METHODS & TOOLS

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

Summer 2011 (Volume 19 - number 2) www.methodsandtools.com

When Agile Values Meet Mere Mortals Behavior

We are celebrating the tenth anniversary of the Agile Manifesto. As Agility has become popular in software development organizations, you can see more and more material about the fact that Agile has "lost its soul". People would be more "doing Agile", that is following blindly some practices, than "being Agile". Individuals are the main pillar of the Agile culture as "individuals and interactions" is the first of the Agile manifesto values. As the issues in plan-driven project could be attributed to a lack of consideration of human factors, the problems in Agile project will rather the dependence on people. Individuals are the most important success factor in software development projects, but they are also the most important failure factor, due to the inherent weaknesses of our human condition. Trying to blame the "Waterfall" for all failures is a nice way to say that every project could succeed, but things might be a little bit more complicated than this. Agile requires that developers are self-disciplined and emphasize team achievement over their individual interest. The product owner is a second pillar for the success of agile project. It is difficult to achieve meaningful results if the end users are not able to explicit and prioritize requirements, or worse are not very concerned by the outcome of the project. You need to find Agile users on the other side of the interactions. Finally, it is more difficult to operate in an Agile way within an organization that doesn't necessarily shares or implements the value of the Agile Manifesto. Trust and openness are important factors if you need to give more weight on "individuals and interactions". The emergence of Agile approaches ten years ago had a positive impact on a domain that was often more concerned with processes and resources than individuals. At that time, the word was spreading that Agile was the key to every project success. Surveys were showing improved productivity, quality, customer satisfaction and success rate for Agile projects and those who failed were just not Agile enough. Today Agile might be facing the wall of the Pareto law: converting the 20% of more favorable people and project contexts might have created 80% of the improvements. Improving the quality of software development activity will always face the limits of our human condition and cultural influences. We might consider ourselves "rational", but we are more often that we think looking for silver bullets to solve our problems. Agile is not a silver bullet and if we have to retain only one principle of Agile is that it is important to respond to change.

Inside

Continuous Delivery Using Build Pipelines With Jenkins and Ant

James Betteley, http://jamesbetteley.wordpress.com
Caplin Systems

My idea of a good build system is one which will give me fast, concise, relevant feedback, but I also want it to produce a proper finished article when I've checked in my code. I'd like every check-in to result in a potential release candidate. Why? Well, why not?

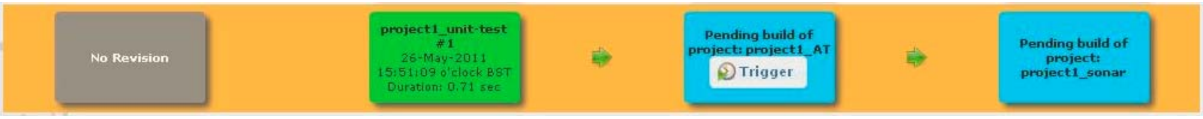
I used to employ a system where formal release candidates were produced separately to my check-in builds (also known as "snapshot" builds). This encouraged people to treat snapshot builds as second rate. The main focus was on the release builds. However, if every build is a potential release build, then the focus on each build is increased. Consequently, if every build could be a potential release candidate, then I need to make sure every build goes through the most rigorous testing possible, and I would like to see a comprehensive report on the stability and design of the build before it gets released. I would also like to do all of this automatically, and involve as little (preferably none at all) human intervention as possible.

This presents us with a problem: we want instant feedback on check-in builds, and to have full static analysis performed on them and yet we still want every check-in build to undergo a full suite of testing, be packaged correctly AND be deployed to our test environments. Clearly this will take a lot longer than I'm prepared to wait! The solution to this problem is to break the build process down into smaller sections, and run them in parallel if necessary.

Pipelines to the Rescue!

The concept of build pipelines has been around for a few years (Sam Newman is allegedly credited with first documenting the idea back in early 2005). So it's nothing new, but it's not yet standard practice, which is a pity because I think it has some wonderful advantages. The concept is simple: the build as a whole is broken down into sections, such as the unit test, acceptance test, packaging, reporting and deployment phases. The pipeline phases can be executed in series or parallel, and if one phase is successful, it automatically moves on to the next phase (hence the relevance of the name "pipeline").

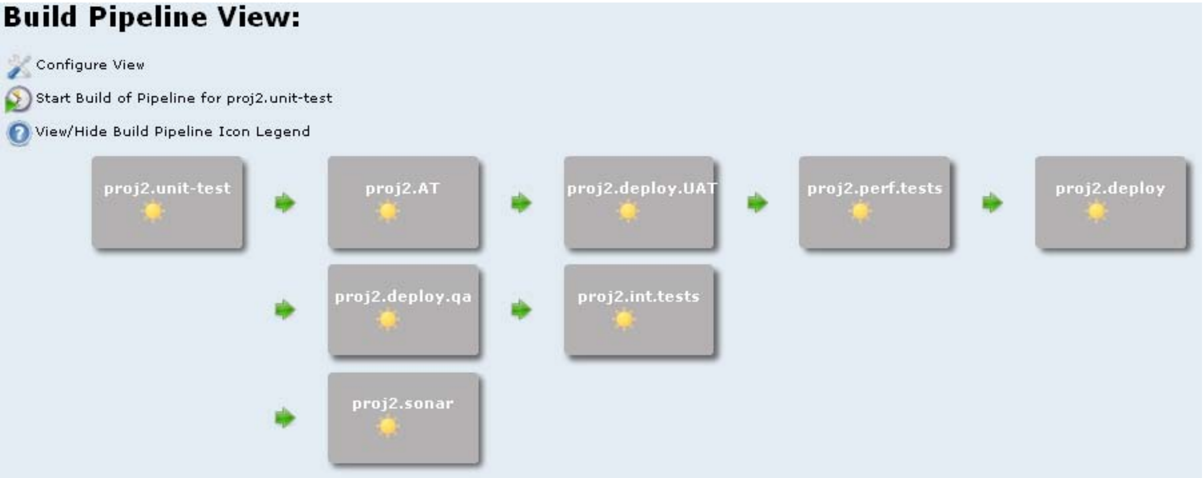
A good build pipeline will have a relatively high degree of parallelization. That is to say, the pipeline tasks won't simply all run in series – otherwise you're doing little more than daisy-chaining together a collection of build jobs, and the value of the system is greatly reduced.



In the example above, our pipeline is just a series of individual tasks. This might be sufficient for small, quick-running builds and deployments. However, if this was a large project and our unit and acceptance tests were very slow running, and our deployment steps took a long time to complete, then this pipeline may take hours to complete. Chances are you will have finished work for the day and gone home before the final stage executed! This is far from ideal, obviously, because if there's a problem with any part of my pipeline, you want to know about it as soon as possible so that you can fix it.

• Deliver quality ... ALM ... software development ... ALM is Your advantage by streamlining processes •

A good practice is to parallelize the tasks where possible. Acceptance tests, unit-test coverage and static analysis can often be performed in parallel. My own personal preference is to run unit-tests as a first step, and if they pass then run as many other tasks as possible in parallel. This assumes that the unit tests take less than 3 minutes to complete. If they take longer than 5 minutes, I would recommend running the unit tests and acceptance tests in parallel on check-in.
The example below shows a pipeline with parallelization:



In this example we’re executing the Acceptance Tests (AT) at the same time as running the build reports and deploying our project to our QA/integration test environment. In the next stage we’re executing integration tests while simultaneously deploying our project to the UAT environment. This will then kick off the performance tests.

Continuous Delivery

Continuous delivery has also been around for a while. It’s basically the logical evolution of Continuous Integration. The practice of Continuous Integration covers many of the fundamentals of Continuous Delivery, for instance, the concepts of unit testing, static analysis, “failing fast” and automated testing are core to Continuous Integration. Continuous Delivery simply builds on this foundation and extends it more into the domain of deployment and project delivery. Continuous Delivery, while not a new concept, has enjoyed a recent increase in exposure, thanks largely to the publication of Jez Humble and David Farley’s excellent book "Continuous Delivery". Again the concept is simple, roughly speaking it means that every build gets made available for deployment to production if it passes all the quality gates along the way. Continuous Delivery is sometimes confused with Continuous Deployment. Both follow the same basic principle, the main difference is that with Continuous Deployment it is implied that each and every successful build *will* be deployed to production, whereas with continuous delivery it is implied that each successful build will be made *available* for deployment to production. The decision of whether or not to actually deploy the build to the production environment is entirely up to you.

Continuous Delivery using Build Pipelines

You can have continuous delivery without using build pipelines, and you can use build pipelines without doing continuous delivery, but the fact is they seem made for each other. Here's my example framework for a continuous delivery system using build pipelines.

I check some code in to source control - this triggers some unit tests. If these pass it notifies me, and automatically triggers my acceptance tests AND produces my code-coverage and static analysis report at the same time. If the acceptance tests all pass my system will trigger the deployment of my project to an integration environment and then invoke my integration test suite AND a regression test suite. If these pass they will trigger another deployment, this time to UAT and a performance test environment, where performance tests are kicked off. If these all pass, my system will then automatically promote my project to my release repository and send out an alert, including test results and release notes.

So, in a nutshell, my "template" pipeline will consist of the following stages:

- Unit-tests
- Acceptance tests
- Code coverage and static analysis
- Deployment to integration environment
- Integration tests
- Scenario/regression tests
- Deployments to UAT and Performance test environment
- More scenario/regression tests
- Performance tests
- Alerts, reports and Release Notes sent out
- Deployment to release repository

Introducing the Tools

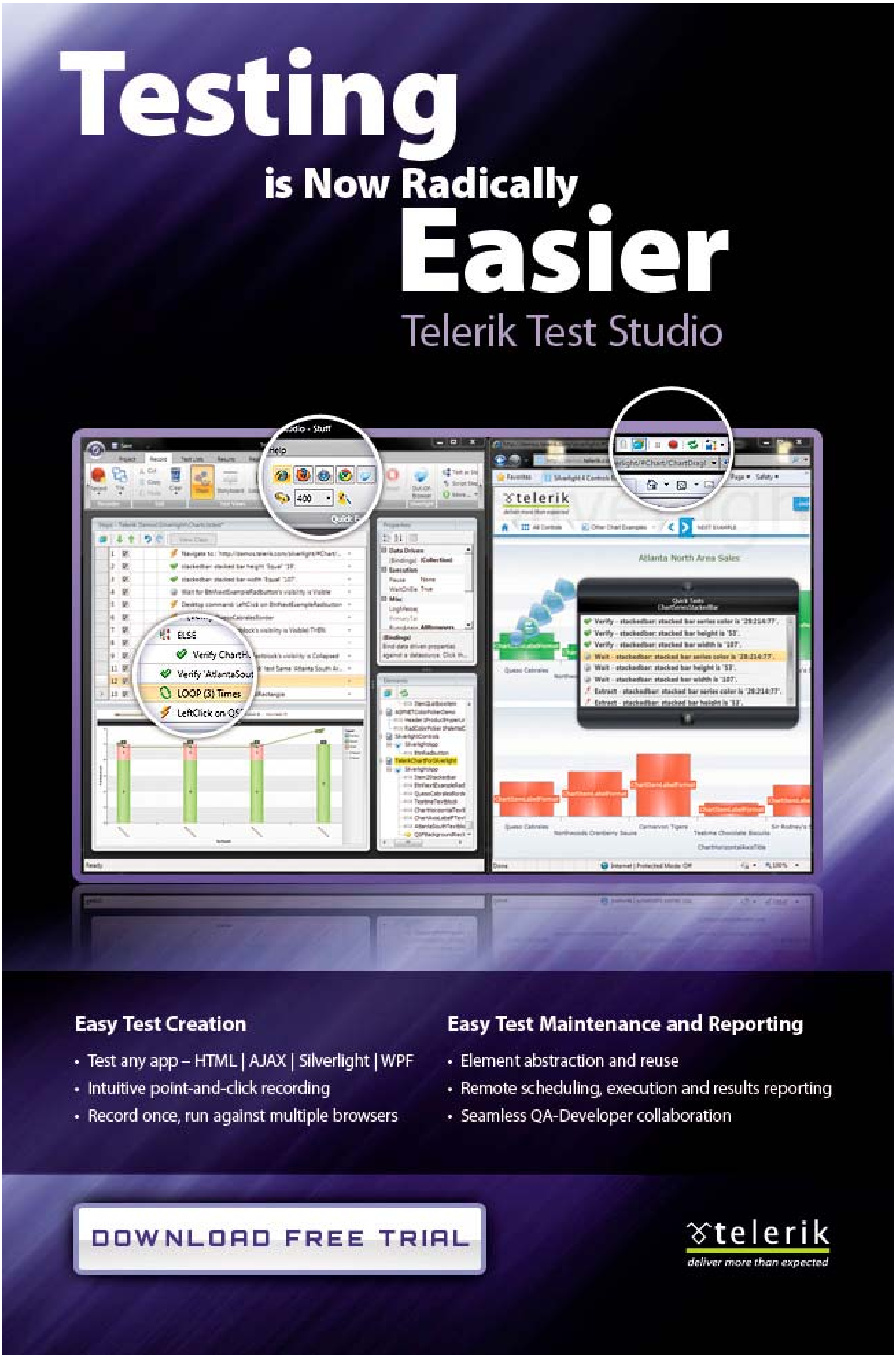
Thankfully, implementing continuous delivery doesn't require any special tools outside of the usual toolset you'd find in a normal Continuous Integration system. It's true to say that some tools and applications lend themselves to this system better than others, but I'll demonstrate that it can be achieved with the most common/popular tools out there.

Who is this Jenkins person?

Jenkins is an open-source Continuous Integration application, like Hudson, CruiseControl and many others (it's basically Hudson, or was Hudson, but isn't Hudson any more. It's a trifle confusing*, but it's not important right now!). So, what is Jenkins? Well, as a CI server, it's basically a glorified scheduler, a cron job if you like, with a swish front end. Ok, so it's a very swish front end, but my point is that your CI server isn't usually very complicated, in a nutshell it just executes the build scripts whenever there's a trigger. There's a more important aspect than just this though, and that's the fact that Jenkins has a build pipelines plugin, which was written recently by Centrum Systems. This pipelines plugin gives us exactly what we want, a way of breaking down our builds into smaller loops, and running stages in parallel.

Ant

Ant has been possibly the most popular build scripting language for the last few years. It's been around for a long while, and its success lies in its simplicity. Ant is an XML based scripting language tailored specifically for software build related tasks (specifically Java. Nant is the .Net version of Ant and is almost identical).


Testing
is Now Radically
Easier
Telerik Test Studio
DOWNLOAD FREE TRIAL
telerik
deliver more than expected

Sonar

Sonar is a quality measurement and reporting tool, which produces metrics on build quality such as unit test coverage (using Cobertura) and static analysis tools (Findbugs, PMD and Checkstyle). I like to use Sonar as it provides a very readable report and contains a great deal of useful information all in one place.

