new 'entire report' export functionality you recently incorporated into the Reportal. It produces a fantastic looking report, and the table of contents is a wonderful feature. It is also a HUGE time saver."*

Figure 9. An example of pilot customer (Microsoft) feedback

On the *second* release (Confirmit 9.0) using Planguage, and specifically the Evo method, the Vice President (VP) of Marketing proudly named the Evo development method on the FIRM website (see Figure 10. A line executive bragging about a development method is somewhat exceptional!).

"FIRM, through evolutionary development, is able to substantially increase customer value by focusing on key product qualities important for clients and by continuously asking for their feedback throughout the development period. Confirmit is used by the leading market research agencies worldwide and Global 1000 companies, and together, we have defined the future of online surveying and reporting, represented with the Confirmit 9.0."

Figure 10. Comments by FIRM's VP of Marketing, Kjell Øksendal

Details of the quantified improvements were also given to their customers (see Figure 11, which is an extract from the product release for Confirmit 9.0 published on the organization's website).

News release

2004-11-29: Press Release from FIRM

New version of Confirmit increases user productivity up to 80 percent

NOVEMBER 29th, 2004: FIRM, the world's leading provider of online survey & reporting software, today announced the release of a new version of Confirmit delivering substantial value to customers including increased user productivity of up to 80 percent.

FIRM is using Evolutionary (EVO) development to ensure the highest focus on customer value through early and continuous feedback from stakeholders. A key component of EVO is measuring the effect new and improved product qualities have on customer value. Increased customer value in Confirmit 9.0 includes:

* Up to 175 percent more intuitive user interface*
* Up to 80 percent increased user productivity in questionnaire design and testing*
* Up to 1500 percent increased performance in Reportal and Panel Management*

Figure 11. Confirmit 9.0 release announcement from the FIRM website http://www.firmglobal.com. It gives detail about the method and the quantified product results

## Impact on the developers

Use of Evo has resulted in increased *motivation* and *enthusiasm* amongst the FIRM developers, because it has opened up *'empowered creativity' (Trond Johansen, FIRM Project Director).* The developers can now determine their own design ideas, and are not subject to being dictated the design ideas by marketing and/or customers, who often tend to be amateur technical designers. Daily, and more often, product builds, called Continuous Integration (CI, using Cruise Control), were introduced. Evo combined with CI, is seen as a vehicle for innovation and inspiration. Every week, the developers get their work out onto the test servers, and receive feedback.

By May 2005, FIRM had adopted the approach of using a 'Green Week' once monthly. In a Green Week, the internal stakeholders are given precedence over the client stakeholders and can choose what product improvements they would like to see implemented. The FIRM developers chose to focus on the evolutionary improvement of about 12 internal stakeholder qualities (such as testability and maintainability).

## Initial difficulties in implementing Planguage

Even though Planguage was embraced, there were parts of Planguage that were initially difficult to understand and execute at first. These included:

- Defining good requirements ('Scales' of measure) sometimes proved hard; (they only had one day training initially, but after the first release saw the value in a weeks training!)
- It was hard to find 'Meters' (that is, ways of measuring numeric qualities, to test the current developing quality levels), which were practical to use, and at the same time measured real product qualities;
- Sometimes it took more than a week to deliver something of value to the client; (this was mainly a test synchronization problem they quickly overcame).
- Testing was sometimes 'postponed' in order to start the next step. Some of these test postponements were then not in fact done in later testing.

## Lessons learned with respect to Planguage, especially the Evo method

Some of the lessons learnt about the use of Planguage, and especially the Evo method, included:

- Planguage places a focus on the measurable product qualities. Defining these clearly and testably requires training and maturity. It is important to *believe* that everything can be measured and to seek guidance if it seems impossible;
- Evo demands dynamic re-prioritization of the next development steps using the ratio of delivering value for clients versus the cost of implementation. Data to achieve this is supplied by the weekly feedback. The greatest surprise was the power of focusing on these ratios. What seemed important at the start of the project may be replaced by other solutions based on gained knowledge from previous steps;
- an *open architecture* is a pre-requisite for Evo;
- management support for changing the software development process is another pre-requisite, but this is true of any software process improvement;
- The concept of daily builds, CI, was valuable with respect to delivering a new version of the software every week;
- It is important to control expectations. ‘Be humble in your promises, but overwhelming in your delivery’ is a good maxim to adopt;
- There needed to be increased focus on feedback from clients. The customers willing to dedicate time to providing feedback need identifying. Internal stakeholders (like sales and help desk staff) can give valuable feedback, but some interaction with the actual customers is necessary;
- Demonstrate new functionality automatically, with screen recording software or early test plans. This makes it easier for internal and external stakeholders to do early testing;
- Tighter integration between Evo and the test process is necessary.

## Conclusions of the Case Study

The positive impacts achieved on the Confirmit product qualities has proved that the Evo process is better suited than the Waterfall process (used formerly) to developing the Confirmit product.

Overall, the whole FIRM organization embraced Planguage, especially Evo. The first release, Confirmit 8.5 showed some of Planguage’s great potential. By the end of November 2004, with the second release (Confirmit 9.0), there was confirmation that the Evo method can, consistently and repetitively, produce the results needed for a competitive product. Releases 9.5 and 10.0 of Confirmit continued this pattern of successful product improvements delivered to the customers (as of November 2005).

It is expected that the next versions of Confirmit will show even greater maturity in the understanding and execution of Planguage. The plan is to continue to use Planguage (Evo) in the future.

## Article Summary

Use of quantified requirements throughout the implementation of a project can provide many benefits as has been demonstrated by the FIRM organization’s use of Planguage (including Evo).

The key messages of this article can be summarized in twelve Planguage principles (see Figure 12). By adopting such principles, agile methods would be much better suited for use in the development of industrial products.

**Twelve Planguage Principles**

1. Control projects by a small set of quantified critical results (that is, not stories, functions, features, use cases, objects, etc.). Aim for them to be stated on one page!
2. Make sure those results are *business* results, not technical.
3. Align your project with your financial sponsor's interests!
4. Identify a set of designs. Ensure you decompose the designs into increments of the smallest possible deliverables.
5. Estimate the impacts of your designs, on *your* quantified goals.
6. Select designs with the best performance to cost ratios; do them first.
7. Decompose the workflow and/or deliveries, into weekly (or 2% of budget) time boxes.
8. Give developers freedom, to find out *how* to deliver those results.
9. Change designs, based on quantified experience of implementation (feedback).
10. Change requirements, based in quantified experience (new inputs).
11. Involve the stakeholders, every week, in setting quantified goals.
12. Involve the stakeholders, every week, in *actually using* increments.

Figure 12. Twelve Gilb Planguage principles for project management/software development.

## References

Agile Alliance, 2006, URL: http://www.agilealliance.com/.

Beck, Kent, *Extreme Programming Explained: Embrace Change*, 2000, Addison-Wesley. ISBN 0201616416.

Gilb, Tom, *Competitive Engineering: A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage*, 2005, Elsevier Butterworth-Heinemann, ISBN 0750665076.

Johansen, Trond, FIRM: From Waterfall to Evolutionary Development (Evo) or How we rapidly created faster, more user-friendly, and more productive software products for a competitive multi-national market, *Proceedings of European Software Process Improvement (EuroSPI)*, Trondheim, Norway, November 10-12, 2004, Torgeir Dingsøyr (Ed.), Lecture Notes in Computer Science 3281, Springer 2004, ISBN 3-540-23725-9. See also Proceedings of INCOSE 2005 (Johansen and Gilb 2005) and FIRM website, http://www.confirmit.com/news/release_20041129_confirmit_9.0_mr.asp/.

# Mocking the Embedded World: Test-Driven Development, Continuous Integration, and Design Patterns

Michael Karlesky, Greg Williams, William Bereza, Matt Fletcher
Atomic Object, 941 Wealthy Street SE, Grand Rapids, MI 49506
http://atomicobject.com

Despite a prevalent industry perception to the contrary, the agile practices of Test-Driven Development and Continuous Integration can be successfully applied to embedded software. We present here a holistic set of practices, platform independent tools, and a new design pattern (Model Conductor Hardware - MCH) that together produce: good design from tests programmed first, logic decoupled from hardware, and systems testable under automation. Ultimately, this approach yields an order of magnitude or more reduction in software flaws, predictable progress, and measurable velocity for data-driven project management. We use the approach discussed herein for real-world production systems and have included a full C-based sample project (using an Atmel AT91SAM7X ARM7) to illustrate it (see Appendix). This example demonstrates transforming requirements into test code; system, integration, and unit tests driving development; daily "micro design" fleshing out a system's architecture; the use of MCH itself; and the introduction of mock functions to automated unit tests.

## Introduction

Heads around the table nodded, and our small audience listened to us thoughtfully. Atomic Object had been invited to lunch with a potential client. We delivered our standard introduction to Agile [1] software development methods. While it was clear there was not complete buy-in of the presented ideas, most in the room saw value, agreed with the basic premises, or wanted to experiment with the practices. And then we heard an objection we had never heard before: "Boy, guys, it sounds great, but you can't do this with firmware code because it's so close to the hardware." Conversation and questions were replaced with folded arms and furrowed brows. It was the first time we had had in-depth conversation with hardcore embedded software developers.

In standard Atomic Object fashion, we took our experience as a dare to accomplish the seemingly impossible. We went on to apply to embedded software what we knew from experience to be very effective and complementary techniques – in particular Test-Driven Development (TDD) and Continuous Integration (CI). Inspired by an existing design pattern, we discovered and refined an approach to enable automated system tests, integration tests, and unit tests in embedded software and created a small C-based test framework [2]. As we tackled additional embedded projects, we further refined these methods, created a scriptable hardware-based system test fixture, developed the means to auto-generate mock functions for our integration tests, wrote scripts to generate code skeletons for our production code, and tied it all together with an automated build system.

The driving motivation for our approach is eliminating bugs as early as possible and providing predictable development for risk management. We know by experience that the methods we discuss here reduce software flaws by an order of magnitude or more over the average [4]. They also allow a development team to react quickly and effectively to changes in the underlying hardware or system requirements. Further, because technical debt [3] is virtually eliminated, progress can be measured and used in project management. A recent case study of a real-world, 3 year long, Agile embedded project found the team out-performed 95th percentile, "best in class" development teams [4]. Practices such as Test-Driven Development (TDD) and Continuous Integration (CI) are to thank for such results. Within embedded software circles, the practices of TDD and CI are either unknown or have been dismissed with skepticism. The direct interaction of programming and hardware as well as limited resources for running test frameworks seem to set a hurdle too high to clear. Our approach has been successfully applied in systems as small as 8 bit microcontrollers with 256 bytes of RAM and scales up easily to benefit complex, heavy-duty systems.