Setting up the Tools

Installing Jenkins is incredibly simple. There's a debian package for Operating Systems such as ubuntu, so you can install it using apt-get. For Redhat users there is a rpm, so you can install via yum.

Alternatively, if you're already running Tomcat v5 or above, you can simply deploy the jenkins.war to your tomcat container.

Yet another alternative, and probably the simplest way to quickly get up and running with Jenkins is to download the war and execute: java -jar jenkins.war

This will run Jenkins through its own Winstone servlet container.

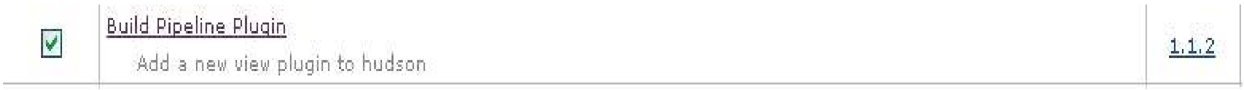
You can also use this method for installing Jenkins on Windows, and then, once it's up and running, you can go to "manage Jenkins" and click on the option to install Jenkins as a Windows Service! There's also a Windows installer, which you can download from the Jenkins website

Ant is also fairly simple to install, however, you'll need the java jdk installed as a pre-requisite. To install ant itself you just need to download and extract the tar, and then create the environment variable ANT_HOME (point this to the directory you unzipped Ant into). Then add ${ANT_HOME}/bin (or %ANT_HOME%/bin if you're on Windows) to your PATH, and that's about it.

Configuring Jenkins

One of the best things about Jenkins is the way it uses plugins, and how simple it is to get them up and running. The "Manage Jenkins" page has a "Manage Plugins" link on it, which takes you a list of all the available plugins for your Jenkins installation.

To install the build pipeline plugin, simply put a tick in the checkbox next to "build pipeline plugin" (it's 2/3 of the way down on the list) and click "install". It's as simple as that.



☑	Build Pipeline Plugin Add a new view plugin to hudson	1.1.2

The Project

The project I'm going to create for the purpose of this example is going to be a very simple java web application. I'm going to have a unit test and an acceptance test stage. The build system will be written in Ant and it will compile the project and run the tests, and also deploy the build to a tomcat server. Sonar will be used for producing the reports such as test coverage and static analysis.

The Pipelines

For the sake of simplicity, I've only created 6 pipeline sections, these are:

- Unit test phase
- Acceptance test phase
- Deploy to test phase
- Integration test phase
- Sonar report phase
- Deploy to UAT phase

The successful completion of the unit tests will initiate the acceptance tests. Once these complete, 2 pipeline stages are triggered:

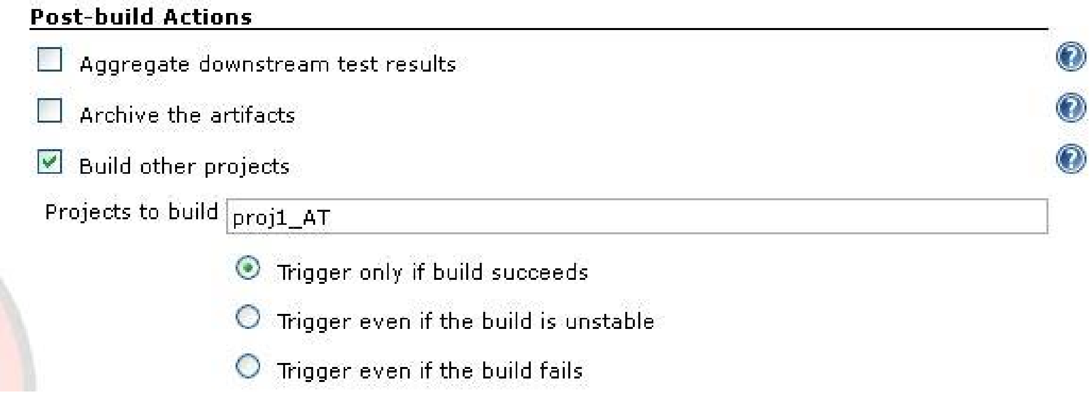
- Deployment to a test server

and

- Production of Sonar reports.

Once the deployment to the test server has completed, the integration test pipeline phase will start. If these pass, we'll deploy our application to our UAT environment.

To create a pipeline in Jenkins we first have to create the build jobs. Each pipeline section represents 1 build job, which in this case runs just 1 ant task each. You have to then tell each build job about the downstream build which is must trigger, using the "build other projects" option:



Post-build Actions

□ Aggregate downstream test results

□ Archive the artifacts

☑ Build other projects

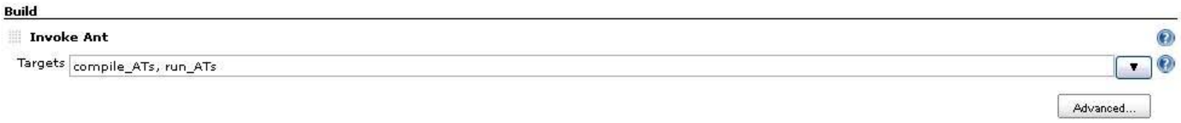
Projects to build: proj1_AT

● Trigger only if build succeeds

○ Trigger even if the build is unstable

○ Trigger even if the build fails

Obviously you only want each pipeline section to do the single task it’s designed to do, i.e. I want the unit test section to run just the unit tests, and not the whole build. You can easily do this by targeting the exact section(s) of the build file that you want to run. For instance, in my acceptance test stage, I only want to run my acceptance tests. There’s no need to do a clean, or recompile my source code, but I do need to compile my acceptance tests and execute them, so I choose the targets “compile_ATs” and “run_ATs” which I have written in my ant script. The build job configuration page allows me to specify which targets to call:

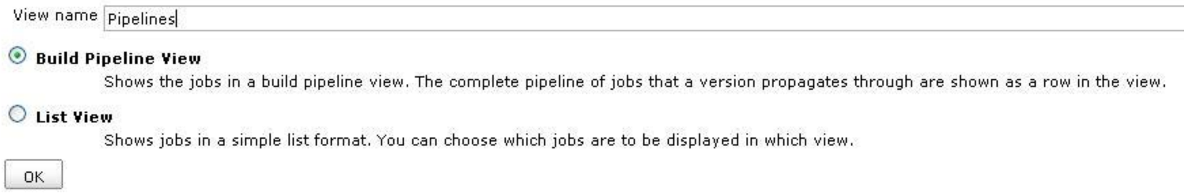


Build

Invoke Ant

Targets: compile_ATs, run_ATs

Once the 6 build jobs are created, we need to make a new view, so that we can start to visualise this as a pipeline:

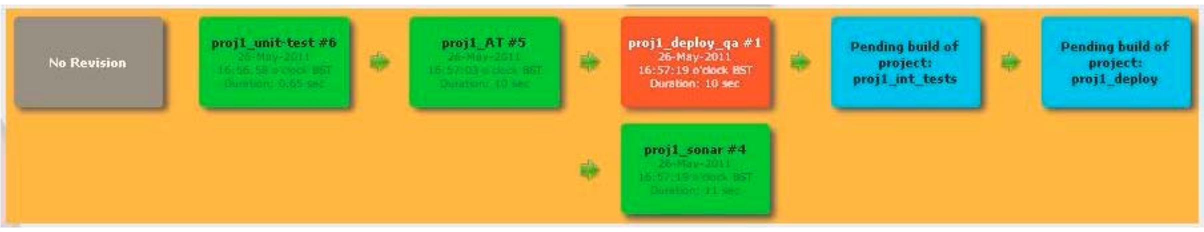


View name Pipelines

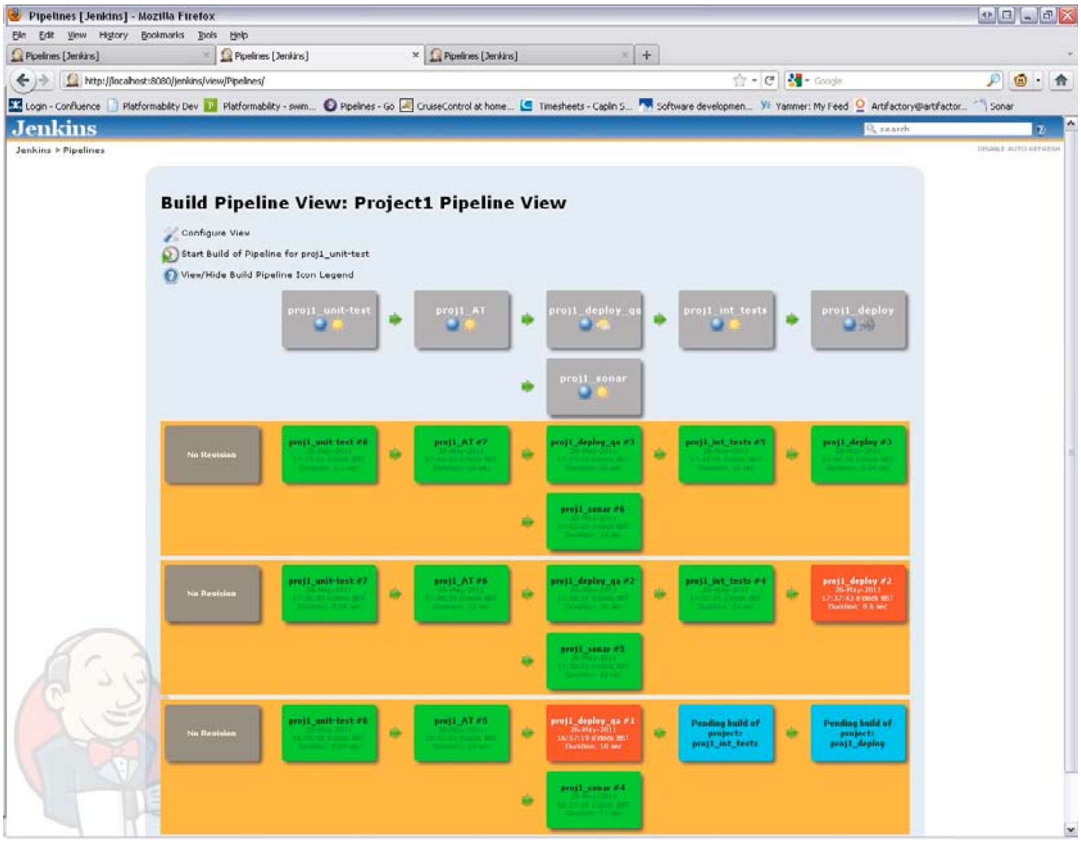
● Build Pipeline View
Shows the jobs in a build pipeline view. The complete pipeline of jobs that a version propagates through are shown as a row in the view.

○ List View
Shows jobs in a simple list format. You can choose which jobs are to be displayed in which view.

We now have a new pipeline! The next thing to do is kick it off and see it in action:



Oops! It looks like the deploy to QA has failed. It turns out to be an error in my deploy script. But what this highlights is that the sonar report is still produced in parallel with the deploy step, so we still get our build metrics! This functionality can become very useful if you have a great deal of different tests which could all be run at the same time, for instance performance tests or OS/browser-compatibility tests, which could all be running on different Operating Systems or web browsers simultaneously. Finally, I've got my deploy scripts working so all my stages are looking green! I've also edited my pipeline view to display the results of the last 3 pipeline builds:



Alternatives

The pipelines plugin also works for Hudson, as you would expect. However, I'm not aware of such a plugin for Bamboo. Bamboo does support the concept of downstream builds, but that's really only half the story here. The pipeline "view" in Jenkins is what really brings it all together. TeamCity also supports “Build Chains” which is the same as a downstream build. It also allows you to break down your builds into several parts which can be run in series or in parallel, which is much the same, in principle, as a pipeline. However, as with Bamboo this functionality isn’t encapsulated within a pipeline “view” yet.

CruiseControl, which is a very popular lightweight Continuous Integration System, also supports downstream builds, but beyond that it doesn’t offer itself to supporting build pipelines.

"Go", the enterprise Continuous Integration effort from ThoughtWorks not only supports pipelines, but it was pretty much designed with them in mind. Suffice to say that it works exceedingly well, in fact, I use it every day at Caplin Systems where I work. On the downside though, the enterprise version costs money, whereas Jenkins doesn't. However, there is a “free” alternative, the Go Community Edition, which still apparently has some support for pipelines. What Go (the enterprise edition) does offer though, is a wonderful facility for managing multiple build agents (at Caplin we’re running about 150 build agents) which allows us to create a build grid. It’s very important that the system should support multiple build agents as this allows you to run so many parts of the pipeline in parallel. Of course, it’s not too difficult to do this without the support of your CI system – that is to say, you can do a similar thing using Jenkins by explicitly instructing your scripts where to deploy your project or run your tests, but having your agents managed by the CI system is actually hugely valuable, and a lot easier in my opinion.

As far as build tools/scripts/languages are concerned, this system is largely agnostic. It really doesn’t matter whether you use Ant, Nant, Gradle or Maven, they all support the functionality required to get this system up and running (namely the ability to target specific build phases). However, Maven does make hard work of this in a couple of ways – firstly because of the way Maven lifecycles work, you cannot invoke the “deploy” phase in an isolated way, it implicitly calls all the preceding phases, such as the compile and test phases. If your tests are bound to one of these phases, and they take a long time to run, then this can make your deploy seem to take a lot longer than you would expect. In this instance there’s a workaround – you can skip the tests using –DskipTests, but this doesn’t work for all the other phases which are implicitly called. Another drawback with maven is the way it uses snapshot and release builds. Ultimately we want to create a release build, but at the point of check-in we want a snapshot build. This suggests that at some point in the pipeline we’re going to have to recompile in “release mode”, which in my book is a bad thing, because it means we have to run ALL of the tests again. As usual, there is a workaround! Axel Fontaine, a software development consultant in Munich, has written a great blog on how to create a release in Maven2 without using the release plugin. Alternatively, and this is my preferred solution, you can just make sure your CI builds are not snapshots, by running the Maven release prepare command in one of your pipeline stages.

* A footnote about the Hudson/Jenkins “thing”: It's a little confusing because there's still Hudson, which is owned by Oracle. The whole thing came about when there was a dispute between Oracle, the "owners" of Hudson, and Kohsuke Kawaguchi along with most of the rest of the Hudson community. The story goes that Kawaguchi moved the codebase to GitHub and Oracle didn't like that idea, and so the split started.

References

DevOps Ramblings http://jamesbetteley.wordpress.com/

Caplin Systems Tech blog http://blog.caplin.com/

Continuous Delivery Book http://martinfowler.com/books.html#continuousDelivery

Dave Farley http://www.davefarley.net/

Jez Humble http://continuousdelivery.com/

Maven http://maven.apache.org/

Ant http://ant.apache.org/

Sonar http://www.sonarsource.org/

Jenkins CI http://jenkins-ci.org/

Hudson http://hudson-ci.org/

Centrum Systems http://www.centrumsystems.com.au/

GO CI from Thoughtworks http://www.thoughtworks-studios.com/go-agile-release-management

Bamboo http://www.atlassian.com/software/bamboo/

Nant http://nant.sourceforge.net/

Gradle http://www.gradle.org/

Sam Newman’s blog http://www.magpiebrain.com/

Cruise Control http://cruisecontrol.sourceforge.net/

Axel Fontaine’s blog http://www.axelfontaine.com/2011/01/maven-releases-on-steroids-adios.html

Everything You Always Wanted to Know About Software Measurement

Sue Rule, P. Grant Rule, s.rule @ smsexemplar.com
Software Measurement Services Ltd., www.smsexemplar.com

What Does Functional Size Measurement Mean?

When Grant Rule was working with early pioneers of agile methods back in the late 1980s, estimating was a big issue. No one, it seemed, knew how to determine accurately in advance how big a project was, how long it would take, or how much it would cost. This was particularly a problem for a software house delivering fixed price contracts. Focusing on rapid, iterative delivery to ensure the user gets the right software product is good; but at what price?

While advocates of Lean-Agile rightly emphasise the importance of delivering the right thing, it is essential that professional software managers also keep reliable accounts of software process performance for estimating costs, auditing performance, and improving productivity. Effectiveness is *delivering optimum value* for the *minimum cost*.

The solution to Grant's estimating issue, and many associated software project control issues, was the IFPUG function point measurement method developed by Alan Albrecht in 1979. Yet for various reasons, functional size measurement has failed to become widely adopted as a software project management tool. Perhaps it seems arcane - difficult - part of that top-heavy bureaucracy development teams want to be rid of. But as a result, estimating remains a widespread problem for Agile and non-Agile developers alike. Process performance remains hugely variable and unpredictable, with measurement often still the most neglected area of process improvement. By stripping away much of the dysfunctional and redundant process control that can accumulate around the software process, the current interest in Agile is bringing many of these issues to the surface. But many IT professionals seem stuck in a rut, ploughing new land with a newly Agile team but using the same blunt plough and running into the same obstacles.

CIOs, bid teams, development teams, vendor managers and legal advisors seeking better ways of contracting - particularly for Lean-Agile software development - are still faced with the dilemmas Albrecht was seeking to tackle. How do you know your development teams - in-house or outsourced - are delivering best value for money? How does a supplier manage the risk of a fixed price contract? How does he demonstrate competitive productivity levels and on-going improvement? How will Agile benefit *my* organisation? How does the business manage the risk if we can't detail the requirements up-front? How do you measure the benefits of outsourcing IT? Which technology should we choose? Will a change of technology affect the value delivered? What are the relative merits of buying it, building it or outsourcing it?

Alan Albrecht's answer to addressing these issues was to measure software size in terms of the output delivered to the user - its *functionality* - rather than simply in terms of the time and effort put into the development. Deriving a software size from the user's requirements which can then be mapped to the time, resources and budget needed to realise those requirements gives you the necessary objectivity to tackle the scope, project and performance control issues. It also takes you at least part way to measuring the business value and benefit delivered by software. At least you know how much it will cost and how long it will take to deliver the specified functionality; the software should do what it "says on the tin" and it should be delivered on schedule, within budget. If the customer has bought beans instead of tomatoes, or needed a three course meal and not a tin of beans at all, there is something amiss elsewhere in the process.

Prague and Amsterdam Goto Conferences - Click on ad to reach advertiser web site

Functional size measurement means the ability to make objective performance comparisons across teams, across market sectors, and between different suppliers. It means reliable productivity and pricing benchmarks. It means accurate baselines and measures of improvement. It means early and accurate estimations of software project cost and duration. That was why Grant Rule first adopted the technique back in the 1980s - and remains a knowledgeable advocate of it today.

Estimating

The predictability of software projects remains a major issue for many business managers. Agile can exacerbate this if delivered software products or software product components have to be routinely re-factored. Customers considering outsourced Agile development will rightly require some assurance of value for money, and suppliers need a reliable oversight of costs to ensure prices remain competitive. The ‘Story Point’ measures used by many Agile teams to estimate and manage projects are little more than a new take on our old friend, the measure of input. Estimates using Story Points rely on team knowledge and capability, based on individual experience. The process of arriving at them is even known as Planning Poker, an indication of the level of personal skill and informed guesswork required. Story Points cannot be compared between one team and another (let alone one market sector or one supplier and another). Their use either as an effective and reliable estimating tool or as objective and comparable measures of productivity is therefore very limited.

In 2010, Grant Rule published a short paper on using the most modern of the functional size methods - COSMIC - to derive accurate estimates from User Stories (used by Agile teams to capture requirements.) As measurement specialists, SMS recommend the use of COSMIC as the quickest, most reliable, and most easily learned method of introducing robust software estimating practice for developers using Agile or more traditional development practices.

Size Matters - Accurate Early Estimating

Project failure rates have been shown to relate strongly to the size of project undertaken, with the largest projects accounting for the highest failure rate. Rule’s Relative Size Scale was a table developed by Grant Rule to provide a quick and easy early range estimate of project size and cost in terms of clothing sizes - Small, Medium, Large, Extra Large. The scale uses benchmark data to provide a valuable and reliable early-stage feasability check.

COCOMO II Profiling - Accounting for Non-Functional Requirements

The Constructive Cost Model (COCOMO) is an algorithmic software cost estimation model developed by Barry Boehm. The model uses a basic regression formula, with parameters that are derived from historical project data and current project characteristics. COCOMO enables more refined estimates to be derived from simple unadjusted function points by taking into account differences in non-functional project attributes (Cost Drivers). Detailed COCOMO can also account for the influence of individual project phases.

COCOMO II was brought out in 2000, to adapt the original model for use estimating software projects in modern development conditions such as incremental development and rational unified process. COCOMO II is continually evolving to adapt to changing software methods.

What are Function Points?

Function Points measure the end users requirements in terms of data movements. This means that an early functional size estimate can be derived before any code has been written. They are then also used to measure the actual cost of developing the requirements, which can be compared directly to the estimate and used to calibrate a scale for a particular development environment. If development productivity figures are known, the cost of developing new software can therefore be estimated early enough to make direct comparisons to the cost of buying a software package, or simply using a non-technical solution. Function Point measures are independent of technology so can compare different suppliers, technologies and different teams to give an objective basis for a cost benefit decision.

Because they encompass both cost (development effort consumed) and benefit (requirements fulfilled), Function Points unlock the door to measuring productivity; velocity; quality; and value in software development.

- The capacity to measure the amount of **output produced for input provided** gives an objective **productivity** figure.
- Measured against time - E.g. **FP delivered per elapsed month** - function points provide objective and comparable measures of **velocity**.
- The number of defects detected over total size of delivery - i.e. **no of defects delivered over no of FP delivered** - gives a measure of **quality**.
- By comparing **costs, effort and time expended per function point delivered**, we can also demonstrate the huge waste generated by ineffective software processes compared to the standards of measurable efficiency shown by **effective**, Right-shifted organisations. Although a long way from being a comprehensive measure of value delivered to the customer, this offers a measure of the component of value for which the software developer is solely responsible. Ensuring the functionality delivered is aligned to business need is the joint responsibility of the business user and the development team and outside the scope of functional size measurement.

A Comparison of Functional Size Methods

The most widely used function point methods are IFPUG, NESMA, Mk II and COSMIC. Of these, IFPUG is based on the original method designed by Alan Albrecht. COSMIC is the newest method, developed by an international team of metrics experts to support modern software.

Where organisations have already made some investment in IFPUG capability, it may make sense to use this more widely known method. It is actively managed and maintained by the International Function Point User Group, with the latest version 4.3 released in January 2010. The Certified Function Point Specialist qualification is a recognised standard for IFPUG counting competency. There is considerable IFPUG benchmarking data available, although it is questionable how valuable some of the older data is in terms of benchmarking modern software performance. While the older functional size measures - typically, IFPUG - can be used for Agile projects, there are significant problems. IFPUG counting is not the easiest process to learn or apply, and even with trained practitioners, time can be wasted debating fuzzy areas. Interpretation of the IFPUG counting rules for complex modern software environments can be tricky, sometimes resulting in differences of opinion between supplier and customer.

COSMIC offers all the advantages of functional size measurement, without some of the shortcomings of earlier methodologies which were not designed with 21st century software in mind. It will prove easier to introduce, easier for both business users and software developers to understand and apply, and a more effective communication mechanism for negotiating the delivery of value. It is an established standard with recognised training qualifications. There is a growing repository of benchmark data available. See the table below for a more detailed comparison.

Types of measurement scale and permissible operations using them

The type of scale depends on the nature of the relationship between values on the scale. Four types of scale are commonly defined:

Nominal – arbitrary labels, classification data, no ordering – the measurement values are categorical but it makes no sense to state that one category is 'greater than' another. For example: Yes/No; Black/White/Yellow/Red; male/female, animal/vegetable/mineral; the classification of defects by their type.

Ordinal – ordered but differences between values are not important – the measurement values are rankings. For example: restaurant 'star' ratings; political parties on left to right of the spectrum are given labels Red, Orange, Blue; Likert scales that rank 'user satisfaction' on a scale of 1..5; the assignment of a severity level to defects.

Interval – ordered, constant scale, but no natural zero – the measurement values have equal distances corresponding to equal quantities of the attribute. For example: dates, temperature on Celsius or Fahrenheit scales – differences make sense, but ratios do not (e.g., 30°-20° = 20°-10°, but 20° is not twice as hot as 10°! Other examples: cyclomatic complexity has the minimum value of one, but each additional path increments the count by one.

Ratio – ordered, constant scale, natural zero – the measurement values have equal distances corresponding to equal quantities of the attribute where the value of zero corresponds to none of the attribute. For example: height; weight; age; length; temperature on Kelvin scale (e.g. absolute zero = 0°K, and 200°K is twice as hot as 100°K); the size of a software source listing in terms of Non-Commentary Source Statements (or Source Lines Of Code).

The method of measurement usually affects the type of scale that can be used reliably with a given attribute. For example, subjective methods of measurement usually only support ordinal or nominal scales.

Only certain operations can be performed on certain scales of measurement. The following list summarizes which operations are legitimate for each scale. Note that you can always apply operations from a 'lesser scale' to any particular data, e.g. you may apply nominal, ordinal, or interval operations to an interval scaled datum.

Nominal Scale. You are only allowed to examine if a nominal scale datum is equal to some particular value or to count the number of occurrences of each value. For example, gender is a nominal scale variable. You can examine if the gender of a person is F (female) or to count the number of Ms (males) in a sample. Valid statistics: mode, chi square.

Ordinal Scale. You are also allowed to examine if an ordinal scale datum is less than or greater than another value. Hence, you can 'rank' ordinal data, but you cannot 'quantify' differences between two ordinal values. For example, political party is an ordinal datum with the Liberal Democratic Party to the left of the Conservative Party, but you can't quantify the difference. Another example are preference scores, e.g. ratings of eating establishments where 10=good, 1=poor, but the difference between an establishment with a 10 ranking and an 8 ranking can't be quantified. Valid statistics: mode, chi square, median, percentile.

Interval Scale. You are also allowed to quantify the difference between two interval scale values but there is no natural zero. For example, temperature scales are interval data with 25C warmer than 20C and a 5C difference has some physical meaning. Note that 0C is arbitrary, so that it does not make sense to say that 20C is twice as hot as 10C. Valid statistics: mode, chi square, median, percentile, mean, standard deviation, correlation, regression, analysis of variance.

Ratio Scale. You are also allowed to take ratios among ratio scaled variables. Physical measurements of height, weight, and length are typically ratio variables. It is now meaningful to say that 10 metres is twice as long as 5 metres. This ratio holds true regardless of which scale the object is being measured in (e.g. metres or yards). This is because there is a natural zero. Valid statistics: mode, chi square, median, percentile, mean, standard deviation, correlation, regression, analysis of variance, geometric mean, harmonic mean, coefficient of variation, logarithms.

A comparison of the most common Function Size Measurement (FSM) Methods

General Information	**IFPUG FPA r4.3**	**NESMA FPA v2.0**	**Mark II FPA r1.3.1**	**COSMIC FSM r3.0.1**
Origin	Created by Allan Albrecht at IBM in 1978 Latest release (January 2010) of the original method	Believed to have been created by NESMA (aka NEFPUG) in mid-1980s Derived from IFPUG	Created by Charles Symons at Nolan Norton in 1984 (put into public domain 1991) Updated method for use with DBMS, structured methods, CASE tools, etc	Created by international consortium of industry subject matter experts and academics from 19 countries in 1997 Updated method for use with OOA/D, layered architectures, Web2.0, lean/agile, etc
Counting Practices Manual	Available to IFPUG members	Available for sale	Available - public domain	Available - public domain

General Information	IFPUG FPA r4.3	NESMA FPA v2.0	Mark II FPA r1.3.1	COSMIC FSM r3.0.1
Counting Practices Manual - languages available	English & some other language versions available to members	Dutch-language version English-language version	English-language version	9 language versions: Arabic, Chinese, Dutch, English, French, German, Italian, Japanese, Spanish
Used by	Public & private sector organisations, large & small, both customers & vendors, around the world Mostly MIS users Stable user base – international	Public & private sector organisations, large & small, both customers & vendors, primarily in The Netherlands Mostly MIS users Declining user base – mostly The Netherlands	Originally HM Government's preferred method for sizing & estimating software. Now used by a few public sector customers & their vendors Declining user base – mostly United Kingdom	Public & private sector organisations, large & small, both customers & vendors, around the world MIS and Engineering users Growing user base – international
Terminology used	Founded in the 1970s	Founded in the 1970s	Uses structured methods terminology	Compatible with OOA/D, & software eng. principles
Availability	Available only to members of IFPUG (but easy to join organisation)	Public domain -– download from NESMA	Public domain – download from UKSMA	Public domain – download from COSMIC
Design Authority (independent of vendors)	International Function Point Users Group (IFPUG) www.ifpug.org	Netherlands Software Metrics Association (NESMA) www.nesma.nl	United Kingdom Software Metrics Association (UKSMA) www.uksma.co.uk	COmmon Software Measurement International Consortium (COSMIC) www.cosmicon.com

Common Features

1. Compliance

All four methods comply with the international standard for Functional Size Measurement Methods – ISO14143:

IFPUG FPA r4.3	NESMA FPA v2.0	Mark II FPA r1.3.1	COSMIC FSM r3.0.1
ISO/IEC 20926:2003 ISO Standard applies only to unadjusted FP	ISO/IEC 24570:2005 ISO Standard applies only to unadjusted FP	ISO/IEC 20968:2002 Recommended method for HM Government (UK)	ISO/IEC 19761:2003/2010 BCS Technology Award Winner in 2006 Recognised as a National Standard in Spain & Japan