Application of these principles and techniques does not incur extra cost. Rather, this approach drastically reduces final debugging and verification that often breaks project timelines and budgets. Bugs found early are less costly to correct than those found later. Technical debt is prevented along the way, shifting the time usually necessary for final integration and debugging mysteries to developing well-tested code prior to product release. Because code is well-tested and steadily and predictably added to the system, developers and managers can make informed adjustments to priorities, budgets, timelines, and features well before final release. In avoiding recalls due to defects and producing source code that is easy to maintain and extend (by virtue of test suites), the total software lifecycle is less costly than most if not all projects developed without these practices.

## A Note on "Mocking" and This Article's Title

Mocking in software development is a specific practice that complements unit testing (in particular, interaction-based testing). The majority of a system's code consists of calls making calls to other parts of the codebase. A mock is a specialized substitution for any part of the system with which the code under test interacts.

Advertisement – MKS Integrity - Click on ad to reach advertiser web site

**I AM** A DEVELOPMENT MANAGER.
**I OVERSEE** PROJECTS AROUND THE GLOBE.
**I NEED MKS.**

My team is in four different locations, working with millions of lines of code and thousands of changes. They need to be productive and efficient.

My CIO needs assurance that my team's work aligns to the needs of the business. He needs alignment and compliance.

**I need ONE solution for application lifecycle management.**
**I need MKS.**

Its one architecture, one solution, total visibility platform delivers new levels of productivity, efficiency and process automation across our entire IT organization.

With MKS, we are one.

***Coming soon: MKS Integrity 2007.***
*Requirements reuse.*
*Test management.*
*SAP and PeopleSoft/Oracle change management.*
*Visit our web site www.mks.com for more details.*

MKS
APPLICATION LIFECYCLE MANAGEMENT
FOR THE ENTERPRISE

The mock not only mimics the function call interface of the system code outside the code under test it also provides the means to capture the parameters of function calls made upon it, record the order of calls made, and provide any function return value a programmer requires for testing scenarios. With mocks we can thoroughly test all of the logic within a function and verify that this code makes calls to the rest of the system as expected. Mocking is covered in more depth later.

Automated unit testing is far more prevalent in high-level software systems than in embedded systems though certainly even here it is not widespread. To our knowledge, automatically generating and unit testing with mocks in embedded software (particularly in small systems and those using C) such as we have done is a new development in the embedded space. This article's title is a play on the uniqueness of the mocking concept to embedded software and a reaction to those in the industry that may say practices such as TDD are impossible to implement or have no value in embedded software development.

## The Value of TDD and CI

Test-Driven Development and Continuous Integration are complementary practices. Code produced test-first tends to be well designed and relatively easy to integrate with other code. Incrementally adding small pieces of a system to a central source code control system ensures the whole system compiles without extensive integration work. Running tests allows developers to find integration problems early as new code is added to the system. An automated build system complemented by regression test suites ensures a system grows responsibly in features and size and exists in a near ready-to-release fashion at all times.

## Test-Driven Development Overview

Traditional testing strategies rarely impact the design of production code, are onerous for developers and testers, and often leave testing to the end of a project where budget and time constraints threaten thorough testing. Test-Driven Development systematically inverts these patterns. In TDD, development is not writing all the functional code and then later testing it, nor is it verifying code by stepping through it with a debugger. Instead, testing drives development. A developer looks for ways to make the system testable, does a small amount of design, writes test programming for the piece of the system currently under development, and then writes functional code to meet the requirements of the test-spawned design. Designing for testability in TDD is a higher calling than designing "good" code because testable code *is* good code.

At the highest levels (e.g. integration and system testing) fully automated testing is unusual. However, at the lowest level, automated unit testing is quite possible. In automated unit testing, a developer first writes a unit test (a test that validates correct operation of a single module of source code – for instance, a function or method) and then implements the complementary functional code. With each system feature tackled, unit test code is added to an automated test suite. Full regression tests can take place all the time. Further high-level integration or system testing will complement these unit tests and ideally will include some measure of automation.

System Test-Driven Development follows these steps:

1. Pick a system feature.
2. Program a system test to verify that feature.
3. Compile; run the system test with the system itself and see it fail.
    4. Identify a piece of functionality within the feature (a single function or method).

    5. Program integration and unit tests to verify that functionality.
    6. Stub out the functional code under test (to allow the test code to compile).
    7. Compile; run the integration and unit tests and see them fail (to verify expectations).
    8. Flesh out the functional, production code.
        9. Compile; run the integration and unit tests.
        10. Refactor the production code.
        11. Repeat 9-10 until the integration and unit tests pass and the functional code is cleanly implemented.
    12. Compile; run the system test.
    13. Repeat 4-12 until the system test passes.
14. Repeat 1-13 until all features of the system are implemented.

TDD provides several clear benefits:

- Code is always tested.
- Testing drives the design of the code. As a side effect, the code is well designed because of the decoupling necessary to create testable code.
- The system grows organically as more knowledge of the system is gained.
- The knowledge of the system is captured in tests; the tests are “living” documentation.
- Developers can add new features or alter existing code with confidence that automated regression testing will reveal failures and unexpected results and interactions.
- Tests catch the majority of bugs and leave for a human mind difficult testing issues like timing collisions or unexpected sub-system interactions.

## Continuous Integration Overview

The technique of continuous integration regularly brings together a system’s code (possibly from multiple developers) and ensures via regression tests that new programming has not broken existing programming. Automated build systems allow source code and tests to be compiled and run automatically. These ideas and tools are important complements to effective TDD. When TDD and CI are used together the system’s code-base is always thoroughly tested and has few, if any, integration problems among subsystems or sections of code. Integration problems are discovered early when it is cheapest to correct them. Further, any such problem will be discovered close to where and when the problem was created; here, understanding is greatest and good design choices are most likely.

## Particular Advantages of TDD and CI in Embedded Software

In the context of embedded software, TDD and CI provide two further advantages beyond those already discussed. First, because of the variability of hardware and software during development, bugs are due to hardware, software, or a combination of the two. TDD and CI promote a strong separation of concerns such that it becomes far easier to pinpoint, by process of elimination, the source of unexpected system behavior. Well-tested software can generally be eliminated from the equation or, in fact, used to identify hardware issues. Second, because these techniques encourage good decoupling of hardware and software, significant development can occur without target hardware.

## Russian Dolls & Our Embedded Software Development Approach

A set of Russian dolls comprises individual dolls of decreasing size and corresponding detail nested one inside another. Our approach fleshes out a system's architecture with Russian doll-like levels of test-driven design. The architecture of a system and its requirements drive system tests. System tests in turn drive integration tests. Integration tests drive unit tests. Each nested level of executable testing drives the design of the production code that will satisfy it. Implementing testable code forces a series of small design decisions at the keyboard. The result is a system aggregated of high quality, thoroughly tested pieces nested together to support the overall architecture and satisfy the system requirements.

In this section, we provide background on specialized techniques we employ, discuss the tools supporting our approach, and finally present a summary of the steps to implement a single system feature from start to finish. Our techniques and tools are synergistic; each supports and enhances the others. The paper concludes with an in-depth discussion of the Model Conductor Hardware design pattern introduced in this section and an end-to-end working example with tests, mocks, and source code.

## Techniques

### System, Integration, and Unit Testing

Requirements are composed of one or more features. We satisfy requirements by implementing features. Each feature necessitates creating a system test that will exercise and verify it once it exists. This system test will operate externally to the system under test. If pressing a button is to generate bytes on a bus, a system test specifies in programming the initiation of the button signal and the verification of the bytes on the bus.

A single system feature is composed of one or more individual functions. We begin creating each of these functions by programming integration tests to verify the interrelation of function calls. Subsequent unit tests verify the output of each function under various input conditions. After creating the integration and unit tests, we run them and see them fail. Next, we write the production code that will satisfy these tests. Tests and source are optionally refactored until the code is clean and the tests pass. Integration and unit tests can be run on target hardware, cross-compiled and run on the development machine, or run in a simulator, depending on the development environment and target system.

### Interaction-based Testing & Mock Functions

Few source code functions operate in isolation; most make calls on other functions. The composition of inter-function relationships, in large part, constitutes the implementation of a system feature. Testing this interaction is important. Mock functions facilitate testing these interactions and have become a key component of our development at the integration level. We practice interaction-based testing for integration tests and state-based testing for unit tests [5].

Whenever a function's work requires it to use another complex function (where a complex function is one requiring its own set of tests), we mock out that helper function and test against it. A mock presents the same interface to the code under test as the real module with which it interacts in production. Functionality within the mock allows tests to verify that only the expected calls with expected parameters were made against it in the expected order (and no unexpected calls were made). Further, a mock can produce any return results the tester-developer requires. This means that any scenario, even rare corner cases, can be verified in tests.

### Hardware and Logic Decoupling (Model Conductor Hardware design pattern)

Specialized hardware and accompanying programming interacting with it is the single greatest complication in thoroughly testing an embedded system. We discovered our Model Conductor Hardware (MCH) design pattern in the process of segregating and abstracting hardware from logic to enable automated testing. The Model, Conductor, and Hardware components each contain logically related functions and interact with one another according to defined rules. With these abstractions, divisions, and behaviors, the entire system can be unit and integration tested without direct manipulation of hardware. A later section of this paper provides a more in-depth explanation of MCH.

### Conductor First

A complete embedded system is composed of multiple groups of Model, Conductor, and Hardware components. A single interrelated group of these components is called an MCH triad. Each triad represents an atomic unit of system functionality. The Conductor member of an MCH triad contains the essential logic that conducts the events and interactions between the Hardware and Model comprising the functionality under test.

Conductor First (inspired by Presenter First [6,16]) is our approach to allow TDD to occur at the embedded software unit and integration level. We start by selecting a piece of system functionality within a system requirement. From this, we write integration and unit tests for the Conductor with a mock Hardware and a mock Model. Production code is then written to satisfy the Conductor tests.

Software Test & Performance CONFERENCE FALL
October 2–4, 2007
The Hyatt Regency Cambridge
Boston, MA
Register by Aug. 3 To Take Advantage of The Super Early-Bird Rate!
SUPERB SPEAKERS!
James Bach • Scott Barber • Rex Black
Michael Bolton • Clyneice Chaney
Jeff Feldstein • Robert Galen
Robin Goldsmith • Linda Hayes
Robert Sabourin • Mary Sweeney
Robert Walsh and dozens more!
TOTAL IMMERSION!
Choose From 70+ Classes
Pick From 8 In-Depth Tutorials
Hands-On Tool Demo Sessions
Network With Colleagues
Ice Cream Social
Reception in Demonstration Hall
Pose Your Questions To 25+ Exhibitors
Mingle With More Than 40 Speakers
Testing Secrets REVEALED!
register at www.stpcon.com

This technique allows us to discover the needed Hardware and Model interfaces. The Hardware and Model are then implemented in a similar fashion, beginning with tests; the Hardware and Model tests reveal the needed use of the physical hardware and the interface to other triad Models.

Starting in the Conductor guides development of function calls from the highest levels down to the lowest. Developing the Conductor with mocks allows the control logic necessary to satisfy the system requirement to be designed and tested with no coupling to the hardware or other system functionality. In this way, unnecessary infrastructure is not developed and system requirements are implemented as efficiently and quickly as possible.

## Tools

All of the following tools, with the exception of the miniLAB 1008 hardware test module, are freely available. Some are publicly available tools used by developers the world over. The others are custom tools we developed and have made available through our website packaged together with our sample project (improved and documented versions of these same tools will be made available soon).

Systir – System Test Framework
*Systir* [7] stands for "System Testing in Ruby." Ruby [8] is the reflective, dynamic, object-oriented scripting language Systir is built upon and extends. In TDD, we use Systir to introduce input to a system and compare the collected output to that which is expected in system tests. Systir builds on two powerful features of Ruby. First, Systir uses Ruby-based drivers that can easily bind to libraries of other languages providing practically any set of features needed in a system test (e.g. proprietary communication libraries). Second, Systir allows us to create Domain Specific Languages [9] helpful in expressing tests in human readable verbiage. We developed Systir for general system testing needs; it has also proven effective for end-to-end embedded systems testing.

Scriptable Hardware Test Fixture
System testing embedded projects requires simulating the real world. The miniLAB 1008 [10] is the device we have adopted as a hardware test fixture. It provides a variety of analog and digital I/O functions well suited to delivering input and collecting output of an embedded system under development. A proprietary library allows a PC to communicate with a miniLAB 1008 via USB. We developed a Ruby wrapper around this library to be used by Systir system tests. Other test hardware and function libraries (e.g. LabWindows/CVI ) could also be driven by Systir tests with the inclusion of new Ruby wrappers.

Source, Header, and Test File Code Generation
We decouple hardware from programming through the MCH design pattern. We also utilize interaction-based testing with mock functions. Both of these practices tend to require a greater number of files than development approaches not using MCH and interaction-based testing. At the same time, because of the pattern being used, we have very repeatable (i.e. automation friendly) file creation needs. To simplify the creation of source, header, and test files, we created a Ruby script to generate source, header, and test skeleton files.

Argent-based Active Code Generation
The skeleton files created by our file generation script contain pre-defined function stubs and header file include statements as well as Argent code insertion blocks. Argent [11] is a Ruby-based, text file processing tool that populates tagged blocks with the output of specified Ruby code. Our file generation script places Argent tags in the skeleton files that are later replaced

with C unit test and mock function management code necessary for all project test files.

Unity – Unit Test Framework for C
The mechanics of a test framework are relatively simple to implement [12]. A framework holds test code apart from functional code, provides functions for comparing expected and received results from the functional code under test, and collects and reports test results for the entire test suite. Unity [13] is a unit testing framework we developed for the C programming language. While there is a very small number of C unit test frameworks available, we found no good, lightweight framework we liked so we created our own. We customize Unity reporting per project (e.g. printing results through stdio, a serial port, or via simulator output).

CMock – Mock Function Library for C
CMock [13] is a Ruby-based tool we created to automate creation of mock functions for unit testing in the C language. CMock generates mocks from the functions defined in a project's header files. Each mock contains functionality for capturing and comparing calls made on the mock to expectations set in tests. CMock also allows tests to specify return results from functions within the mock. CMock alleviates the pain of creating and maintaining mocks; consequently, developers are motivated to make good design changes, since they need not worry about updating the mocks manually.

Dependency Generator & Link Time Substitution – File Linking Management for C
To facilitate linking of source files, test files, and mocks for testing or release, we developed a Ruby-based tool to manage these relationships automatically. This dependency tool inspects source and header files and assembles a list of files to be linked for testing or release mode.

In an object-oriented language, we would normally compose objects with delegate objects using a form of dependency injection [14]. Because C has no formalized notion of objects, constructor injection cannot be used. Instead, we substitute the CMock generated mock functions for real functions at link time.

Rake – Build Utility
Rake [15] is a freely available build tool written in Ruby ("Ruby make"). We create tasks in Rake files to compile and link a system under development, generate mocks with CMock, run Argent code generation for unit and integration testing, run our Unity unit test framework, and run our Systir system tests.

Subversion & CruiseControl.rb – Code Repository & Automated Build System
We use the freely available source code control system Subversion to manage our project files, source code, and unit and system test files. As tests and source code are implemented and those tests successfully pass, we check our work into Subversion. Upon doing so, CruiseControl.rb, an automated build system pulls down the latest version of the project from Subversion, builds it, runs its tests, and reports the results through a web-based interface. This process repeats itself upon every check in.

## Implementing a System Feature – An Overview

### Step 1 – Create One or More System Tests

The process begins with creating a system test to verify an aspect of a requirement with a specific test case. Some amount of Systir driver code will be necessary to establish the language of the system test and provide sufficient functionality (e.g. digital I/O, analog voltages, network traffic, etc.) to exercise the hardware of the embedded system. For direct hardware interaction, we make calls to our miniLAB wrapper. For data over communication channels (e.g. RS232 or Ethernet) calls to other libraries are necessary.

A typical system test generates a stimulus input and then gathers output from the board under development. Assertions comparing the expected and collected output will determine whether the system test passes or fails. Of course, at this point in the process, there no production code for this feature exists so this system test will fail until the feature is complete. The constraints of the system test, however, even at this early stage, will guide implementation and force disambiguation of the system requirement currently driving development.

**Step 2 – Generate an MCH Triad and Unit Tests**

No production code or unit tests yet exist. Development continues with using the file generation script. A Model, Conductor, and Hardware are each generated with a source, header, and unit test skeleton file.

**Step 3 – Add calls to the Executor for the Conductor's Init and Run Functions**

A system of even modest complexity will contain several MCH triads. For multi-triad systems, a single Executor contains the main execution loop and calls each Conductor to run its logic. Because the order in which calls made on the Conductors is usually significant, the Executor must be edited by hand. Before calls to the new Conductor are added to the Executor, however, the Executor's unit tests are updated to ensure that all the Conductors are being called properly. Within the Executor tests, CMock generated mocks are used to stand in for the actual Conductors.

**Step 4 – Create Conductor Unit Tests & Production Code**

Starting from the system requirement currently being implemented, we next determine the individual functions necessary to satisfy that requirement. These are expressed as Conductor unit tests. Once the tests are created, production code to satisfy these tests is added to the Conductor.

The logic of the Conductor is tested against a mock Hardware and Model. Each function call made against the Hardware or Model must be added in the Hardware or Model's header file. CMock will automatically generate the corresponding mocks from the Hardware and Model header files prior to running the unit tests.

**Step 5 – Create Model & Hardware Unit Tests & Production Code**

The Model and Hardware triad members are next implemented. Tests are implemented first; in the process, the interfaces for calls on the physical hardware and any interaction with other triad Models are revealed (interaction among multiple triads is discussed in depth in the MCH section).

Both the Hardware and Model will likely have initialization functions called by the Conductor. Functions in the Hardware and Model (already specified in the Conductor tests) may best be implemented by delegating responsibility to other helpers. In such cases, the module generation script is used to create these helper modules. Tests in the Model and Hardware will make calls against mocks of these helper functions.

The Hardware triad component, of course, makes calls to the physical hardware itself. Unit tests here generally involve modifying registers, calling the functions of the triad's Hardware member, and then making assertions against the resulting register values. This can be complicated by read-only and write-only registers. Simulators often allow violations of these restrictions which may benefit or hinder test development.

## Step 6 – Complete All Test and Production Programming

The tests and functional code of the Model, Conductor, Hardware, and helpers may require refactoring and subsequent re-running until the code is clean and all tests pass. When the unit tests are run, a Rake task will: call CMock to automatically generate Hardware and Model (and helper) mocks from header files, call Argent to insert Unity and CMock functions in appropriate files, use the dependency generation tool to create a list of files to be linked (including the unit tests, mocks, and Unity framework itself), run the compiler and linker, and then run the resulting test executable appropriately. Executing the test executable may require starting a simulator or flashing the target hardware and collecting test results from a serial port or other communication channel.

Once all the unit tests pass, the system can be compiled in non-test mode, loaded onto the hardware, and run against the system tests and hardware test fixture. Changes and fixes may be necessary to cause system tests to pass. Once tests pass, the new and modified files are checked into the source code repository; our automated build system will build and test the entire project and report results. Work on the next feature then begins anew.

## Model Conductor Hardware Design Pattern & Conductor First

Design patterns are documented approaches to solving commonly occurring problems in software development. A multitude of patterns exist that address common situations in elegant and language-independent ways [12]. The Model Conductor Hardware pattern decouples functional logic from hardware for testing. With this decoupling, automated regression suites of integration and unit tests can be run on-chip, cross-compiled on a PC, or executed in a platform simulator. Working in this manner can even generate significant progress without target hardware.

## Model Conductor Hardware Similarities to Model View Presenter

The basic MCH concepts and naming are modeled on the MVP design pattern. The Model of both patterns serves the same basic function – to model the system (e.g. mathematical equations, business logic, data access, etc.). The Hardware component of MCH is analogous to the View of MVP – both are lightweight abstractions that allow decoupling of system components and programming logic. The Conductor component of MCH serves a similar purpose to the Presenter of MVP – both define the order and composition of interactions within the triad upon an event occurring in either of the other triad components. Development in MCH starts with tests in the Conductor similar in fashion to the Present First [6,16] method of MVP development.

## Model

The Model in MCH models the system (e.g. mathematical equations, control logic, etc.), holds state, and provides a programmatic interface to the portion of the system that exists outside a single MCH triad. The Model is only connected to the Conductor and has no direct reference to the Hardware.

## Conductor

The Conductor in MCH conducts the flow of data between the Model and Hardware and is stateless. The Conductor acts when triggered by the Hardware or Model. It captures a system function in Model and Hardware interactions: setting state within the Model, querying the state contained by the Model, querying the state contained in the Hardware, moving data from the Model to the Hardware, moving data from the Hardware to the Model, and initiating the hardware functions encapsulated by the Hardware.

The Conductor was so named because of its role as a system director and because of its proximity to actual electrical components. Because of the correspondence to MVP, one could naturally assume that “Model Conductor Hardware” should be called “Model Hardware Conductor.” The apparent incongruity is due to the importance we place on defining the triad interactions at the Conductor first. As the Conductor is central to the behavior of an MCH triad, we elected to name the design pattern to reflect this.

## Hardware

The Hardware in MCH represents a thin layer around the physical hardware itself (e.g. ports, registers, etc.) and all its accompanying functions and state-based facilities. The Hardware notifies the Conductor of events by providing functions that check state in the hardware. State can be hardware registers or flags set by Interrupt Service Routines (ISRs) and other Hardware routines. The Hardware is only connected to the Conductor and has no direct reference to the Model.

## Unit Testing with Model Conductor Hardware

With mocks constructed for each member of the MCH triad, clear testing possibilities become apparent (please see preceding sections for a discussion of mocks and interaction-based testing). The states and behavior within the Model are tested independently of hardware events and control logic. Each system requirement’s logic in the Conductor is tested with simulated events and states from the Hardware and Model. With mocks, even hardware register configuration code and ISRs can be tested. An MCH example follows in a later section of this paper.