2. Certification

All four methods operate certification schemes for training measurement staff:

IFPUG FPA r4.3	NESMA FPA v2.0	Mark II FPA r1.3.1	COSMIC FSM r3.0.1
Yes Certified Function Point Specialist (CFPS)	Uses IFPUG CFPS	Yes Certified Function Point Analyst (CFPA)	Yes COSMIC Practitioner Certification

Benchmarking Data

All four methods are supported by the International Software Benchmarking Standards Group (ISBSG). There are differences, however, in the size of the comparative data pool:

IFPUG FPA r4.3	NESMA FPA v2.0	Mark II FPA r1.3.1	COSMIC FSM r3.0.1
Large data set compiled over many years – the utility of antique data is questionable	Large Comparisons use IFPUG data	Small Some native data; can be compared to IFPUG data if care is taken	Moderate and growing Data since 1997; ISBSG benchmark released 2009; can be compared to older data if care is taken

All four methods share the following characteristics:

- Oriented toward user-required functionality
- Helps verify consistency & completeness of user-required functionality
- Analyses can be used as basis for construction of tests independent of code & test activities
- Measures functional size of dynamic (behavioural) aspects of system (expressed as e.g. use cases, conversational dialogues, user stories, epics & themes, etc)
- Measures development of new requirements
- Measures adaptive maintenance (enhancements)
- Designed for MIS systems - flat & indexed files, batch systems, OLTP systems
- Can be used to measure Functional User Requirements before design, code & test
- Can be used to measure Functional User Requirements after design, code & test
- Can be used to (re)estimate during product life-cycle
- Size can be used as input into top-down software cost models such as COCOMO.II.2000, SLIM, SEER, Price-S, etc
- Can be used to construct product burndown charts, calculate takt time, #sprints, etc
- Independent of product non-functional requirements
- Independent of project constraints
- Independent of developer experience
- Independent of process, project management & development methods
- Early estimates of functional size can be made based on incomplete knowledge of Functional User Requirements – enabling consistent use of one size scale for estimating & measurement throughout project:

IFPUG FPA r4.3	NESMA FPA v2.0	Mark II FPA r1.3.1	COSMIC FSM r3.0.1
Can produce early estimates using various methods: e.g. Fast Eddy, File-Based Approach, Transaction-Based approach	Can produce early estimates using various methods: e.g. Fast Eddy, File-Based Approach, Transaction-Based approach	Can produce early estimates using various methods: e.g. Data Model Approach (CRUDL), Transaction-Based approach	Can produce early estimates using various methods: Event-Based Approach, Object-Based Approach, Story-Based Approach

NONE of the four methods deliver the following characteristics:

- Measures corrective maintenance (fixes)
- Measures perfective maintenance (refactoring for improved performance)
- Measures algorithmic complexity
- Measures reuse of code

Differences between the four main methods:

Characteristic	**IFPUG FPA r4.3**	**NESMA FPA v2.0**	**Mark II FPA r1.3.1**	**COSMIC FSM r3.0.1**
Measures functional size of static (data storage) aspects of a system (expressed as files, tables, entity types, classes, etc)	Yes	Yes	Regarded as 'double accounting' only information processing measured	Regarded as 'double accounting' only information processing measured
Compatible with modern methods of requirements analysis	Partially (1975/85s concepts) requires data model	Partially (1980/85s concepts) requires data model	Yes (1980/95s concepts) requires data model	Yes (1995/2010s concepts) incl. incremental
Designed for MIS systems - Relational DBMS	No But mapping rules have been developed	No But mapping rules have been developed	Yes	Yes
Designed to be applicable to real-time and/or embedded systems	No MIS concepts only	No MIS concepts only	No terminology can be re-interpreted for real-time	Yes one common model applicable across MIS, real-time & embedded systems
Can be used to measure complex, layered architectures	No Rules assume monolithic system – infrastructure & middleware is 'invisible'	No Rules assume monolithic system – infrastructure & middleware is 'invisible'	Yes Limited – can recognise 3-tier architecture	Yes Designed to recognise 'layered architectures' – measures all functional requirements allocated to software systems
Scale type: Nominal – distinguishes members of sets, unordered Ordinal – relationship between sets, unequal intervals Interval – comparisons, equal intervals, arbitrary zero Ratio – comparisons, equal intervals, a natural zero ref: ISO/IEC CD 15939.	'Nominal/Ordinal' Scale Unequal intervals between Low & Average, and between Average & High	'Nominal/Ordinal' Scale Unequal intervals between Low & Average, and between Average & High	'Ordinal/Interval' Scale Weights derived so that 1 MkII fp = 1 IFPUG fp approximately comparing functional processes	Ratio Scale Empirical data suggests 1 cfp = 1 IFPUG fp approximately comparing functional processes

Characteristic	**IFPUG FPA r4.3**	**NESMA FPA v2.0**	**Mark II FPA r1.3.1**	**COSMIC FSM r3.0.1**
Permissible arithmetic & statistical operations	Categories assigned relative weights: Data can be 'ranked', but 'quantifying' differences between values is difficult due to 'cut off' (Low is c. half of High) – ratios are problematic	Categories assigned relative weights: Data can be 'ranked', but 'quantifying' differences between values is difficult due to 'cut off' (Low is c. half of High) – ratios are problematic	Ordered, synthetic scale with a natural zero: Data can be ranked; differences & ratios between values can be quantified within limits but are problematic due to the use of weights	Ordered, constant scale with a natural zero: Data can be ranked; differences between values can be quantified; ratios make sense (i.e. 20 is twice the size of 10, and 2000cfp is twice 1000cfp).
Accounts for information processing by:	Sizing static data and dynamic behaviour	Sizing static data and dynamic behaviour	Sizing dynamic behaviour, the use of data	Sizing dynamic behaviour, the use of data
Models the functional user requirements as:	File Types and Elementary Process (= Input-Process-Output)	File Types and Elementary Process (= Input-Process-Output)	Logical Transactions (= Input-Process-Output)	Functional Processes (= Input-Process-Output)
Equivalent of stimulus/response message pair (i.e. a 'thread of control with some input, related processing, and some output)	Elementary Process either: External Input (EI), External Output (EO) or External Query (EQ) depending on 'primary intent'	Elementary Process either: External Input (EI), External Output (EO) or External Query (EQ) depending on 'primary intent'	Logical Transaction (LT) All stimulus/response message pairs regarded at LT irrespective of 'primary purpose'	Functional Process (FP) All stimulus/response message pairs regarded at FP irrespective of 'primary purpose'
Rules for measuring size	Different rules apply depending on elementary process type	Different rules apply depending on elementary process type	Same rules apply to all logical transactions	Same rules apply to all functional processes
Base Functional Component(s)	Internal Logical File External Interface File External Input External Output External Query	Internal Logical File External Interface File External Input External Output External Query	Input Data Element Entity Reference Output Data Element	Data Movement (either: Entry, eXit, Read, or Write depending on direction of movement)
Contributors to functional size	Per File Type: #static Data Element Types & #Record Element Types Per Transaction Type: #dynamic Data Element Types & #File Type References	Per File Type: #static Data Element Types & #Record Element Types Per Transaction Type: #dynamic Data Element Types & #File Type References	Per Logical Transaction: #Input Data Elements #Entity References #Output Data Elements	Per Functional Process: #Data Movements i.e. the movement (Entry, eXit, Read or Write) of one Data Group
Unit of measure	Different weights assigned to 5 function types depending on their relative 'complexity' Unit = 1 fp (IFPUG)	Different weights assigned to 5 function types depending on their relative 'complexity' Unit = 1 fp (NESMA)	Weights assigned to the 'minimum size logical transaction' add to 2.5 to establish comparability between MkII and IFPUG Unit = 1 fp (MkII)	1 Data Movement = 1 COSMIC Function Point Unit = 1 cfp

Characteristic	IFPUG FPA r4.3	NESMA FPA v2.0	Mark II FPA r1.3.1	COSMIC FSM r3.0.1
Sensitivity to small changes to requirements	Low (only detects changes at boundaries between Low, Average, High categories)	Low (only detects changes at boundaries between Low, Average, High categories)	High (detects changes of single data element types and single entity references)	Moderate (detects changes to single data-groups)
Integrity of measures (how well do the measures reflect the thing measured?)	Artificial limits (weights, thresholds, uneven intervals) limit size of function types measured. Integrity is limited.	Artificial limits (weights, thresholds, uneven intervals) limit size of function types measured. Integrity is limited.	No artificial limits imposed on size of functional process. Integrity is good.	No artificial limits imposed on size of functional process. Integrity is excellent.
Sensitivity to variation in functional size of dynamic model of system i.e. functional processes	Stepped: minimum step 3fp maximum step 7fp	Stepped: minimum step 3fp maximum step 7fp	Stepped: minimum step either 0.26, 0.58 or 1.66 maximum step infinity	Accommodates size variation from zero to infinity in steps of 1 cfp
Sensitivity to variation in functional size of static model of system i.e. data stores	Stepped: minimum step 5 fp maximum step 15 fp	Stepped: minimum step 5 fp maximum step 15 fp	Data stores are considered to deliver functionality only when the data is referenced in transactions	Data stores are considered to deliver functionality only when the data is used in functional processes
Smallest feasible functional process	3 fp	3 fp	2.5 fp	2 cfp
Smallest feasible enhancement	3 fp	3 fp	0.26 fp	1 cfp

The Art of Mocking

Gil Zilberfeld, Product Manager Typemock. Dror Helper, Technical Lead, Better Place
www.typemock.com, http://www.gilzilberfeld.com, http://blog.drorhelper.com

One of the challenges developers face when writing unit tests is how to handle external dependencies. In order to run a test, you may need a connection to a fully populated database, or some remote server; or perhaps there is a need to instantiate a complex class created by someone else. All these dependencies hinder the ability to write unit tests. When such dependencies need a complex setup for the automated test to run, the end result is fragile tests that break, even if the code under test works perfectly. This article will cover the subject of mocks (also known as test doubles, stubs and fakes, amongst other names) and creating manual mocks vs. using a full-fledged mocking framework.

But first - What are unit tests?

Unit tests are short, quick, automated tests that make sure a specific part of your program works. They are performed by testing a specific functionality of a method or class which has a clear pass/fail condition. By writing unit tests, developers can make sure their code works, before passing it to QA for further testing. Let’s look at a C# class with a method we’d like to test.

```
public class UserService
{
     public bool CheckPassword(string userName, string password)
     {
          ...
     }
}
```

We can then write a test that checks for a valid user and password, the method it returns *true*:

```
[Test]
public void CheckPassword_ValidUserAndPassword_ReturnTrue()
{
    UserService classUnderTest = new UserService();
    bool result = classUnderTest.CheckPassword("user", "pass");
    Assert.IsTrue(result);
}
```

By writing several unit tests, we can test for various inputs to CheckPassword, we get expected results, and make sure it performs according to specifications.

Good unit tests run quickly and are isolated from other tests. Both these traits are difficult to achieve out of the box.

Writing unit tests: The obstacles

After a few days of writing tests, every developer hits a wall. The problem is that not all code is created equal and not all methods are simple to test. A common case: What happens when the method-under-test requires access to a database, to check if a user entry exists? Yes, we can set up a database before running the test however it would cost us precious time in which the developer is waiting for his ~1000 tests to run instead of writing code. Would you set up a database specific for every test and then clean it up after the test runs? We wouldn’t.

Sometimes, the code-under-test works perfectly but then you need to set up the environment that the tests depend on. This dependency makes your tests very brittle and they could fail. If tests *sometimes* fail, depending on the environment, you won’t trust them, which defeats the point. The key to solving these issues lies in the usage of mocking. The mocking mechanism replaces the production code behavior (like the database calls), with code that we can control.

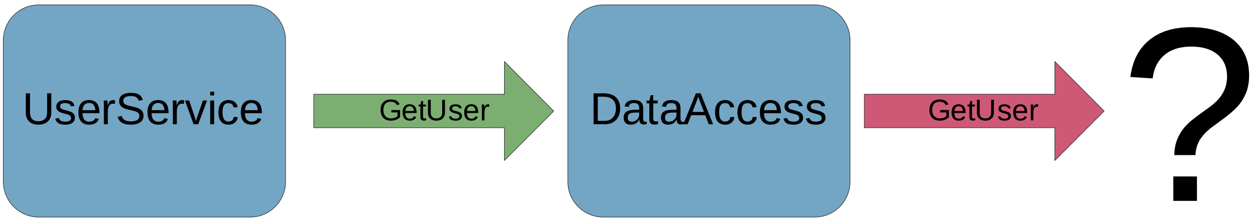
Hand-Rolled Mocking

The first mocking experience usually starts out with hand-rolled mocks. These are classes that are coded by the developer and can replace the production objects for the purpose of testing.
Creating your very first mock is simple - all you need is a class that looks like the class you're replacing, which can be done using inheritance (in Object Oriented languages).

Here is the previous example- now with the *CheckPassword* method fully implemented:

```
public class UserService
{
    private IDataAccess _dataAccess;

    public UserService(IDataAccess dataAccess)
    {
        _dataAccess = dataAccess;
    }

    public bool CheckPassword(string userName, string password)
    {
        User user = _dataAccess.GetUser(userName);

        if (user != null)
        {
            if (user.VerifyPassword(password))
            {
                return true;
            }
        }

        return false;
    }
}
```

Our production code uses external data (Database) in order to store the application's users. Because deploying and populating a database would increase the test's run time and might fail some other solution is in order.



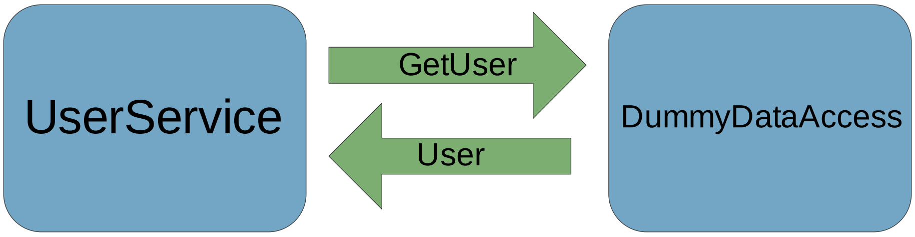
In order to test the *CheckPassword* method we need to provide an object that would implement *IDataAccess*, but that we control. Doing so is quite simple: all we need to do is to create a new class that will implement the desired interface, only whilst the real *DataAccess* class' implementation goes to the database, ours just returns a value we supply:

```
public class DummyDataAccess : IDataAccess
{
    private User _returnedUser;

    public DummyDataAccess(User user)
    {
        _returnedUser = user;
    }

    public User GetUser(string userName)
    {
        return _returnedUser;
    }
}
```

The class we've created does just one thing - it returns a *User* when called, instead of calling the database. Using this *DummyDataAccess* we're now able to alter our test to make it pass:

```
[Test]
public void CheckPassword_ValidUserAndPassword_ReturnTrue()
{
    User userForTest = new User("user", "pass");

    IDataAccess          fakeDataAccess          =          new
DummyDataAccess(userForTest);

    UserService          classUnderTest          =          new
UserService(fakeDataAccess);

    bool result = classUnderTest.CheckPassword("user", "pass");

    Assert.IsTrue(result);
}
```

First we create a *User* and a *DummyDataAccess* object that would return that *User*. Then we create a real *UserService* (the class we want to test), and supply it with the *DummyDataAccess.* We then call the method under test, which eventually uses our fake implementation to return the supplied *User* data.