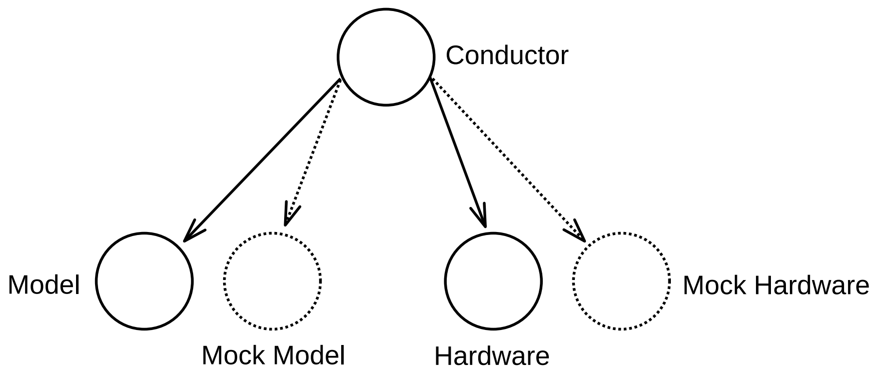


Fig. 1. Relationships of a Conductor to its complementary MCH triad members and mocks. The depicted mocks stand in for the concrete members of the triad (linked in at build time) and allow testing of the logic within the Conductor. Mocks are automatically generated capture function parameters and simulate return values used in test assertions.

## Model Conductor Hardware & Multi-Triad Systems

Multiple triads are necessary to compose an entire system and simplify testing. Multiple triads exist independently of one another. A central Executor contains the system's main execution loop; it initializes and continually runs each triad by making calls on the Conductors. If there is communication necessary among triads in a multi-triad system, it will occur between Models. In a real-time system, individual triads can map well to tasks or threads.

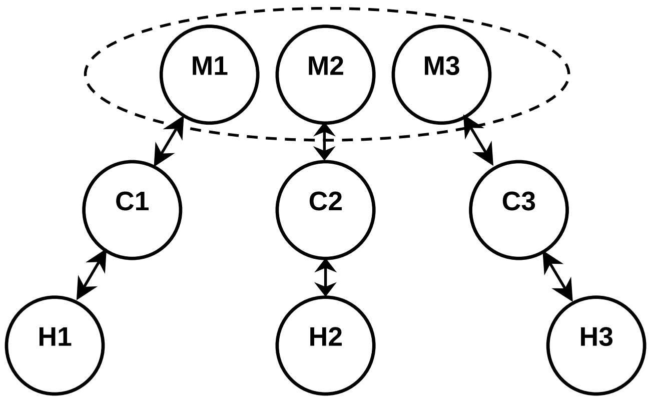


Fig. 2. A system composed of multiple MCH triads. Inter-triad communication occurs among Models through direct calls or shared helpers. An external Executor runs each triad by making calls to service each Conductor.

## Overhead & Development Cost

MCH adds little to no overhead (i.e. memory use or processor time) to a production embedded system. Of course, mocks and unit tests are not compiled into the final release system. Further, MCH is essentially naming, organization, and calling conventions; any overhead incurred by these conventions is easily optimized away by the compiler.

TDD shifts the time normally spent repaying technical debt [3] at project conclusion (and beyond) to the development phase with test-first practices. With this shift comes predictable development allowing decision makers to manage project risks with progress data. With TDD and CI clear savings are realized over the system's lifetime in reduced bugs, reduced likelihood of recall, and ease of feature additions and modifications. New developers can work on the system and make changes with confidence because of test suites. Those tests, in fact, operate as "living" documentation of the system. Looking at the entire life cycle of a system, TDD and CI are considerably more cost effective than traditional or ad hoc methods of development.

## Conclusion

Picture a desperate embedded systems engineer surrounded by wires, boards, and scopes, trying to track down a mysterious bug in a product that's three weeks past its deadline. Now imagine a calm, productive developer over the course of many months regularly defining system behavior through tests and checking in code for the continuous integration server to build and test. Further, imagine a project manager confidently estimating and adjusting project completion goals and costs from measurable progress metrics. Regression testing, decoupled design, and disciplined development can all but eliminate heroic debugging efforts and costly defects released into production.

## Acknowledgments

Authors wish to thank Matt Werner of Savant Automation for the opportunity to implement and originally prove out in a production system the ideas that inspired this paper. Authors also wish to thank their colleagues at Atomic Object for their many contributions to this work including but not limited to Presenter First, Systir, the dependency generation tool, and, of course, much proofreading and editing assistance.

## References

[1] Kent Beck. Extreme Programming Explained. Reading, MA: Addison Wesley, 2000.

[2] Michael Karlesky, William Bereza, and Carl Erickson. "Effective Test Driven Development for Embedded Software." IEEE EIT2006. East Lansing, Michigan. May 2006.

[3] Technical Debt, http://www.c2.com/cgi/wiki?TechnicalDebt

[4] Nancy Van Schooenderwoert. "Embedded Agile: A Case Study in Numbers", http://www.ddj.com/dept/architect/193501924, November 6, 2006.

[5] Steve Freeman, Nat Pryce, Tim Mackinnon, Joe Walnes. "Mock Roles, not Objects." OOPSLA 2004. Vancouver, British Columbia.

[6] M. Alles, D. Crosby, C. Erickson, B. Harleton, M. Marsiglia, G. Pattison, C. Stienstra. "Presenter First: Organizing Complex GUI Applications for Test-Driven Development." Agile 2006. Minneapolis, MN. July 2006.

[7] System Testing in Ruby, http://atomicobject.com/pages/System+Testing+in+Ruby

[8] Ruby Programming Language, http://www.ruby-lang.org/en/

[9] Domain Specific Language, http://en.wikipedia.org/wiki/Domain_Specific_Language

[10] Measurement Computing Corp. miniLAB 1008, http://www.measurementcomputing.com/cbicatalog/cbiproduct_new.asp?dept_id=412&pf_id=1522

[11] Argent, http://rubyforge.org/projects/argent/

[12] Kent Beck. "Simple Smalltalk Testing: With Patterns," http://www.xprogramming.com/testfram.htm.

[13] http://atomicobject.com/pages/Embedded+Software

[14] Martin Fowler. "Dependency Injection," http://www.martinfowler.com/articles/injection.html.

[15] http://rake.rubyforge.org/

[16] David Crosby and Carl Erickson. "Big, Complex, and Tested? Just Say 'When': Software Development Using Presenter First", Better Software Magazine. February 2007.

## Appendix

### Sample Project

To demonstrate the ideas in this article, we created a full sample project for an Atmel ARM7-based development system (AT91SAM7X). This example system samples a thermistor, calculates a weighted average temperature, outputs the value through an RS232 serial port, and handles error conditions. The project includes three MCH triads: analog-to-digital conversion (ADC), timer management, and serial data output. The full sample project including support tools is available at http://atomicobject.com/pages/Embedded+Software. In the near future, much improved versions of our custom tools with documentation will be made freely available.

For discussion here, we will walk through development of the ADC triad from beginning to end. Along the way we will also demonstrate the use of the tools we developed to complement the principles espoused earlier.

### Summary of Steps

1. Create system tests for the temperature conversion feature to be developed.
2. Generate a skeleton Model, Conductor, Hardware, and accompanying test files.
3. Add tests and production code for initialization to the Executor, ADC triad, and helper functions.
4. Add tests and production code to the Executor, ADC triad, and helper functions for runtime temperature capture and conversion. Start with the Conductor first.
5. Verify that all unit tests and system tests pass.

### Create Temperature Conversion System Tests

Using Systir, the miniLAB driver, and a serial port driver, our system test provides several input voltages to the development system (simulating a thermistor in a voltage divider circuit) and expects temperatures in degrees Celsius to be read as ASCII strings from the development PC’s serial port. The system test is running externally to the development board itself.

```
proves "voltage read at temperature sensor input is translated and reported in degrees C"

# Verifies the 'stable' range of the sensor (10C-45C). Instability is due to the flattening
# of the temperature sensor voltage to temperature response curve at the extremes.

set_temperature_in_degrees_C(10.0)
verify_reported_temperature_in_degrees_C_is_about(10.0)

set_temperature_in_degrees_C(25.0)
verify_reported_temperature_in_degrees_C_is_about(25.0)

set_temperature_in_degrees_C(35.0)
verify_reported_temperature_in_degrees_C_is_about(35.0)

set_temperature_in_degrees_C(40.0)
verify_reported_temperature_in_degrees_C_is_about(40.0)
```

Fig. 3. Systir system test for verification of temperature conversion. Driver code handling timing, test voltage calculation, and communications is not shown.

## Generate Skeleton MCH Triad and Unit Test Files

We create skeleton versions of the source file, header file, and test file for each member of an ADC MCH triad. The use of the generation script and an Empty ADC Conductor are shown in the following figures. For each component in the system (whether a member of a triad or a helper), this same process is repeated to create skeleton files.

```
> ruby auto/generate_src_and_test.rb AdcConductor
> ruby auto/generate_src_and_test.rb AdcModel
> ruby auto/generate_src_and_test.rb AdcHardware
```

Fig. 4. Generating a skeleton MCH triad and tests with the module generation Ruby script.

```
#ifndef _ADCCONDUCTOR_H
#define _ADCCONDUCTOR_H

#include "Types.h"

#endif // _ADCCONDUCTOR_H
```

Fig. 5. AdcConductor skeleton header.

```
#include "AdcConductor.h"
```

Fig. 6. AdcConductor skeleton source file.

```
#include "unity_verbose.h"
#include "CMock.h"
#include "AdcConductor.h"

static void setUp(void)
{
}
static void tearDown(void)
{
}

static void testNeedToImplement(void)
{
  TEST_FAIL("Implement me!");
}

//[[$argent require 'generate_unity.rb'; inject_mocks("AdcConductor");$]]
//[[$end$]]

//[[$argent require 'generate_unity.rb'; generate_unity();$]]
//[[$end$]]
```

Fig. 7. TestAdcConductor skeleton unit test file. Note inclusion of Argent blocks. Before compilation, a Rake task will execute Argent, and these blocks will be populated with C code to use the Unity test framework and CMock generated mock functions (mocks are generated from header files by another build task prior to Argent execution).

## Add Tests and Production Code for Initialization

The Executor initializes the system by calling initialization functions in each triad's Conductor. The Conductors delegate necessary initialization calls. The ADC Hardware will use a helper to configure the physical analog-to-digital hardware. For brevity, the Executor's tests and implementation are not shown here.

```
static void testInitShouldCallHardwareInit(void)
{
  AdcHardware_Init_Expect();
  AdcConductor_Init();
}
```

Fig. 8. Initialization integration test within the Conductor's test file. The "_Expect" function is automatically generated by CMock from the interface specified in the Hardware header file.

```
      void AdcConductor_Init(void)
      {
        AdcHardware_Init();
```

Fig. 9. Conductor's initialization function that satisfies its unit test.

```
static void testInitShouldDelegateToConfiguratorAndTemperatureSensor(void)
{
  Adc_Reset_Expect();
  Adc_ConfigureMode_Expect();
  Adc_EnableTemperatureChannel_Expect();
  Adc_StartTemperatureSensorConversion_Expect();

  AdcHardware_Init();
}
```

Fig. 10. Initialization integration test within the TestAdcHardware's test file. Calls are made to an AdcHardwareConfigurator helper and an AdcTemperatureSensor helper. The "_Expect" and "_Return" functions are automatically generated by CMock from the interfaces specified in header files.

```
void AdcHardware_Init(void)
{
  Adc_Reset();
  Adc_ConfigureMode();
  Adc_EnableTemperatureChannel();
  Adc_StartTemperatureSensorConversion();
}
```

Fig. 11. AdcHardware's initialization function that satisfies its integration test.

```
static void testResetShouldResetTheAdcConverterPeripheral(void)
{
  AT91C_BASE_ADC->ADC_CR = 0;
  Adc_Reset();
  TEST_ASSERT_EQUAL(AT91C_ADC_SWRST, AT91C_BASE_ADC->ADC_CR);
}

static void testConfigureModeShouldSetAdcModeRegisterAppropriately(void)
{
  uint32 prescaler = (MASTER_CLOCK / (2 * 2000000)) - 1; // 5MHz ADC clock
  AT91C_BASE_ADC->ADC_MR = 0;
  Adc_ConfigureMode();
  TEST_ASSERT_EQUAL(prescaler, (AT91C_BASE_ADC->ADC_MR & AT91C_ADC_PRESCAL) >> 8);
}

static void testEnableTemperatureChannelShouldEnableTheAppropriateAdcInput(void)
{
  AT91C_BASE_ADC->ADC_CHER = 0;
  Adc_EnableTemperatureChannel();
  TEST_ASSERT_EQUAL(0x1 << 4, AT91C_BASE_ADC->ADC_CHER);
}
```

Fig. 12. AdcHardwareConfigurator's initialization unit tests.

```
void Adc_Reset(void)
{
  AT91C_BASE_ADC->ADC_CR = AT91C_ADC_SWRST;
}

void Adc_ConfigureMode(void)
{
  AT91C_BASE_ADC->ADC_MR = (((uint32)11) << 8) | (((uint32)4) << 16);
}

void Adc_EnableTemperatureChannel(void)
{
  AT91C_BASE_ADC->ADC_CHER = 0x10;
}
```

Fig. 13. AdcHardwareConfigurator's initialization functions that satisfy its unit tests.

```
static void testShouldStartTemperatureSensorConversionWhenTriggered(void)
{
  AT91C_BASE_ADC->ADC_CR = 0;
  Adc_StartTemperatureSensorConversion();
  TEST_ASSERT_EQUAL(AT91C_ADC_START, AT91C_BASE_ADC->ADC_CR);
}
```

Fig. 14. AdcTemperatureSensors's start conversion unit test.

```
void Adc_StartTemperatureSensorConversion(void)
{
  AT91C_BASE_ADC->ADC_CR = AT91C_ADC_START;
}
```

Fig. 15. AdcTemperatureSensors' start conversion function that satisfies its unit test.

## Add Tests and Production Code for Temperature Conversion

The models of all the triads are tied together for inter-triad communication. The timing Model repeatedly updates a task scheduler helper with time increments. The ADC Model returns to the Conductor a boolean value processed by the task scheduler helper to control how often an analog-to-digital conversion occurs.

The ADC Conductor is repeatedly serviced by the Executor once initialization is complete (the other triad Conductors are serviced in an identical fashion). Each time the ADC Conductor is serviced, it checks the state of the ADC Model to determine whether an analog-to-digital conversion should occur. When a conversion is ready to occur, the ADC Conductor then instructs the ADC Hardware to initiate an analog-to-digital conversion. Upon completion of a conversion, the Conductor provides the raw value in millivolts to the ADC Model for temperature conversion. The timing and serial communication triads will cooperate to average and periodically output a properly formatted temperature string. Here, we walk through the tests and implementation of the ADC Conductor's interaction with the ADC Hardware and ADC Model down through to the hardware read of the analog-to-digital channel.

```
static void testRunShouldNotDoAnythingIfItIsNotTime(void)
{
  AdcModel_DoGetSample_Return(FALSE);
  AdcConductor_Run();
}

static void testRunShouldNotPassAdcResultToModelIfSampleIsNotComplete(void)
{
  AdcModel_DoGetSample_Return(TRUE);
  AdcHardware_GetSampleComplete_Return(FALSE);
  AdcConductor_Run();
}

static void
testRunShouldGetLatestSampleFromAdcAndPassItToModelAndStartNewConversionWhenItIsTime(void)
{
  AdcModel_DoGetSample_Return(TRUE);
  AdcHardware_GetSampleComplete_Return(TRUE);
  AdcHardware_GetSample_Return(293U);
  AdcModel_ProcessInput_Expect(293U);
  AdcHardware_StartConversion_Expect();
  AdcConductor_Run();
}
```

Fig. 16. TestAdcConductor integration tests for interactions with AdcHardware and AdcMcodel. The "_Expect" and "_Return" functions are automatically generated by CMock from the interfaces specified in header files.

```
void AdcConductor_Run(void)
{
  if (AdcModel_DoGetSample() && AdcHardware_GetSampleComplete())
  {
    AdcModel_ProcessInput(AdcHardware_GetSample());
    AdcHardware_StartConversion();
  }
}
```

Fig. 17. AdcConductor's run function (called by the Executor) that satisfies the Conductor unit tests.

```
static void
testGetSampleCompleteShouldReturn_FALSE_WhenTemperatureSensorSampleReadyReturns_FALSE(void)
{
  Adc_TemperatureSensorSampleReady_Return(FALSE);
  TEST_ASSERT(!AdcHardware_GetSampleComplete());
}

static void
testGetSampleCompleteShouldReturn_TRUE_WhenTemperatureSensorSampleReadyReturns_TRUE(void)
{
  Adc_TemperatureSensorSampleReady_Return(TRUE);
  TEST_ASSERT(AdcHardware_GetSampleComplete());
}

static void testGetSampleShouldDelegateToAdcTemperatureSensor(void)
{
  uint16 sample;
  Adc_ReadTemperatureSensor_Return(847);

  sample = AdcHardware_GetSample();
  TEST_ASSERT_EQUAL(847, sample);
}
```

Fig. 18. Integration tests for AdcHardware. Note that the “_Expect” and “_Return” functions are automatically generated by CMock from the interfaces specified in header files.

```
void AdcHardware_StartConversion(void)
{
  Adc_StartTemperatureSensorConversion();
}

bool AdcHardware_GetSampleComplete(void)
{
  return Adc_TemperatureSensorSampleReady();
}

uint16 AdcHardware_GetSample(void)
{
  return Adc_ReadTemperatureSensor();
}
```

Fig. 19. AdcConductor’s functions satisfying the Conductor integration tests. Note that AdcHardware calls helper functions in AdcTemperatureSensor

```
static void testShouldStartTemperatureSensorConversionWhenTriggered(void)
{
  AT91C_BASE_ADC->ADC_CR = 0;
  Adc_StartTemperatureSensorConversion();
  TEST_ASSERT_EQUAL(AT91C_ADC_START, AT91C_BASE_ADC->ADC_CR);
}

static void testTemperatureSensorSampleReadyShouldReturnChannelConversionCompletionStatus(void)
{
  AT91C_BASE_ADC->ADC_SR = 0;
  TEST_ASSERT_EQUAL(FALSE, Adc_TemperatureSensorSampleReady());
  AT91C_BASE_ADC->ADC_SR = ~AT91C_ADC_EOC4;
  TEST_ASSERT_EQUAL(FALSE, Adc_TemperatureSensorSampleReady());
  AT91C_BASE_ADC->ADC_SR = AT91C_ADC_EOC4;
  TEST_ASSERT_EQUAL(TRUE, Adc_TemperatureSensorSampleReady());
  AT91C_BASE_ADC->ADC_SR = 0xffffffff;
  TEST_ASSERT_EQUAL(TRUE, Adc_TemperatureSensorSampleReady());
}

static void testReadTemperatureSensorShouldFetchAndTranslateLatestReadingToMillivolts(void)
{
  uint16 result;

  // ADC bit weight at 10-bit resolution with 3.0V reference = 2.9296875 mV/LSB
  AT91C_BASE_ADC->ADC_CDR4 = 138;
  result = Adc_ReadTemperatureSensor();
  TEST_ASSERT_EQUAL(404, result);

  AT91C_BASE_ADC->ADC_CDR4 = 854;
  result = Adc_ReadTemperatureSensor();
  TEST_ASSERT_EQUAL(2502, result);
}
```

Fig. 20. Unit tests for AdcTemperatureSensor helper.

```
void Adc_StartTemperatureSensorConversion(void)
{
  AT91C_BASE_ADC->ADC_CR = AT91C_ADC_START;
}

bool Adc_TemperatureSensorSampleReady(void)
{
  return ((AT91C_BASE_ADC->ADC_SR & AT91C_ADC_EOC4) == AT91C_ADC_EOC4);
}

uint16 Adc_ReadTemperatureSensor(void)
{
  uint32 picovolts = ConvertAdcCountsToPicovolts(AT91C_BASE_ADC->ADC_CDR4);
  return ConvertPicovoltsToMillivolts(picovolts);
}


static inline uint32 ConvertAdcCountsToPicovolts(uint32 counts)
{
  // ADC bit weight at 10-bit resolution with 3.0V reference = 2.9296875 mV/LSB
  uint32 picovoltsPerAdcCount = 2929688;

  return counts * picovoltsPerAdcCount; // Shift decimal to preserve accuracy in fixed-point
}

static inline uint16 ConvertPicovoltsToMillivolts(uint32 picovolts)
{
  const uint32 halfMillivoltInPicovolts = 500000;
  const uint32 picovoltsPerMillivolt = 1000000;

  // Add 0.5 mV to result so that truncation yields properly rounded result
  picovolts += halfMillivoltInPicovolts;

  return (uint16)(picovolts / picovoltsPerMillivolt); // Divide to convert to millivolts
}
```

Fig. 21. AdcTemperatureSensor helper functions satisfying the AdcTemperatureSensor unit tests.

**Verify that all unit tests and system tests pass.**

The process of adding tests and production code to satisfy a system requirement, of course, requires an iterative approach. Design decisions and writing test and production code require multiple passes.

The code samples presented in the preceding section represent finished product. As the sample project was developed, tests and code were refactored and run numerous times. With the automation provided by our unit and system test frameworks, CMock, and the build tools, this process was nearly painless.

```
> rake clean
> rake test:units
> rake test:system
```

Fig. 22. Running rake tasks to build and test the system. The definitions of the rake tasks and the calls to CMock & Argent, dependency generation, compiling, and linking are defined within the Rakefile and not shown.