At first this might look like cheating - it seems that the test does not use the production code fully, and therefore does not really test the system properly. But remember: we're not interested in the working of the data access in this particular unit test; all we want to test is the business logic of the *UserService* class.

We should also have additional tests; these ones test that we read and write data correctly into our database. These are called integration tests

Using the *DummyDataAccess* class achieves three goals:

1. Our test does not need external dependencies (i.e. a database) to run.
2. Our test will execute faster because we do not perform an actual query.

So what is all this mocking about?

While you can spend the whole day categorizing and naming different sorts, we’ll just mention the classic mocks and stubs.

- A Stub is an object that is used to replace a real component without calling any of the real component functionality.
- A Mock objects is a stub that is also used as an observer point for the test. Using a Mock object, a test can verify that a specific method was called, and can use this information as a pass/fail criterion.

You'll see that people give their own definitions using the same words. Because of the loaded definitions of the two, from this point on we'll call both "fake objects" (or "fakes" for short). While the difference exists, it becomes less apparent in modern mocking frameworks, and just not worth the fuss.

Why not stop at hand-rolled fakes?

At first, using manually written fake objects seems like a good idea. All software developers know how to write code, and implementing an interface or deriving a new class is a no-brainer.
The problem is that the simple fake class created yesterday, becomes a maintenance nightmare today. Here are a couple of reasons why:

1. **Adding new methods to an existing interface**
   lets go back to the example from the beginning of this article. What happens when we add a new method to the *IDataAccess* interface? We now need to also implement the new method in the fake object (usually we'll have more than one, so we'll need to implement in the other fakes too). As the interface grows, the fake object is forced to add more and more methods that are not really needed for a particular test just so the code will compile. That's usually necessary work, with almost no value.

2. **Adding new functionality to a base class**
   one way around the method limitation is to create a real class and derive the fake object from it, only faking the methods needed for the tests to pass. Sometimes it can prove risky, though.

   The problem is that once derived, our fake objects have fields and methods that perform real actions and could cause problems in our tests. A recent example we encountered shows why: A hand rolled fake was inherited from a production class: it had an internal object opening a TCP connection upon creation. This caused very strange failures in my unit tests, until we were able to track it down. In this case, we wasted time because of the way we created the fake object.

3. **Adding new functionality to our fake object**
   as the number of tests increases, we'll be adding more functionality to our fake object. For some tests method *X* returns null, while for other tests it returns a specific object. As the needs of the tests grow and become distinct, our fake object adds more and more functionality until it becomes so complicated that it may need unit testing of its own.

All of these problems require us to look for a more robust, industry grade solution - namely a mocking framework.

Mocking Frameworks

A mocking framework (or isolation framework) is a 3rd party library, which is a time saver. In fact, comparing the saving in code lines between using a mocking framework and writing hand rolled mocks, for the same code, can go up to 90%! Instead of creating our fake objects by hand, we can use the framework to create them, with a few API calls. Each mocking framework has a set of APIs for creating and using fake objects, without the user needing to maintain irrelevant details of the specific test - in other words, if a fake is created for a specific class, when that class adds a new method nothing needs to change in the test. One final remark: a mocking framework is just like any other piece of code and does not "care" which unit testing framework is used to write the test it's in.

Mocking framework types

Different frameworks work in different ways - some create a fake object at run-time, others generate the needed code during compilation, and yet others use method interception to catch calls to real objects and replace these with calls to a fake object. Obviously the framework’s technology dictates its functionality.

For example: if a specific framework works by creating new objects at run-time using inheritance, then that framework cannot fake static methods and objects that cannot be derived. It's important to understand the differences between frameworks, prior to committing to one. Once you build a large amount of tests, replacing a mocking framework can be expensive.

What can a mocking framework do for me?

Mocking frameworks perform three main functions:

1. Create fake objects
2. Set behavior on fake objects
3. Verify methods were called

In the next examples, we've used Typemock Isolator, a .NET mocking framework. Wikipedia has an extensive list of mocking frameworks sorted by programming language at http://en.wikipedia.org/wiki/List_of_mock_object_frameworks.

1. Creating fake objects

Once a fake object is created, how does it behave? It might return a fake object, or throw an exception when called, depending on the mocking framework used. Most of the time, to use the new fake object in a test, additional code is required to set its behaviors.

2. Setting behavior on fake objects

After creating a fake object, its behavior needs to be configured. The following example takes the code from the beginning of this article and uses a mocking framework to create a set behavior of a fake *DataAccess* object:

```
[Test]
public void CheckPassword_ValidUserAndPassword_ReturnTrue()
{
    User userForTest = new User("user", "pass");

    IDataAccess fakeDA = Isolate.Fake.Instance<IDataAccess>();
    Isolate.WhenCalled(() =>  .......................... ...
     fakeDA.GetUser(string.Empty)).WillReturn(userForTest)

    UserService          classUnderTest          =          new
UserService(fakeDataAccess);

    bool result = classUnderTest.CheckPassword("user", "pass");

    Assert.IsTrue(result);
}
```

The test is very similar to the test we had before. This time, we don't need to create and maintain a class to fake the *DataAccess* class. We can create tests without worrying about any maintenance penalty.

3. Verify methods were called

Test frameworks can test state value: fields, properties, variables. Comparing the actual value to the expected ones are the pass/fail criteria. Mocking frameworks add another way to test - checking whether methods were called, in which order, how many times and with which arguments. For example, let’s say that a new test is required for adding a new user: if that user does not exist than call *IDataAccess.AddNewUser* method. Note that the *AddUser* method doesn’t have a return value we can check on. We need to know if the call actually happened.

```
[Test]
public void AddUser_UserDoesNotExist_AddNewUser()
{
    IDataAccess fakeDA = Isolate.Fake.Instance<IDataAccess>();
    Isolate.WhenCalled(() =>  .........................
     fakeDA.GetUser(string.Empty)).WillReturn(null);

    UserService classUnderTest = new UserService(fakeDA);

    classUnderTest.AddUser("user", "pass");

    Isolate.Verify.WasCalledWithAnyArguments(()                =>
fakeDA.AddUser(null));
}
```

The test is very similar to the previous test, except for two points:

1. *null* is returned when *GetUser* is invoked to specify that the user does not exist
2. At the end of the test, instead of an assertion on a result, the mocking framework is used to verify that a specific method was called.

Some mocking frameworks have additional capabilities, other than the basic main three such as the ability to invoke events on the fake object or cause the creation of a specific fake object inside the product code. The three basic capabilities are the core functionality expected from every mocking framework. Additional features should be compared and checked when deciding which mocking framework to use.

Choosing a mocking framework

Changing an existing mocking framework requires updating all existing unit tests and so choosing the right mocking framework is important. There are no clear rules to how one should decide which framework to use but there are some key factors that need to be taken into consideration:

- **Functionality**

  Not all mocking framework are created equal. Some frameworks might offer additional capabilities that other do - for example if a mocking framework uses inheritance to create fake objects it cannot fake static and non-virtual methods, while a framework that employs instrumentation and/or method interception can. Look for the features that are not strictly "mocking" features such as event and method invocation - those can help write better unit tests faster. Almost every mocking framework has a site or white paper detailing its features.

Compare several frameworks features side by side and see which scores higher.

- **API and ease of use**

  Just like any other 3rd party library it is important that a mocking framework should be easy to use. Try several frameworks to see which makes most sense to you. A simple, discoverable and readable API (application programming interface) is important because it would directly affect how readable the tests are that use it. Easy to use as well as easy to read are definitely factors worthy of consideration.

- **Price**

  Some mocking frameworks are free while others cost money. Usually (but not always) there is a good reason that a certain framework is not given for free. Users might be entitled to premium support or training that would help getting up to speed with the new tool. The paid version might have features that the free frameworks do not have. Check the licensing scheme, keep in mind that you might need to purchase a new license for each developer in the team as well as build servers. At the end of the day it's all about ROI (return on investment) even a pricey tool that saves 50% of the time of each developer is worth the investment.

Best Practices and common pitfalls

When using a mocking solution it's easy to forget that it's merely a tool - and as such can be abused. Just like any other development tool it is up to the developer to learn to use it well.

Here are a couple of aspects you should know how to handle.

1. **Know what to isolate**
   The key advice when using a fake object is to understand what is under test and what is the dependency. The object under test would not usually be faked - so finding the target and scope of the test helps finding out what to fake.

2. **Fake as little as needed**
   Using too many fake objects creates fragile tests - tests that are likely to break when production code changes occur. It is advisable to fake as little as possible. Faking chatty interfaces should be avoided because a small change in the order of calls would break your test. Start by writing the test without the fake objects then fake only what you need to make the test pass.

3. **Fake the immediate neighbors**
   Fake the objects directly called by the subject of the test. Unless you want to test the interaction between several classes try to limit the scope of the fake object to those directly affecting the class under test.

4. **Don't misuse *Verify***
   When you have a fake object, the entire world looks like it should be verified. It's easy to fall to the trap of making sure that method "A" calls method "B" - most of the time it does not really matter. Methods are refactored and changed frequently, so *Verify* should only be used where it's the only pass/fail criterion of your test.

5. **One (or two) assertions per test**
   When a test has more than one *assert* it may be testing too many things at the same time. The same principle applies to using *Verify*. Testing if three different methods where called should be done in three separate tests. If the first *verify* that fails throws an exception, we don't have any clear knowledge about the success or failure of the other two. This keeps us further from fixing the problem.

Summary

Unit testing is a major component of every agile methodology. The early feedback you receive from your tests help you feel confident that you didn't introduce new bugs, and that you gave QA actual working code.

This article was written to give you an understanding into the world and art of Mocking.
Mocking frameworks are essential tools for writing unit tests. In fact, without a tool like this, you're bound to fail in your effort - either you won't have unit tests that give early feedback, or no tests at all.

This is why deciding on a mocking framework or some other similar solution is as important as deciding the unit testing framework used. Once you pick a framework, master it. It helps make your unit testing experience easy and successful.

Restructure101 for Java

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

Restructure101 is a tool that allows refactoring a software architecture in a sandbox environment without risks, removing unnecessary complexity and dependencies. The end result is an action list, exported to the IDE, that will guide developers in the improvement of the actual codebase.

Web Site: http://www.headwaysoftware.com/
Version Tested: Restructure101 for Java 1.0 Build 1235, tested on Windows XP, period from May to June 2011
System Requirements: Restructure101 runs on Windows, OS/X and Linux/Unix platforms
License & Pricing: Commercial, but free for open source projects, price $900 per user
Support: Email support and free upgrades are priced at $250/year

Software Architecture Problems

Even when it has been well conceived in the beginning, maintaining the quality of software architecture is not obvious. Trade-offs are made to achieve short-term schedule objectives and team turnover often leaves people working in the maintenance phase with little knowledge of the initial architectural principles. This gradual loss of the architecture's quality is represented by "technical debt" or "architecture decay". Although a team should not necessarily try to achieve a completely "clean" status of their architecture during project development or software maintenance, the deterioration of the architecture's quality over time could lead to a situation where the evolution of the system is no longer cost-effective (too much “technical debt”). Developers don't control anymore the consequences of their modifications, as the relationships between the system components are so complex that it is difficult to evaluate the side effects of any change to the codebase. A situation often made worse by the absence of post-change control using existing regression tests.

Software development teams usually try to tackle this problem with code analysis tools. Headway Software itself supplies also a tool called Structure101 that fits in this category. These tools act as a diagnostic tool on the code base, trying to find issues like code dependencies, code duplication or dangerous code.

Restructure101 provides a complementary vision to code analysis. Based on a diagnostic of the architectural quality of an application, it allows improving the software architecture using a heuristic process of exploratory changes to the structure of the application in the Restructure101 sandbox. Restructure101 doesn't work on the coding quality of your Java application, but rather on the packaging aspect of your system through the binaries. It is naturally possible to visualise the code behind the package.

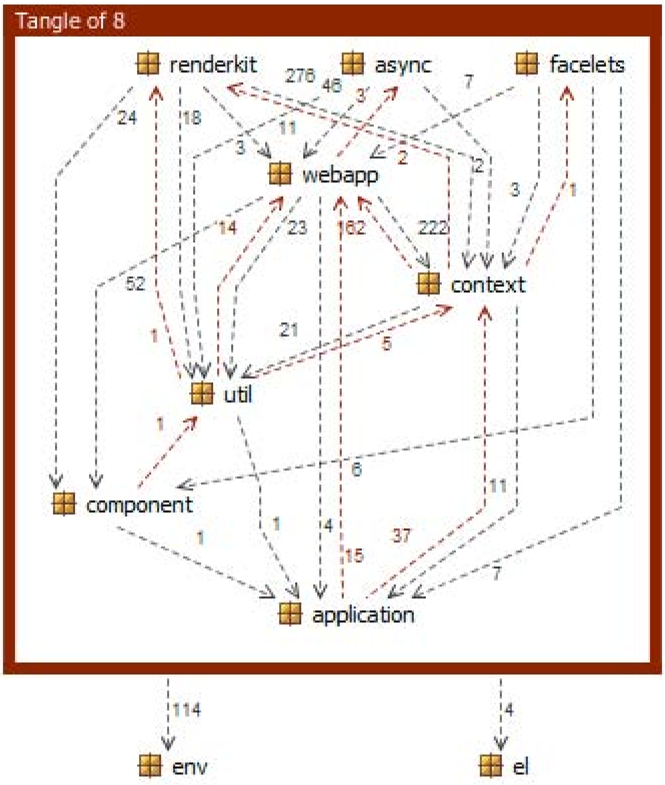
The Levelized Structure Map (LSM)

The main tool used to visualise the application architecture is the Levelized Structure Map (LSM). In this model, items (Maven POMs, jars, packages and classes) are levelized into rows, or levels, so that every item depends on at least one item on the level immediately below it. Items in the same row do not depend on each other (unless there are cyclic dependencies), and items on the lowest level do not depend on any other items at the same scope. This arrangement conveys a lot of dependency information so that most of the item-to-item dependency can be hidden without loss of context.