# Personal Quality Management with the Personal Software Process

Alan S. Koch
ASK Process, http://www.ASKProcess.com

## Abstract

Software development organizations have a variety of mechanisms at their disposal to help in managing and improving the quality of the products they produce. Quality Assurance organizations, problem reporting systems, software process improvement and peer reviews (to name just a few) are important tools for product quality enhancement. But an often-overlooked piece of the quality puzzle may well provide the most effective means to improve product quality: the individual software engineer.

After a short introduction to what the Personal Software Process (PSP) $^{SM}$ ("Personal Software Process" and "PSP" are Service Marks of Carnegie Mellon University) is, we will highlight the ways in which individual engineers (and their organizations) can benefit from adding the PSP's Personal Quality Management techniques to their professional repertoires. We will take a brief look at the benefits that have been achieved by those who have already learned to apply these principles in their work. Then we will examine in more detail the specific activities PSP-trained engineers engage in to manage the quality of the software they produce. We will look at everything from simple defect logging, to personal and peer reviews, to developing a personal quality plan and using it to guide your work.

## What is the PSP?

The concepts and activities discussed in this article are the quality management aspects of the Personal Software Process (PSP) developed by Watts S. Humphrey of the Software Engineering Institute (SEI), and described in his book, *PSP(sm): A Self-Improvement Process for Software Engineers* [Humphrey].

The PSP is more than just training; it is a boot camp consisting of about 40 hours of classroom instruction, 10 programming assignments, and 3 data-analysis exercises, requiring a total of about 150 hours for the average programmer to complete.

The result of the PSP boot camp is that the programmers don't just learn about good processes, they actually improve their own processes, measure the effects of those process changes, quantify the benefits they have experienced, and set goals for further improvements. The PSP achieves these results by leading students through three steps.

In PSP0, they lay a simple foundation for the learning to come:

- Following simple process scripts,
- Collecting three basic measures of their work (time spent, size of products produced, and defects corrected), and
- Performing a simple post-project analysis.

In PSP1, they begin to build the capability to plan and manage their own work, setting the stage for Personal Quality Management:

- Following a defined project planning process,
- Using their own prior data to make increasingly more accurate estimates for each programming assignment, and
- Planning their work at a level of detail that allows them to track and manage their progress.

In PSP2, they focus on achieving significant quality improvements by learning how to engage in Personal Quality Management:

- Using their prior data to plan for incremental improvements in the quality of their programs,
- Removing defects early using personal review techniques guided by their own prior defect performance, and
- Identifying and capitalizing on defect prevention opportunities in their program design and implementation methods.

Those who complete the PSP boot camp emerge with the knowledge and skills to make accurate plans, work to those plans, and produce superior quality products.

Advertisement – AjaxWorld Conference - Click on ad to reach advertiser web site

Register as a Methods & Tools reader with coupon code "mtgold" and get a $50 discount on Golden Pass: https://www3.sys-con.com/sep07/registernew.cfm

Web 2.0 and Rich Internet Apps
AJAXWORLD™
CONFERENCE & EXPO
September 24-26, 2007
Santa Clara Convention Center
Santa Clara, CA
Early-Bird
Save $200!
To Register:
Call 201-802-3020 or visit...
www.AJAXWorld.com
SYS-CON EVENTS
Join Over 3,600 Early AJAX Adopters Who Attended the#1 Web 2.0 Event in the World!
Providing developers and IT managers alike with comprehensive information and insight into the biggest paradigm shift in website design, development, and deployment since the invention of the World Wide Web itself a decade ago.

## The Motivation for Personal Quality Management

Most programmers have never learned to apply personal quality management techniques like those described in this article in their work. This is because our educational system has generally ignored the topic, and in the software business, “quality assurance” often refers to little more than testing. This is unfortunate, because vast opportunities for improving the quality of the software we produce are being missed as we continue with business as usual.

This is surprising, given that these techniques are anything but new. Other engineering disciplines have embraced them for decades and manufacturers and engineering companies have been applying them for nearly fifty years. Methods like these have been researched and described by such luminaries as Walter A. Shewhart and W. Edwards Deming as well as their more modern counterparts.

Watts Humphrey, in the PSP has given programmers the opportunity to understand these methods and apply them to software development. Those who have done so have found the effects on their work to be dramatic, with the improvement in the quality of their software the most obvious and rewarding aspect. These individuals have begun to take control of the task of writing programs, and are taking the first steps toward changing “software engineering” into a true engineering discipline.

## Defect Content as a Dimension of Quality

When we think of software quality, we tend to think about defects. This is a natural tendency, since defective software isn’t good for much. But defects (or more correctly, lack of defects) is not the only, or even the most important dimension of quality. Other dimensions of quality include Usability, Maintainability, Install-ability, Security, and Reliability. In any product, each of these dimensions is important to one degree or another, and every development project must be careful to pay appropriate attention to them.But from the point of view of Personal Quality Management, defect management is first priority. This is true for two reasons:

- The other dimensions of quality are issues of either product requirements or organizational standards. While the individual software engineer can affect these things, they are not under his or her direct control.
- Defect injection is a purely personal phenomenon. Each person’s defect numbers, types and patterns are unique, and require personalized actions to effectively detect and eliminate them.

You may note throughout this article that I will never refer to “bugs”. I believe that this term trivializes a very important issue, making it sound minor, or even cute. In fact, defective software wastes millions of dollars for companies and individuals throughout the world. And as we become more and more dependent on software for critical functions, it is becoming important to organizations’ and individuals’ health and life. In order to focus on defects as a quality management priority, we must understand their nature. This nature can be summed up this way:

- Software that contains one or more defects is “defective”.
- Defects are the result of human errors.
- All humans make errors.
- All software engineers are human.

Therefore, defect management must be the first quality management focus for all Software Engineers.

## Career Enhancement by Reducing Both Costs and Defects

As professionals, we Software Engineers must assume responsibility for the quality of the products we produce. Our employers pay us for producing not just software, but *good* software, high-quality software.
The quality assurance organization cannot be thought of as the people who are responsible for quality. Indeed, of all members of the organization, they are in the position that is least able to affect the quality of the product. Testing and other QA activities can detect defects and make sure that they are removed, but quality *cannot* be tested into the product at the end. Quality must be built in or it will be absent. Therefore, the software engineers are the ones who have the most direct impact on the quality of the product.

It is easy to lose sight of the tremendous costs that defects cause for our employers. Integration and system test can account for 50% of development time (and most of that time is spent in remediation and re-testing to eliminate the scores of defects that are commonly found). Then post-release defect mitigation and removal can cost more than the original development effort (including testing)! Supporting customers, investigating problems and releasing patches consume huge amounts of time and effort. And the cost of defects in loss of customer good will and market image goes beyond even that.

Defect removal costs escalate with each passing lifecycle phase. Different people have estimated different costs at the various phases, but they tend to agree that the costs rise (possibly even exponentially) with each passing phase. (e.g. A defect that might cost one dollar to fix in the requirements phase may cost $10 during design, $100 during coding, $1,000 in test and $10,000 in the field.) Regardless of the specific numbers one might cite, the cost escalation is obvious from the fact that we talk about removing “defects per hour” before Integration, and spending “hours per defect” during and after Integration.

Your value to your employer will be enhanced by your ability to produce salable (or usable) software at a minimum cost. Although the early defect detection and prevention methods we will discuss all have costs associated with them, they are demonstrably less expensive than removing the same defects later in the software lifecycle. So you can reduce costs through defect prevention and early defect removal, while at the same time producing more salable or usable software with fewer delivered defects. You enhance your career and value to your employer by exercising professional responsibility and managing your own quality.

## Quality of Work-Life Issues

Practitioners of Personal Quality Management have also found that it has a positive impact on the quality of their work-life.

Most of us went into programming because we enjoy building things and seeing the results of our labor. We like the challenge of figuring out how to attack a problem, of designing the structures and algorithms that will get the job done, and even of writing the code to make it all happen. And we like to see the fruit of our labor integrated into a system that meets people’s needs or provides an important service.

Advertisement – Open Source Conferences - Click on ad to reach advertiser web site

ubuntu LIVE

July 22 – 24, 2007
Portland, Oregon

Register Now and Save an Additional 10%
Use Discount Code ubu07mtht

Listen. Discuss. Learn. Ubuntu in action.

- An interactive, in-depth, and comprehensive educational experience
- The opportunity to connect face to face with other Ubuntu users
- A well-edited, coherent conference program including tutorials, sessions and keynotes
- Information and tools to help managers decide to switch to Ubuntu and the developers implement that transition
- An open-minded meeting ground for hackers, developers and ITmanagers
- An exhibit hall filled with hardware and software businesses showcasing open source products and services

www.ubuntulive.com

Co-presented by CANONICAL O'REILLY

O'REILLY

OSCON
Open Source Convention

July 23 – 27, 2007
Portland, Oregon

Now in its ninth year, the O'Reilly Open Source Convention is the bazaar of open source technologies, welcoming new voices and projects alongside the platforms, languages, and apps that started the open source movement. OSCON brings together over 2500 experts, visionaries, open source professionals, IT managers, sys admins, hackers, and entrepreneurs to explore the depth and breadth of open source.

**Register Now and Save an Additional 10%**
Use Discount Code **os07mtht**

www.conferences.oreilly.com/oscon

But between the coding and use, there is a long and (for most of us) painful time of fighting our way past defect after defect. It starts with the compiler leading us around by our noses, complaining about every little typo and syntax error, and often totally misinterpreting what the code was supposed to say. Then after achieving a clean compile, we try over and over again to get the program to do the right thing. It loops, it hangs, it crashes; and now the debugger is the thing that is dragging us around by the nose. For each problem we fix, there seems always to be another one lurking in the shadows. And even more frustrating is that too often our fix was the cause of the ensuing problem!

But releasing the software doesn't end this frustration. Right when we're in the middle of designing the next interesting program, we get interrupted to fix a latent problem. There is a fuming customer and an unhappy boss, so we must drop everything to make an emergency fix. And to add insult to injury, the interruption of our design work is likely planting the seeds for future problems and their interruptions and frustrations.

Personal Quality Management allows us to gain control over defects. Instead of being ruled by the defects in our software, we can gain the upper hand. We can understand them, find them, remove them earlier and even avoid them altogether. That means that we will spend a much greater proportion of our time on the tasks we enjoy, and less on the problems we missed. That is a significant improvement in our work-life!

**Results in the Classroom & Field**

Personal Quality Management is a significant component of the PSP. A growing body of evidence shows that the PSP significantly improves the engineers' ability to remove defects early without compromising their productivity.

The SEI published a study of the data from 298 PSP students [Hays] that showed the differences in programmers' performance as they progressed through the class. Among other things, they removed more than five times as many defects before testing by the end of the course than at the beginning,

Companies that have adopted PSP have confirmed that these dramatic improvements can be achieved in business. Teradyne, Advanced Information Services, Hill Airforce Base and Harris Communications have publicly reported dramatic results. (See my web site at www.ASKProcess.com for a relatively complete compilation of published results.)