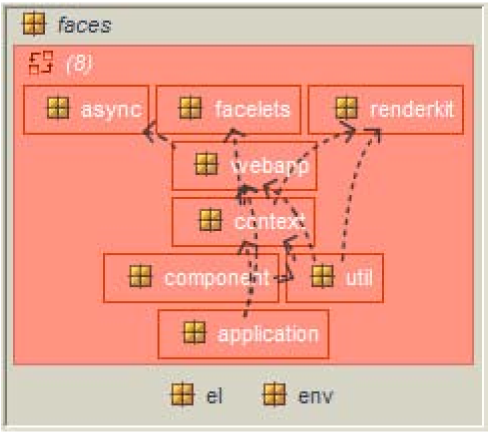
The manipulations of the LSM in a Restructure101 sandbox are aimed at solving the following type of problems (in the order of importance as emphasised in the tool):

* Tangling or cyclical code dependency

Code dependency occurs when a portion of your code depends on code located elsewhere to achieve its functionality. A high level of dependency makes it more difficult to change your application, as the impact of changes has to be assessed in all the dependent code. A circular dependency occurs when two modules are mutually dependent, often indirectly though a chain of other modules. The Levelized Structure Map offers additional value in the analysis of dependencies as it should make it easier to compare the dependency level of a module with its conceptual location in the architecture: lower level modules in the data access layer should have less dependencies than high level modules in the user interface layer.



Dependency graph of a package tangle



LSM of the same package tangle showing “feedback dependencies”

* Fat

Fat is too much complexity at any point of design breakout, for example the number of sub-packages in a higher level package, too many classes included in a package or too many methods in a class. Design items naturally accumulate an increasing number of contained items as a code base grows. When fat exceeds a complexity threshold, it is a good idea to perform a step of incremental design to split it into smaller abstractions that are easier to understand and extend.

*Split packages

Packages whose child dependency graph (of subpackages and/or classes) contain disconnected groups of items having low cohesion. Especially where the separate groups of items have divergent dependencies, it may make sense to split the package to better represent the true abstractions it contains.

* Mixed packages

These are packages that contain both sub-packages and classes, with dependencies between them. Architects sometimes deem this mixing of abstraction levels undesirable and moving the classes into new or existing sub-packages as indicated by their dependency level will give a consistent level of abstraction within the parent package.

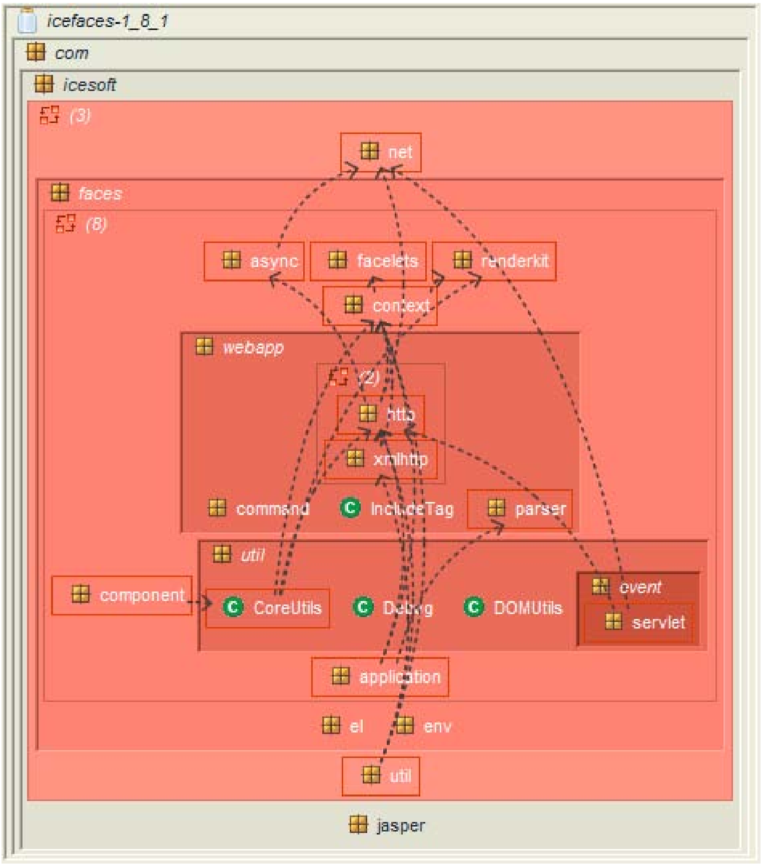
This product review is focused on the Java version of Restructure101, but the same tool allows a similar approach on .NET, PHP or Actionscript systems, and C/C++ through Headway's partners: Coverity and Programming Research.

Using Restructure101

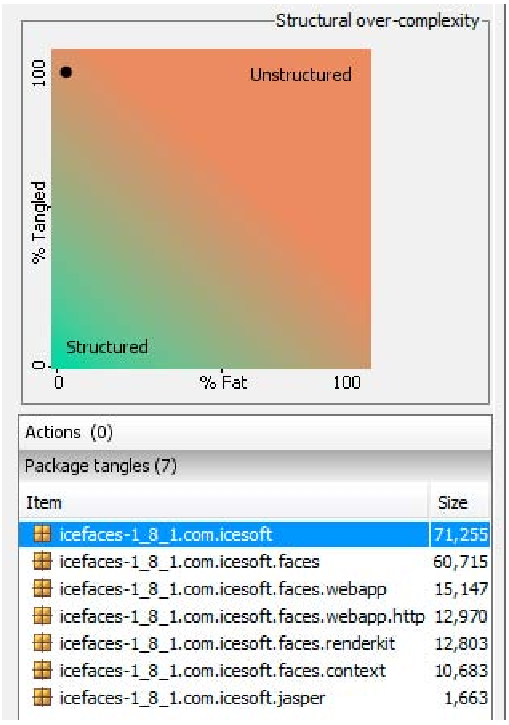
To examine a new application, you create a sandbox indicating where the jar files are located. A number of options allow you to choose how your want to see the packages interactions, the level of detail, if you want to include external libraries and if you want to be able to visualise the source code in the case you want to access it. These properties can be changed after importing the model. Besides local projects, you can also work with architecture stores in a repository, either on a local file system or on the web. You can create an indefinite number of sandboxes in which to experiment with refactoring your codebase. Sandboxes can be duplicated if you want to save a particular state of your refactoring activity.

After parsing the packages, Restructure101 will give you a first diagnostic on the level of tangling and fat of the application and present a high level LSM. The left side of the tool shows you the diagnostic of your recent actions, a number of navigation lists including tangled packages and fat items. Within the LSM, you can double-click on an item to open or close it. You can perform a number of refactoring actions for example dragging a class to another package, which immediately shows the effect that action will have on the LSM (which is re-levelized) and records the action in your left column.

The right column contains configuration switches that allow mainly modifying the display of dependencies on the LSM. For example sets of cyclically dependent items can be grouped into a single higher-level component. In your LSM diagram you can manipulate the existing packages and classes of your application (rename, delete, move) and add new items. The following example shows the result of applying such actions to a highly tangled code base. The following is an LSM of the initial state, expanded to show several levels of nested cycles. The upward arrows are “feedback” dependencies that are causing the packages to be cyclically dependent and preventing the chart from being fully levelized.

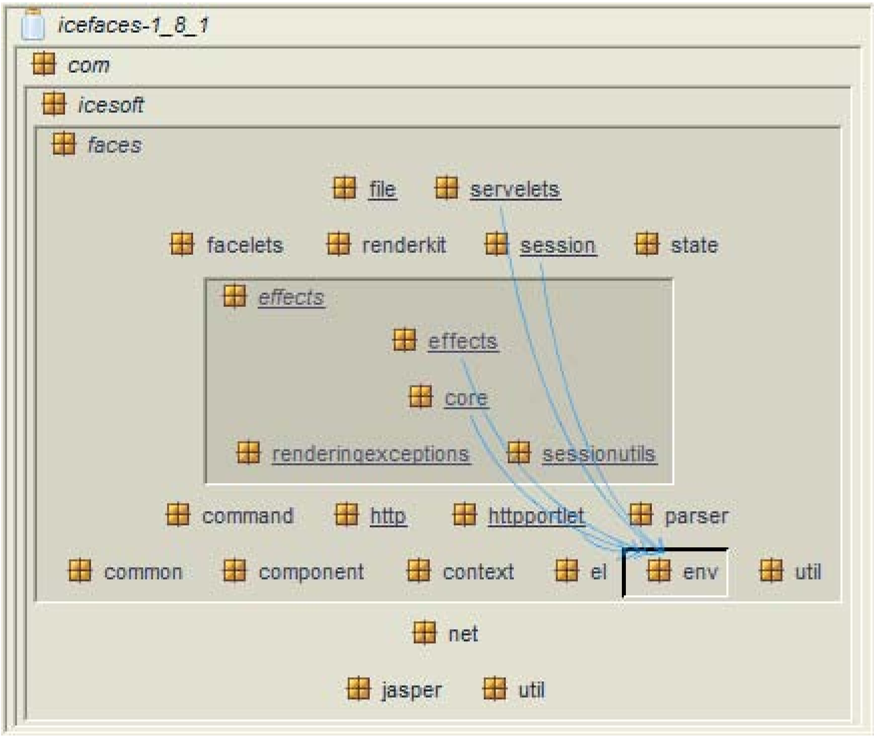


The information panel on the left of the UI shows that almost all of the code in this code-base is involved in package tangles, and the tangled packages are listed. There are no fat items in this example.

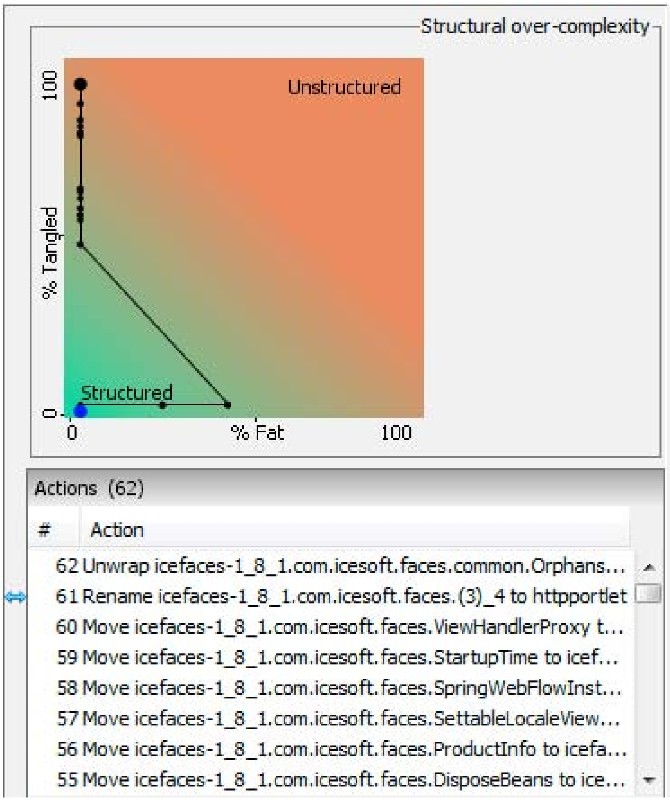


Item	Size
icefaces-1_8_1.com.icesoft	71,255
icefaces-1_8_1.com.icesoft.faces	60,715
icefaces-1_8_1.com.icesoft.faces.webapp	15,147
icefaces-1_8_1.com.icesoft.faces.webapp.http	12,970
icefaces-1_8_1.com.icesoft.faces.renderkit	12,803
icefaces-1_8_1.com.icesoft.faces.context	10,683
icefaces-1_8_1.com.icesoft.jasper	1,663

After a couple of hours, a sequence of manipulations that removed all package cycles was discovered and the LSM looked like this:



There are no package feedback dependencies and therefore no red zones: all the dependencies flow downward. This makes the architecture much easier to understand and the impact of changes more predictable. You can see on the chart above the possible impact of making a change to classes in the com.icesoft.faces.env package. The impact on the structural over-complexity of the code-base with each refactoring action is shown on the chart below:



62 Unwrap icefaces-1_8_1.com.icesoft.faces.common.Orphans...
61 Rename icefaces-1_8_1.com.icesoft.faces.(3)_4 to httpportlet
60 Move icefaces-1_8_1.com.icesoft.faces.ViewHandlerProxy t...
59 Move icefaces-1_8_1.com.icesoft.faces.StartupTime to icef...
58 Move icefaces-1_8_1.com.icesoft.faces.SpringWebFlowInst...
57 Move icefaces-1_8_1.com.icesoft.faces.SettableLocaleView...
56 Move icefaces-1_8_1.com.icesoft.faces.ProductInfo to icefa...
55 Move icefaces-1_8_1.com.icesoft.faces.DisposeBeans to ice...

The temporary increase in the Fat dimension happened when the existing package structure in one region was “flattened”. This removed the associated tangle but resulted in a large number of classes in a single package. When the classes were packaged up without re-introducing cycles, the over-complexity was reduced to the origin.

The 62 refactoring actions retained the existing package structure as much as possible with totally new packaging being created where the existing packages seemed hopelessly tangled. This strategy would help reduce the impact on the developers’ understanding of the code base. If this were not important, creating a new package structure bottom-up from the classes, using the Restructure101 automatic grouping feature would be an alternative and possibly easier strategy.

Other features

If Restructure101 automatically constructs the initial hierarchies from the binaries, transformations give you almost unlimited control over the structure of your model. You can specify a number of expressions that modify the fully qualified names of the classes in your project, in effect moving them to locations other than their physical locations in the code-base. You specify the transformation of a number of items that matches a certain pattern into a new output, which allows you to merge or split easily physical packages that constitute a single logical package (e.g. API and implementation packages).

The final action list can be exported to Eclipse and IntelliJ so that developers can actually implement it and improve the architecture of the application.

Conclusion

Behind its simplicity of manipulation, Restructure101 for Java offers a powerful mechanism to explore and modify the architecture of a Java application without any risk. Restructure101 makes it possible to salvage a decayed architecture with minimal effort. Once it is well structured, the architecture can be pro-actively defined and enforced using the complementary Structure101 product. It is not easy to quantify the benefits, but experience suggests that a well-structured code base with a defined architecture is a lot more efficient to extend and maintain than the proverbial Big Ball of Mud.

Liquibase

Nathan Voxland, nathan.voxland @ liquibase.com

You would never develop code without version control, why do you develop your database without it?

Web Site: http://liquibase.org
Version Tested: 2.0.1
License & Pricing: Apache 2.0 License, Free
Support: Community forum (http://forum.liquibase.org) or commercial support and training from http://liquibase.com

Liquibase is an open source library for tracking, managing and applying database changes that can be used for any database with a JDBC driver. It is built on a simple premise: all database changes are stored in a human readable yet trackable form and checked into source control.

At its simplest form, Liquibase is a tool that reads a list of database changes from a changelog file. The file starts out empty when you begin your application, but as you make changes to your application that require corresponding changes in the underlying database, you append a description of the changes to the changelog file. The description of the changes can be standard SQL commands such as “add column” or more complex descriptions such as “introduce lookup table”. The changelog file is often XML-based, but does not have to be. Once a new change is added to the changelog file, Liquibase will read the file, detect the new change, and apply it to the database. When you commit your code changes to your version control system, you commit the changelog file containing the database “version” alongside it.

What you end up with is a mixture of a database version control system and database refactoring tool. As your database evolves over the course of a project, Liquibase ensures that the database you deploy to production has the same schema that the application code expects and has been tested against.

Compared to other database change tracking tools, Liquibase has three defining differences:

1. Understand how the database changed, not just what changed

A common approach to database change management is:

1. Take a database snapshot at the beginning of a project
2. Allow developers to make the changes they need as the develop their code
3. Compare the final database to the original database.
4. From this comparison, auto-generate the SQL to add, remove and modify tables, columns, views, etc.

A prime example of this is Hibernate’s hbm2ddl library, which allows you to create your Java to database mapping, then determine the SQL needed to make the database match the schema the code expects. While this approach works for many source code version control systems, there is an inherent problem applying it to databases: the **way** you get from the start to the final state is as important as getting there. A simple example is a change that renames the person.fname column to person.first_name. A database comparison would see that there is no longer a person.fname column and there is now a person.first_name column.

The SQL it would generate would be:

```
alter table person drop column fname;
alter table person add column first_name varchar(255);
```

While the resulting database schema is the correct form, when you apply the above SQL to your production database, you will find that all your records have null first_names. Because the tool was not smart enough to understand **how** the schema changed, you have lost data.

There are many similar examples of the importance of knowing how the schema changed. In order to protect the database, Liquibase does not use automatic comparison but instead relies on a changelog being built up manually one changeset set at a time. It may sound tedious at first glance, but as you will see, the process is streamlined and in practice it fits well with standard development techniques.

2. Handle Multiple Developers and Multiple Branches/Merges

There are database versioning tools rely on the manual creation of SQL or SQL-like changesets, but many use a simplistic tracking system that does not scale to multiple developers or code branches. In particular, they are built around a concept of a linear database “version” which starts at version 1. After a change is added, the version is incremented to 2, then 3, etc. When an existing database is set to be updated, the current version is determined and all the changesets after that version are applied.

This works well for projects where only one person adds changesets and/or there are never any branches, but it quickly breaks down when separate developers attempt to add a “version 4” to the database concurrently. This can be managed with good team communication, but falls apart completely when versions 4 to 10 were added in a feature branch that needs to be integrated back into the master branch that already has versions 4 to 20 added.

Liquibase does away with this issue by using a unique identification scheme for each changeset that is designed to guarantee uniqueness across developers and branches while still being easy to manually manage. As you will see in the below examples, each Liquibase changeset contains two attributes: an “id” and an “author”. These two attributes along with the name and path of the file make up the changeset identifier Liquibase uses to determine if it has been ran against a given database. At update time, each changeset is compared against the list of applied changesets and it is executed if and only if it has not been run before. Since the comparison is done for each changeset instead of being based on a single “version”, any new changesets brought into the changelog file--whether from a different developer or from a different branch--will be correctly executed.

3. Higher Level Refactorings

Finally, Liquibase supports not just standard create/alter/update SQL statements, but higher level database “refactorings” such as “split column” and “introduce lookup table” which allow complex database changes to be described and managed easily. This not only makes the initial changelog creation easier, but improves readability and traceability. Furthermore, you gain the ability to support updating and managing the same schema across multiple database vendors using the same changelog file.

Liquibase also supports a powerful extension model which allows you to define arbitrarily complex changes as well as add functionality to the built-in refactorings. This allows you to wrap update patterns that are common to you into an easy to use and manage package.

Using Liquibase

To see how Liquibase works, start with a blank database and a blank changelog file in a directory containing the unzipped Liquibase binary from http://liquibase.org/download. If you name your blank changelog file db.changelog.xml, your directory structure will look like:

- lib
    - JDBC Driver jar file(s) for your database
- db.changelog.xml
- liquibase
- liquibase.bat
- liquibase.jar

Liquibase is built on the Java platform, and therefore requires the Java runtime environment (1.5+) to be installed in your system and the corresponding JDBC driver jar files in the lib folder. Contact your database vendor for information on available drivers and jdbc url structure. If everything is installed correctly, running:

```
>liquibase.bat --version
```

Will output:

```
Liquibase Version 2.0.1
```

Open the blank db.changelog.xml file paste in

```
<databaseChangeLog
xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-
2.0.xsd">
</databaseChangeLog>
```

You can now update your database using the db.changelog.xml file by running

```
>liquibase.bat  --changeLogFile=db.changelog.xml  --username=USER
--password=PWD --url=jdbc:mysql://localhost/liquibase update
```

with correct values for username, password, and url.

You will see the output:

```
INFO 4/14/11 12:08 AM:liquibase: Successfully acquired change
log lock
INFO 4/14/11 12:08 AM:liquibase: Creating database history table
with name: `DATABASECHANGELOG`
INFO 4/14/11 12:08 AM:liquibase: Reading from
`DATABASECHANGELOG`
INFO 4/14/11 12:08 AM:liquibase: Successfully released change
log lock
Liquibase Update Successful
```

If you look at your database, you have two new tables: “databasechangelog” and “databasechangeloglock”. These two tables are used by Liquibase to track changes and execution. There is still nothing else in your database because there is nothing in your db.changelog.xml file.

If you replace insert

```
    <changeSet id="1" author="nvoxland">
        <createTable tableName="person">
            <column name="id" type="int" autoIncrement="true">
            <constraints primaryKey="true" nullable="false"/>
            </column>
            <column name="firstname" type="varchar(50)"/>
            <column name="lastname" type="varchar(50)">
                <constraints nullable="false"/>
            </column>
        </createTable>
    </changeSet>
```

into the <databaseChangeLog> root element and run the Liquibase.bat update command list above again, you will see the output:

```
INFO 4/14/11 12:14 AM:liquibase: Successfully acquired change
log lock
INFO 4/14/11 12:14 AM:liquibase: Reading from
`DATABASECHANGELOG`
INFO 4/14/11 12:14 AM:liquibase: ChangeSet
db.changelog.xml::1::nvoxland ran successfully in 43ms
INFO 4/14/11 12:14 AM:liquibase: Successfully released change
log lock
Liquibase Update Successful
```

and the database now contains a person table.

Building your ChangeLog

The above pattern of:

1. Append new changeSet to databaseChangeLog
2. liquibase.bat update

is repeated over and over as you need new changes to your database. The changeSets can include any of the changes listed at http://liquibase.org/manual/home, raw SQL, or any custom changes you create using the extension framework (http://liquibase.org/extensions). For a common development process with multiple developers, the pattern is extended to:

1. Write code
2. Find you need a database change
3. Append new changeSet to db.changelog.xml
4. liquibase.bat update
5. Test code and database
6. Repeat 1-4 as necessary
7. Update local codebase from version control
8. liquibase.bat update to apply changes from other developers
9. Repeat 1-8 as necessary
10. Commit your code and db.changelog.xml to version control
11. When ready, update QA database with db.changelog.xml built up during development

12. When ready, update production database with db.changelog.xml built up during development

Additional ways to run Liquibase

If the command line interface does not fit your needs, Liquibase can be ran on demand via Ant, Grails or Maven, or can be ran automatically as part of application startup using the built in Spring or Servlet Listener support or interacting with a simple Java façade API.

Managing ChangeLogs

The above example uses a single XML file that contains all the changeSets, but as your project grows, you may want to break changelogs into multiple files using the <include> tag. Depending on your needs, you can create changelog files in:

- XML
- Raw SQL (which is treated as a single changeSet)
- Formatted SQL (which allows you to break it up into multiple changeSets)
- DSL-style format
- Create your own format using the extension framework.

For example, the person example above could be stored in db.changelog.sql and written as:

```
--liquibase formatted sql
--changeset nvoxland:1
CREATE TABLE person (
    id int PRIMARY KEY,
    firstname varchar(255),
    lastname varchar(255) NOT NULL
);
```

or, using the grails plugin from http://grails-plugins.github.com/grails-database-migration/, as

```
databaseChangeLog = {
     changeSet(author: 'nvoxland', id: '1') {

     createTable(tableName: 'person') {

     column(autoIncrement: 'true', name: 'id', type: 'BIGINT') {

     constraints(nullable: 'false', primaryKey: 'true')
      ..................................................}

     column(name: 'firstname', type: 'VARCHAR(255)')

     column(name: 'lastname', type: 'VARCHAR(255)') {
      ..................................................
     constraints(nullable: 'false')
      ..................................................}
      ..................................................}
     }
}
```

Additional Liquibase Features

Beyond tracking and applying changes to a database, Liquibase supports many other powerful features including:

- *Rollback Support:* If you want to undo an update, *liquibase.bat rollback* allows you to roll back changeSets based on number of changeSets, to a given date, or to a given tag stored in the database
- *Update/Rollback SQL Output:* rather than executing updates or rollbacks directly against the database, you can generate the SQL that would be ran for inspection and/or manual execution.
- *Future Rollback Output:* Before you apply an update to a database, you can generate the SQL you would need to run in order to bring the database back to the state it is in now for inspection.
- *ChangeLog and ChangeSet preconditions:* Preconditions can be added to the changeLog or individual changeSets to check the state of the database before attempting to execute them
- *DBDoc:* You are able to generate Javadoc style documentation for your current schema and its history. See http://www.liquibase.org/dbdoc/index.html for example output
- *ChangeSet Contexts:* ChangeSets can be assigned “contexts” in which to run. Contexts are selected at runtime and can be used to have changeSets that only run in test instances or other unique circumstances
- *ChangeSet checksums:* When a changeSet is executed, Liquibase stores a checksum and can fail or alter execution if it detects a change between the original definition of a changeSet when it was run and the current definition.
- *Diff Support:* Although Liquibase is built to use database comparisons for change management, there is support for it in Liquibase which is helpful in many cases such as performing sanity checks between databases.

More Information

For more information, visit http://liquibase.org. There you will find documentation, videos, downloads, and more.

Maven

Evgeny Goldin, http://evgeny-goldin.com/

Maven is a build tool traditionally used in Java and Java EE projects to compile source files, execute unit tests and assemble distribution artifacts. While Maven specializes in Java projects and artifacts, such as .ear and .war applications, it is not limited to those environments and can be equally used for Groovy and Scala projects, which seem to be popular alternatives to Java these days.

Web Site: http://maven.apache.org/
Version Tested: 2.2.1, 3.0.3 on Windows 7 / Server 2008 / Ubuntu 10.04, Java 1.6.0_25 x86
License & Pricing: Open Source (Apache license), Free
Documentation:

- Web Site: http://maven.apache.org/guides/index.html
- Maven plugins: http://maven.apache.org/plugins/, http://mojo.codehaus.org/plugins.html
- Sonatype Maven Books: http://www.sonatype.com/Support/Books

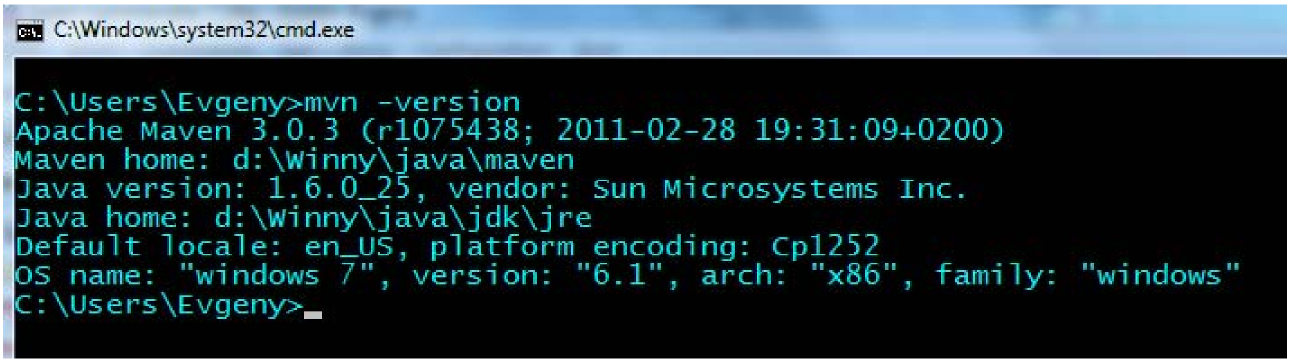
Support:

- Mailing list: http://maven.40175.n5.nabble.com/Maven-Users-f40176.html
- Issue tracker: http://jira.codehaus.org/browse/MNG
- Sonatype: http://www.sonatype.com/Support/

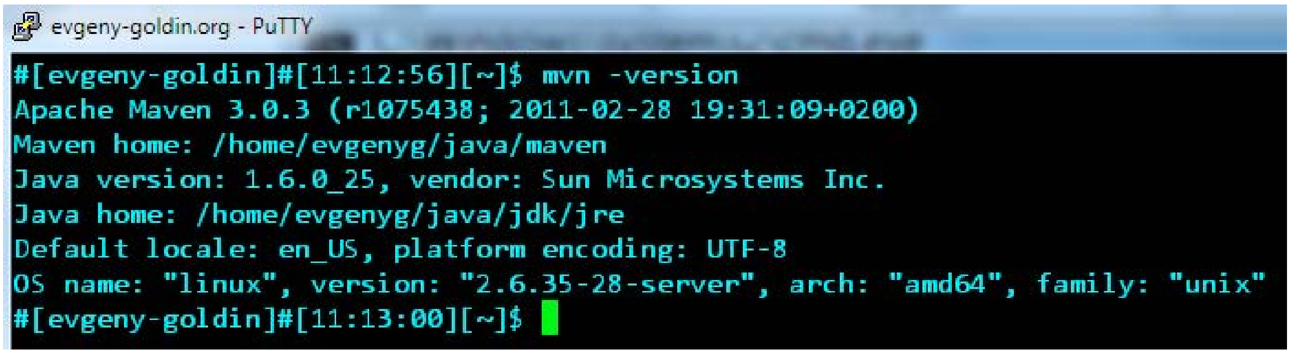
Installation

Installing Maven is simple; all you need to do is download the corresponding archive of Maven 3 from http://maven.apache.org/download.html, unpack it and add the **“bin”** folder to your system **PATH**. Maven is a multi-platform tool. It runs on every operation system on which Java is installed. Please make sure that you have the **JAVA_HOME** environment variable defined properly and that you can execute the **“java -version”** command.