Boeing reported the most striking among the results at the 2000 SEPG Conference, when they compared their first project by PSP-trained engineers with the prior three projects from the same group. In spite of being significantly more than twice as big as the prior releases, system test was complete in less than a week, as compared with a month or more for the others. Boeing attributes this 94% reduction in test time to the 75% reduction in the number of defects that were found in system test. PSP's early defect removal paid significant dividends.

Although there is significant variation from one organization or individual to the next, the available data suggest that reasonable expectations for software engineers would be to remove 75% of all defects before integration without extending the project schedule, followed by significant time savings during integration and system testing. And all of this is achieved while enhancing the professional satisfaction (and hopefully, recognition) of the software engineers.

## Principles and Practice

Personal Quality Management is not a single thing that you can do. It consists of a number of interrelated activities that build upon each other, resulting in the dramatic improvements we just reviewed. The remainder of this article will examine these specific activities.

## Personal Reviews

There is a popular belief that a software engineer can not review his or her own code. The PSP proves that this belief is false if the reviews are done correctly.

An effective code review is not just a matter of reading the code and hoping to find the problems that are hidden in it. Like many other activities, personal reviews are most effective when they are based on a well-defined and structured process. An engineer's personal review process must be structured to direct his or her attention toward the types of defects that he or she has historically injected. The PSP teaches engineers to scan their code multiple times, each time looking for a specific type of defect. This method of focused scanning is remarkably effective at finding defects, and is also quite efficient.

I had been doing personal reviews by simply reading my code for several years, but was finding fewer than 30% of the defects I had injected. After learning the PSP methods, my results were similar to those of many other people, with my personal reviews finding more than 75% of my defects! PSP-trained engineers will agree that personal reviews are definitely worthwhile!

## Defect Logging

As was mentioned above, personal reviews will only be effective if they are personalized so they direct your attention to the defects you normally inject. This presupposes that you know your own defect history. As with anything else, informal memory is not likely to serve you well in understanding your defect history.

There is only one way to construct an actuate defect profile for your work: Log every defect that escapes the phase in which it was injected. The PSP suggests logging the following information for each defect:

- <u>Type</u> – To manage the large volume of defect data that you will quickly amass, you will need a way to categorize and summarize it. The next section will describe this step.
- <u>Phases injected and removed</u> – By identifying when each defect was removed and estimating when it was injected, you can understand which defect detection activities are working well and which need to be improved. (The phases identified by the PSP are Design, Design Review, Code, Code Review, Compile, Unit Test and After Development.)
- <u>Fix Time</u> – The number of minutes required to remove the defect (including research, debugging, reworking the design, reworking the code, compiling and re-testing) allows you to understand the magnitude of the problem posed by each defect.
- <u>Fix Defect</u> – If the defect was injected by fixing a defect, record a reference to the other defect so you can understand the types of validation that defect fixes require.
- <u>Description</u> – A concise description of what was wrong (not the symptom, but the cause) will allow you to recognize similar defects in the future.

Making a defect log entry takes only a few seconds, so it has little impact on your productivity. However, if you fail to log the information immediately, you are unlikely to be able to remember all of the defects and the data for each one. This information is critical to your ability to perform effective personal reviews, and defect prevention (which we will discuss later).

## Defect Categorization and Analysis

Categorizing defects allows you to group them for convenient analysis. Without reasonable categories, your defect data will be a mass of data from which it would be hard to glean patterns. The PSP suggests a preliminary categorization scheme, but it also teaches engineers to modify that scheme based on their actual defect data.

Pareto analysis of categories helps each engineer to identify actions that are warranted based on his or her own data. For example:

- Re-working the defect categories may be indicated if there is:
- An ill-focused category into which many defects fall or
- A number of closely related categories that each has very few defects;
- Changes to your processes may be indicated in order to catch defect types that:
- Occur quite often or
- Require a lot of effort to remove.

The changes to your processes that can target specific defect types fall into two main categories:

- Early defect removal – Changes to your review (or other defect detection) processes can be designed to improve their effectiveness at detecting specific defect types, or
- Defect prevention – Changes to your design or coding processes can eliminate certain defect types. We will discuss this next.

## Defect Prevention

Often, when we identify problems with the way we do our jobs, our first reaction is to resolve to try harder or work more carefully. While such resolve may produce some immediate improvement, in the long run just trying harder will not result in lasting significant change. As soon as other issues attract our attention, our resolve will refocus on those issues, and the initial improvements are likely to slip away.

Preventing defects requires that we make changes to the processes we use to do the job. Those processes then provide the continuous and unwavering resolve we need to make the improvements permanent, and prevent the problem we are attempting to eliminate from returning.

Once you have identified a defect type that is a candidate for prevention, you might eliminate it be doing something like this:

1. Focus on one particularly troubling type of defect. Trying to attack several problems at once is likely to doom your effort to failure. By attacking one single important issue at a time, you can maintain the focus that is more likely to result in success.
2. Identify the conditions under which that defect type occurs. Prevention requires that you understand *why* you inject that particular type of defect. When does it happen? Under what conditions? What normally are the circumstances at the time? Remember that defects are the result of your human errors. So we are searching for the root cause of those errors.

3. Consider how to change the process you use. Your objective is to eliminate the defects that result from your errors. You can accomplish this in one of two ways:
   - Eliminate the error. For example, I eliminated begin/end mismatches by changing my coding process: I now write “Begin” and “End” at the same time, then go back and fill in the code between them.
   - Change the situation so that your error does not result in defects. For example, you might avoid the consequences of forgetting to declare variables by using an editor that highlights undeclared variables as you type (much like the spelling checker in Word).
4. Test the process change and collect data. Don’t just assume that the changes you decide on will be effective. Try them out on at least one pilot project and collect the necessary data to understand if they are effective. If they are not effective, return to step 3 with the new information you collected from the pilot test to try to develop a better solution.
5. If the process change is effective, permanently adopt it. Change your written process description, checklists or other mechanisms you may use to assure that you follow your new process. (If you have never written your process steps down, doing so may be your first step to avoiding errors!)

The sources of problems among different software engineers are as varied as their personalities. But from my experience teaching the PSP, I have discovered that for many software engineers, a highly effective way to prevent a variety of defects is to formalize the design process and/or design notation. Most of us spend insufficient time working through the details of our program designs, and this results in some of our most expensive and frustrating defects. The PSP recommends a set of design templates that will assure that all of the important parts of the design have been examined.

While defect prevention provides the greatest benefit, the reality is that we will not be able to prevent most of our defects. As we said earlier, because we are human we are prone to error, and those errors naturally result in defects. For this reason, the rest of this article will focus on ways to detect defects earlier in the development process, when they are less expensive to remove.

## Review Checklists

Focused checklists are the single most important tool you can use to assure the effectiveness of both personal and peer reviews. While the review process itself will assure that you do the requisite things, only a checklist can ensure that you don’t miss any important detail. Peer reviews will be discussed in the next section, so the remainder of this section will focus on checklists for personal reviews.

To get the maximum benefit from your personal reviews, your checklists must be focused and personalized:

- Your checklists must check for the defects you normally inject. That means that your checklists *must* be based on your personal defect history, and not on other people’s performance or your organization’s norms. Your checklist should check for compliance with organizational standards only to the extent that you violate them.
- Each of your reviews (high-level design, detailed design, code) needs a different checklist. Each checklist should include only the defects that you normally inject in the immediately preceding phase. Do *not* check for defects that escape prior reviews. (If prior reviews are consistently missing certain defects then fix those reviews.)

- If you use multiple programming languages, then you need different detailed design and code review checklists for each language. Although there is likely to be some overlap or duplication between checklists for different languages, it is important that each checklist fully and exclusively support the reviews for which you use it.
- Your choice of items for each checklist must be based upon Pareto or other analysis of your defect data. It would be infeasible to check for every possible defect. To make the best use of your review time, make sure you are checking for the defects that you are most likely to have injected. It is best to limit each checklist to one page. This means that you will be able to check for no more than 50 or so items in each review.

Just having a checklist doesn't automatically improve your review effectiveness. You must learn to use it systematically. For example, most people find that their reviews are most effective when they scan the work product for one checklist item at a time, rather than reading through the work product once looking for all checklist items. While it may seem that scanning through the work product multiple times is wasteful, this turns out not to be the case, because when you are looking for only one thing, you can scan at a relatively high rate.

The most important thing to keep in mind when doing personal reviews is to avoid rushing. The PSP teaches engineers to review code no *faster* than 200 LOC per hour. Most people's data shows that when review rates are higher than this, effectiveness becomes spotty, and more defects escape the reviews. Your objective in the reviews is to assure yourself that the defects for which you are checking are actually absent from the work products. The review must be careful enough so that checking off a checklist item is a personal certification that the defect type it represents is absent from the work product.

The final step in assuring the effectiveness of your reviews is to periodically study your defect data to see what it says. Look for types of defects that are consistently escaping your reviews, then consider what it may be about your review process or checklist that is allowing this to happen. For example:

- The defect type may not be represented on your checklist, suggesting that it should be added. If your checklist is already a full page long, this would mean deciding which checklist item must be dropped to accommodate a new one. Naturally, you should drop one that never or rarely results in identifying defects.
- If your checklist includes an item that covers this defect type, then you may want to re-word the item or split it into two or more items so that your attention is focused more precisely during reviews.
- Finally, the fault may lie with the way you use the checklist. A change to your review process may provide the improvement you need.

The purpose of your review checklist is to provide the focus that will assure that the time you spend doing reviews is effective in improving the quality of the work products you produce.

## Peer Reviews and Inspections

If your organization consistently does peer reviews of technical work, then you should make sure that you take full advantage of them. While personal reviews can find up to 80% of the defects in your work products, the time spent in peer reviews is still very much worth while. By combining effective peer reviews with effective personal reviews, you can remove more that 90% of the defects from your software before the first test!

Peer reviews can take a wide variety of forms. The most representative of those forms are these:

- Walk-throughs involve the author walking through the work product and explaining what was done and why before a group of peers. The peers interject questions about the work product to understand it better and to identify defects. These reviews require the least time of any of the peer review types. However, they tend to be a rather poor vehicle for identifying defects, being more valuable for educational purposes and assuring that team members understand each other's work.
- Full reviews involve peers examining the work product independently, then presenting questions, concerns and defects to the author in some way. The feedback mechanism can be as informal as e-mail or as formal as a walk-through meeting. The problem with this review method usually lies in the reviewers spending enough time in independent examination. Even when a formal feedback meeting is held, peers will often have spent little or no independent time on the work product. The effectiveness of these reviews can be highly variable, and often depends on the level of stress in the organization.
- Inspections (Often called "Fagan Inspections") are the most formal of the peer review types. Participants receive training, are assigned formal roles and are required to fully participate in the process. Inspections are generally checklist-based, with each person spending time in independent examination. The formal meeting is not held unless all participants have completed their examination of the work product. The inspection meeting includes a reader (not the author) walking the reviewers through the work product, with the reviewers commenting or questioning on each part. Questions that can not be quickly resolved in the meeting are assigned to individuals for resolution, and all identified defects are entered into a defect tracking system to assure they are corrected.

The most effective of the various review forms tends to be formal inspections. Although they are more work to establish and require more of people's time to do, they are also the most effective at finding defects, so they result in lower over-all cost of quality. Even if your organization does not have a formal peer review program, you can still capitalize on this powerful tool by asking your peers for their opinions on an informal basis. Obviously, you can not do formal inspections this way, but you can ask your peers to do independent examination and even attend review meetings. Often these sorts of informal reviews result in the desire within the organization to institute a more regular and formal peer review program.

## Personal Quality Planning

> If you don't know where you are going, any road will do.
> If you don't know where you are, a map will not help
>
> Watts S. Humphrey

High quality at low cost does not just happen; You must plan for it! As suggested by Watts Humphrey, there are two parts to planning: knowing where you want to go, and knowing where you are.

"If you don't know where you are going, any road will do."

Most of us never consider Personal Quality Planning. We may be unhappy with the quality of our work. We may be aware of the damage that poor quality does to our reputations or our company's bottom line. We may wish we could spend less time debugging and fixing problems; but we tend to be unsure what we can do about these things. This article provides lots of ideas about things that we can do, but taking advantage of these ideas requires that we take action. Deciding exactly what actions to take can be difficult to do unless you have a plan.

First, you must decide what quality levels you want to achieve. While absolute zero defects springs to mind (and is a laudable long-term goal), it is not a reasonable goal to strive toward. After realizing that you are human and prone to error, you must decide on a goal that is aggressive and challenging but achievable with available time and effort. If you are unsure of what might be a reasonable goal, a good starting point may be some level of incremental improvement over your current defect levels. (See, "If you don't know where you are…" below.)

Merely setting goals doesn't change much. To make material changes in your performance, you must make material changes in your actions. You must determine what steps you must take to achieve your goals, and what day-to-day process changes those steps imply. Do you need to log and track your defects? Do you need to do reviews you currently skip? Do you need to change your approach to certain tasks?

"If you don't know where you are, a map will not help."

Even if you know the quality that you want to achieve, you will still be lost until you know your current performance. What quality level do you normally achieve? … at what costs? … using what methods? If you can not answer these questions, you will not know where you are falling short of your goals or what direction you need to move to reach them. You won't even know when you **have** achieved your goals!

So, you must identify important attributes of your performance. What do you need to measure, and at what level of detail? For example, if you are going to log defects, what pieces of information will be important or useful? What will you care about? What will help you to determine how to improve your process?

Finally, you must actually measure your performance; collect interesting data over time and periodically examine it to see how you are progressing against your goals. If you are not progressing, what changes can you make to your personal processes to improve your performance? If you are progressing, do you expect the improvement to continue? And when you've reached your goal, celebrate! Then check your long-term goals to see if you need to set a new goal and continue improving your work.

Improving your work is hard but rewarding work. It requires good plans, specific actions, and measurement of your progress. Without these things, your best efforts are merely a shot in the dark

## Personal Process Improvement

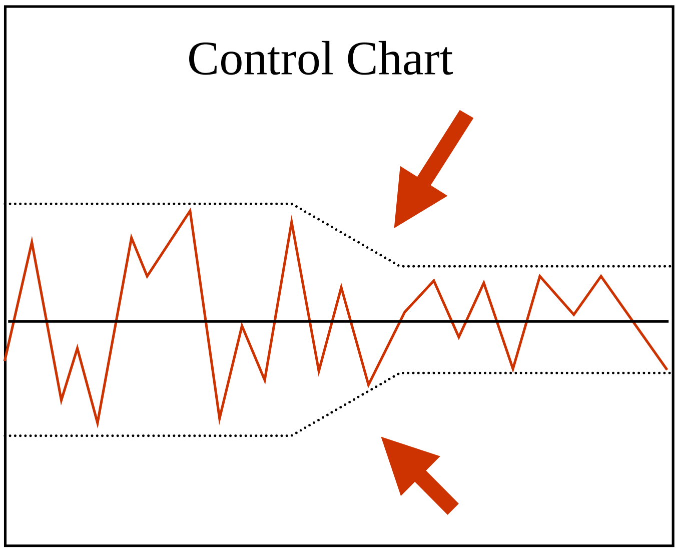
We have been talking about improving our personal processes, but have not yet looked closely at what this means. Every process (personal or otherwise) has a natural capability and a natural range of variation. This is easiest to see in a control chart like those used in manufacturing. (See the figure.)

The solid black line indicates the capability of the process. This is the expected result of the process. The dotted black line indicates the range of variability of the process. This is the range around the process capability within which the actual results generally stay. The red jagged line shows the actual performance of the process in specific cases, with time running from left to right.

The first step in improving a process is to gain control over it. A process that is out of control does not stay within an expected range of variation; it is essentially unpredictable. For most of us, project schedule estimation is like this. Each estimate we make is wrong by a different and unpredictable amount, sometimes over, mostly under, and sometimes by several hundred percent.

Gaining control over a process includes things like understanding the steps in the process, assuring that we do them consistently and mitigating outside influences on the process. For many things, gaining control will mean actually writing a description of the process steps, and creating a checklist or other mechanism to help us to remember to do the steps in the right order each time. Gaining control over the process results in its variability being limited to its natural capability.

Only after you have a process under control can you think about improving it. Improving a process involves things like changing the steps in the process, changing the way the steps are done, and enhancing the skills and tools that you use in the process. The net result of improving a process is a narrowed range of variability (as indicated by the big arrows, above). An improved process has a smaller range of variability, and is more predictable than it had been before.



The important point of this discussion is that you can not improve a process before it is under control. Your first focus must be to bring the process under control. Only after it is under control can you improve it.

## Using Your Personal Quality Plan

A Quality Plan is a waste of effort unless you use it to manage quality. While the thought that goes into making the plan has value, the real value of a plan is in the effect it has on your work. If you make a plan and set it aside until the work is done, then you are losing the major value of the plan. The best use of your quality plan will be in validating each step of your process; that is, using your quality plan to determine if you have done a sufficient job at each step in your process before you move on to the next step.

We tend to think of “validation” as being certain types of activities, like reviews or tests. In fact, *any* activity can and should be validated, even review and test activities. This kind of validation allows you to manage what you do so you can achieve the desired goals.

For example, before moving on to the code review, we should take a few minutes to validate our coding activities by asking questions like:

- Did I build all of the parts I intended to build?
- Is the product as big as I expected it to be?
- Do the pieces fit together as I intended?
- Do I need to take any corrective action?

With our answers to these questions, we know if we are ready to move to the next step and validate the code itself in our code review. Then, after the code review, we should validate the review by asking questions like:

- Did I find fewer (or more) defects than I expected?
- Did I spend as much effort on the review as I should?
- Is the defect rate consistent with my quality target?
- Do I need to take any corrective action?

By using our quality plan in these validations of the steps in our process, we can detect when things begin to go awry, and take corrective action before the whole project is out of control.

**Post-Project Data Analysis**

Whether your organization does post-project reviews (sometimes called “postmortem” or “retrospective” reviews), managing your personal quality performance will generally require that you spend some time with your personal data at the end of each project. A small amount of time at the end of the project can yield major rewards in the next one.

At the very least, you should take a few minutes at the end of each project to assure that all interesting data about the project has been recorded and stored in a safe place. The more time that has passed since you did the work, the less likely it will be that you will be able to reconstruct the data from memory. While you should always strive to record data as you work, you should use the post-project time to assure that nothing has been missed, and fill in any holes to the best of your memory.

The data you collect about your project can be a gold mine. Make sure you analyze it on a regular basis and understand what it is telling you. With the right data stored in a usable form, you can do many things. For example, use your data to:

- <u>Plan your next project.</u> Your best indicator of how you will perform on your next project is how you did on the last one. The best way to estimate product size, effort required and schedule is to compare the new project to others you have done and use your actual performance as a guide. (Don’t expect your performance to be significantly different this time unless the project is very different, or you have made material changes to your process.)
- <u>Set quality goals.</u> In the interest of making incremental improvement in the quality of your software, you will want to use your historical performance as the basis for setting quality goals. But realize that each project will not be better than the prior one was. There will be a certain amount of variability, but you want to aim for improvement over the long run.
- <u>Defend your plans.</u> If your plans are base in historical fact, you will be better able to defend them to your management. Management will always pressure you to promise too much; but

if you can show that you understand your capability and will deliver what you promise, they will learn to respect your estimates and plans.

- <u>Identify candidate processes for improvement.</u> When your data shows that a process is either out of control, or has too wide a range of variation, you should use that information to look for ways to change the process to bring it into an acceptable range of variation.
- <u>Evaluate the effectiveness of process changes you have made.</u> Each process change is not guaranteed to actually improve things. After making any change, you should carefully measure the results of the change to assure it had the desired effect and no undesirable side effects.

## Personal Quality Management and the PSP

The Personal Software Process (PSP) teaches software engineers how to use a variety of disciplined practices, including the Personal Quality Management techniques described in this article. By learning to apply these disciplined methods, programmers can begin to make the transition from programming as an ill-defined craft toward software as an engineering discipline.

## References

[Humphrey] Humphrey, Watts, PSP(sm): A Self-Improvement Process for Software Engineers, Addison Wesley, 2005

[Hays] Hayes, Over “The Personal Software Process (PSP): An Empirical Study of the Impact of PSP on Individual Engineers” CMU/SEI-97-TR-001

**Agile Job Offer: Senior Software Process Engineer.** Pearson Educational Measurement is the world's largest educational test processor and a leader in developing educational software. We currently have an opening for a Senior Software Process Engineer to train and coach software development teams, product owners, and facilitators (ScrumMasters) in the use of agile software development values, principles, and practices. This position will work with software development teams and other engineers/architects to ensure that processes fit within the larger context of all business processes, metrics, and QA activities. The ideal candidate would have experience as a ScrumMaster or implementing agile practices in a large software organization. Qualified candidates will have a Bachelor's degree and a minimum of four (4) years experience with software development or process improvement. Candidates must have knowledge and experience with agile methods and the ability to lead projects. Pearson Educational Measurement is located in Iowa City, Iowa, a vibrant University community. We offer competitive salaries, exceptional relocation program a comprehensive benefit package, opportunity for advancement and a great place to work! If you are interested in joining our team or would like to learn more about us, check us out at: www.pearsonedmeasurement.com. Please apply on-line to posting "Senior Process Engineer" (# 19177) or send your resume to: Margo.Massey [@] Pearson.com

Are you looking for the right development tools for your task. A new directory of tools related to software development has been launched. It covers all software development activities: programming (java, .net, php, xml, etc), testing, configuration management, databases, project management, modelling, etc.

http://www.softdevtools.com/

Advertising for a new Web development tool? Looking to recruit software developers? Promoting a conference or a book? Organizing software development training? This clasified section is waiting for you at the price of US $ 30 each line. Reach more than 45'000 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 30'000 visitors/month of our web sites! To advertise in this section or to place a page ad simply http://www.methodsandtools.com/advertise.html

**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