After adding Maven’s **“bin”** folder to the **PATH** environment variable, running **“mvn –version”** will make sure the installation is successful:



```
C:\Users\Evgeny>mvn -version
Apache Maven 3.0.3 (r1075438; 2011-02-28 19:31:09+0200)
Maven home: d:\Winny\java\maven
Java version: 1.6.0_25, vendor: Sun Microsystems Inc.
Java home: d:\Winny\java\jdk\jre
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 7", version: "6.1", arch: "x86", family: "windows"
C:\Users\Evgeny>_
```


evgeny-goldin.org - PuTTY

```
#[evgeny-goldin]#[11:12:56][~]$ mvn -version
Apache Maven 3.0.3 (r1075438; 2011-02-28 19:31:09+0200)
Maven home: /home/evgenyg/java/maven
Java version: 1.6.0_25, vendor: Sun Microsystems Inc.
Java home: /home/evgenyg/java/jdk/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "2.6.35-28-server", arch: "amd64", family: "unix"
#[evgeny-goldin]#[11:13:00][~]$
```

Two additional system properties can now be defined: **`M2_HOME`** specifies Maven's installation folder and **`MAVEN_OPTS`** specifies JVM options that will be used for every build execution; it can be set to **“`-Xmx512m -XX:MaxPermSize=256m`”** if you plan to run large builds which take a long time to run.

You can find further installation details in Chapter 2 of “Maven: The Complete Reference”, located at http://www.sonatype.com/books/mvnref-book/reference/installation.html and an online manual at http://maven.apache.org/download.html#Installation.

The POM file

Maven operates on “pom.xml” files, commonly referred as POMs for “Project Object Model”. Those XML files declaratively represent your project, its structure and any additional configurations or dependencies applied to the build lifecycle.
A sample POM file is provided below:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns              = http://maven.apache.org/POM/4.0.0
         xmlns:xsi          = http://www.w3.org/2001/XMLSchema-instance
         xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0
                               http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.company</groupId>
    <artifactId>web-app</artifactId>
    <packaging>war</packaging>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.16</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.5</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</project>
```

This POM file provides a description of a web project which is compiled, tested and assembled into a .war archive when **“`mvn clean install`”** is invoked. There's no need to specify any of those build steps since Maven relies heavily on standards and conventions. As long as your project structure follows these conventions, very little needs to be added. This POM file also shows how we can declaratively specify project dependencies or any other project-specific settings. Relying on conventions and declarative definitions is what makes Maven different from older build tools such as Ant, in which all lifecycle steps need to be specified and managed explicitly. Detailed POM references are available online at: http://maven.apache.org/pom.html and http://www.sonatype.com/books/mvnref-book/reference/pom-relationships.html.

Lifecycle, Usage and Plugins

For every module built, Maven activates a build lifecycle composed of phases which build engineers are very familiar with: `“clean”`, `“compile”`, `“test”`, `“package”`, and `“install”`. Many other fine-grained build phases are also available. The full list can be found at http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html and http://www.sonatype.com/books/mvnref-book/reference/lifecycle.html.

To execute a Maven build you need to specify phases to be invoked by typing `“mvn <`*`phase`*`>”`, such as `“mvn clean install”` or `“mvn test”`. When a phase like `“install”` is specified, all phases preceding it in the lifecycle are also implicitly executed. That includes the `“compile”`, `“test”` and `“package”` steps. Note that `“clean”` is not part of a standard lifecycle and needs to be run explicitly.

Each phase execution is handled by one of the corresponding Maven plugins. Maven itself deals with reading and analyzing POM files, dependencies resolutions and lifecycle management but all actual tasks of compiling the sources, running unit tests and assembling the distribution archive are delegated to one of its core plugins. Their list is provided at http://maven.apache.org/plugins/ and each plugin can be configured separately to match the project requirements. By configuring a plugin you can change any of its default settings, for example specifying additional compilation flags or testing options. You can also attach Maven’s plugin execution to one of the lifecycle phases. This allows changing the behavior of core plugins that are always invoked as part of the standard lifecycle phases, such as `“maven-compiler-plugin”`, but it is also possible to add any other Maven plugin to the build lifecycle.

One sample configuration of a compiler plugin is provided below:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>2.3.2</version>
    <configuration>
        <source>1.6</source>
        <target>1.6</target>
        <encoding>UTF-8</encoding>
        <compilerArgument>-Xlint:all</compilerArgument>
        <showDeprecation>true</showDeprecation>
        <showWarnings>true</showWarnings>
    </configuration>
</plugin>
```

This configuration targets Java sources compiled to run on JDK 1.6. In addition, it enables all compiler warnings and deprecation notifications for developers to notice possible code issues as early as possible.

Maven and Groovy

Even though the list of Maven plugins is an impressive one, you may need more plugins at times. Some builds may have additional requirements not covered by existing plugins. In this case there are a number of ways in which Maven can be extended:

- By executing Ant code as part of the build.
- By executing Groovy code as part of the build.
- By executing custom Maven plugins, developed for specific needs.

From these 3 options, executing custom Groovy code can be a good option to try first since developing Maven plugins may require time and resources not readily available while adding Ant code makes POM files significantly harder to read.

For running Groovy code as part of a Maven build you may use GMaven plugin, as shown below:

```
<plugin>
    <groupId>org.codehaus.gmaven</groupId>
    <artifactId>gmaven-plugin</artifactId>
    <version>1.3</version>
    <executions>
        <execution>
            <id>run-groovy</id>
            <goals><goal>execute</goal></goals>
            <phase>package</phase>
            <configuration>
                <providerSelection>1.7</providerSelection>
                <source>
                    println “Build time is [${ new Date()}]”
                </source>
            </configuration>
        </execution>
    </executions>
</plugin>
```

This Groovy code snippet is executed after the build artifact is created, in the “package” lifecycle phase. Groovy can perform any additional build actions, interact with POM data and the artifacts created or invoke any other third-party libraries you may need. Further information about GMaven plugin is available on its Web site at http://docs.codehaus.org/display/GMAVEN/.

Summary

Maven is a very popular and stable build tool, widely used by many developers and organizations these days. It provides a standard build lifecycle, many sensible conventions regarding project organization and declarative ways to define any project-specific settings. Maven provides a great number of core plugins which handle all traditional build tasks like compiling sources, running unit tests and archiving distribution assemblies. If you still find yourself missing some functionality, then extending Maven with Groovy, Ant or custom plugins is also possible. Sonatype is the company standing behind Maven and it provides valuable Maven information, free e-books, support, and training. Further information is available online at http://www.sonatype.com/.

Automated WebTesting with Selenium RC

Maria Marcano, Nearsoft, http://mariangemarcano.blogspot.com/

Selenium RC (or Selenium 1) is a popular tool for writing automated tests of web applications. You can develop automated tests in the programming language of your choice such as c#, java, python, php, perl and ruby as well as running those tests on different combination of browsers such as chrome, firefox or IE.

Web Site: http://seleniumhq.org
Version: Selenium RC 1.0.3
License & Pricing: All Selenium projects are licensed under the Apache 2.0 License.
Support: There are many places where you can find support; here is a list from Selenium's site http://seleniumhq.org/support/

Overview

Selenium project gathers a set of tools for writing automated tests of websites: Selenium RC (remote control), Selenium IDE, Selenium Grid and Selenium 2 (on beta) which is the next version of Selenium RC.

These tools emerged from a javascript library that was used to drive interactions on a webpage on multiple browsers called Selenium Core.

Selenium RC is a client/server based application that allows you to control web browsers using the following components

- **Selenium Server**: Uses Selenium core and browser's built-in JavaScript interpreter to process selenese commands (such as click, type) and report back results.
- **Selenium Client Libraries**: Are the API's for the programming languages to communicate with Selenium server.

Running Selenium Server

Download Selenium RC from http://Seleniumhq.org/download/, the zip contains Selenium server, a Java jar file (Selenium-server.jar).

Selenium server must be running to be able to execute the tests. You can run it using the following command:

```
C:\>java -jar [SeleniumServerPath]\selenium-server.jar -interactive
```

Hello World Selenium RC

The following example uses c#, but a similar approach can be followed using others client driver libraries to develop tests in java, python, php, perl and ruby.

Using Selenium .Net client driver and Visual Studio 2010 (or 2008 Professional Edition)

1. Create a test project.
2. Add a reference to ThoughtWorks.Selenium.Core.dll on the project (this is found in the Selenium RC zip under Selenium-remote-control-1.0.3\Selenium-dotnet-client-driver-1.0.1 directory).

3. Create a test class with the following structure:

```
using Selenium;

namespace TestProject1
{
    [TestClass]
    public class SeleniumPageTest
    {
        private ISelenium Selenium;

        [TestInitialize()]
        public void MyTestInitialize()
        {
            Selenium =
                  new DefaultSelenium("localhost", 4444, "*firefox",
                  "http://seleniumhq.org/");
            Selenium.Start();
        }

        [TestCleanup()]
        public void MyTestCleanup()
        {
            Selenium.Stop();
        }

        [TestMethod]
        public void CheckProjectsLink()
        {
            Selenium.Open("http://Seleniumhq.org/");
            Selenium.Click("link=Projects");
            Selenium.WaitForPageToLoad("3000");
            Assert.IsTrue(Selenium.IsTextPresent("Selenium IDE"));
        }
    }
}
```

Run this test in Visual Studio like you do with a regular unit test.

MyTestInitialize

This method initializes Selenium by creating an instance of DefaultSelenium (Default implementation of Selenium interface) specifying the following parameters:

- Host name on which the Selenium server is running (localhost).
- The port on which Selenium server is listening (when we started Selenium server by default it listens on port 4444).
- The command string used to launch the browser, e.g. "*firefox", "*iexplore" or "c:\\program files\\internet explorer\\iexplore.exe",
- The starting URL, Selenium starts the browser pointing at the Selenium resources on this URL (http://seleniumhq.org/).

The start method lunches the browser and begins a new Selenium testing session.

MyTestCleanup

The stop method ends the Selenium testing session and kills the browser.

CheckProjectsLink

This is a simple test that opens http://seleniumhq.org page, clicks on the “Projects” link, waits for the page to load (with a timeout of 3 seconds) and asserts that the text "Selenium IDE" is present on the page.

Benefits of having Selenium automated tests

Selenium automated tests have provided the following benefits on my projects:

- Execute regression tests easily and have quick feedback about the application’s status.
- Run the same set of tests with different browsers, we’ve caught functional errors present in one browser and not in the others.
- Run the same set of tests on different code branches (and browsers) on daily basis in a continuous integration environment.

When writing Selenium tests remember

- Tests that access elements by id run faster than accessing elements using xpath expressions.
- Use tools like xpather and firebug to quickly locate elements.
- Selenium IDE is handy to record Selenium commands while executing interactions on the UI.
- Run your Selenium tests automatically in a controlled environment using continuous integration tools which involves automated build, deploy and testing process.
- You can run multiple tests at the same time running Selenium server on different ports.

Unstructured Tests

One common approach is to start developing automated tests having basic structure: test method, test initialize and cleanup, as shown in the SeleniumPageTest class.

This may work well at the beginning, but projects ends up with tests like the following:

```
    [TestMethod]
    public void RegisterUserTest()
    {
        // Starting Register User Test
      Selenium.Open("www.mysite.com");
      Selenium.Click("lnkRegister");
      Selenium.WaitForPageToLoad("3000");
      Selenium.Click("btnRegister");
        Assert.IsTrue(Selenium.IsTextPresent("Please enter required
fields"));
      Selenium.Type("id_password", "123456");
      Selenium.Type("id_password_2", "654123");
      Selenium.Click("btnRegister");
      Assert.IsTrue(Selenium.IsTextPresent("Passwords must match"));
      Selenium.Type("id_email", "mytest@email.com");
      Selenium.Type("id_first_name", "John");
      Selenium.Type("id_last_name", "Doe");
      Selenium.Type("id_password", "xxx#ZZ1");
      Selenium.Type("id_password_2", "xxx#ZZ1");
      Selenium.Click("id_acept_terms");
      Selenium.Click("btnRegister");
      Selenium.WaitForPageToLoad("3000");
      Assert.IsTrue(Selenium.IsTextPresent("Welcome John Doe, logout"));
      Selenium.Click("lnkLogout");
      Selenium.WaitForPageToLoad("3000");
      // RegisterUsTest Completed
    }
```

The above test has the following issues:

- Code duplication and tests have high dependency with the page’s HTML structure. This means that changes in a single page will affect different tests. When the application changes, tests will start breaking and this will be hard to maintain over the time.
- Readability issues: Tests are not easy to read. Is difficult to know what the test is doing.

Page Objects

Page Objects is a pattern that helps structure automated test code to overcome maintainability issues; this is how page objects helps:

Methods on a page object represent the "services" that a page offers (rather than exposing the details and mechanics of the page). For example the services offered by the Inbox page of any web-based email system:

- Compose a new email
- Read a single email

How these are implemented shouldn't matter to the test.

The benefit is that there is only one place in your test suite with knowledge of the structure of the HTML of a particular (part of a) page.

Summary of Page Objects

1. Represent the screens of your web app as a series of objects
2. Do not need to represent an entire page
3. Public methods represent the services that the page offers
4. Try not to expose the internals of the page
5. Generally don't make assertions
6. Methods return other PageObjects
7. Different results for the same action are modeled as different methods
8. Check that the “Test Framework” is on the correct page when we instantiate the PageObject

Benefits achieved by applying page objects

- There is one place having the knowledge of the structure of the pages (the page object)
- Navigation between the pages.

- Changes in a page are in one place (reducing duplication).
- Easy to locate code.
- Less dependency between the test cases and Selenium, since most Selenium code will be located on the page object.
- As the amount of tests increases, the page objects represent a smaller percentage of the overall test code.

Page Objects Tests

This is how unstructured tests will look after applying page object pattern:

```
[TestMethod]
public void TestRegisterUserEmptyFieldsValidation()
{
      var user = new User(); // empty user
      var registrationPage = RegisterUserExpectingErrors(user);
      Assert.IsTrue(registrationPage.IsRequiredFieldsMessagePresent());
}
[TestMethod]
public void TestRegisterUserPasswordMustMatch()
{
      var user = new User()
      {
            Password = "123456", Password2 = "654123"
      };
      var registrationPage = RegisterUserExpectingErrors(user);
      Assert.IsTrue(registrationPage.IsPasswordsMustMatchMessagePresent());
}
[TestMethod]
public void TestRegisterUserSucessfully()
{
      var user = new User()
        {
            Email =  "mytest@email.com",
            FirstName = "John",
            LastName = "Doe",
            Password = "xxx#ZZ1",
            Password2 = "xxx#ZZ1"
        };
      var homePage =  new HomePage(Selenium).SelectRegisterUser();
        var adminPage =   registrationPage.FillUpRegistrationForm(user);
                                  .RegisterUserSucessfully();
      Assert.IsTrue(adminPage.IsWelcomeBackMessagePresent(user.FirstName));
      adminPage.Logout();
}
private RegistrationPage RegisterUserExpectingErrors(User user)
{
      var registrationPage = new HomePage(Selenium).SelectRegisterUser();
      .............................................................
      return registrationPage.FillUpRegistrationForm(user)
                        .RegisterUserExpectingErrors();
}
```

Writing maintainable automated tests

Below are key principles our team follows when writing automated tests:

1. **Readability:** We want tests to be written in a way that even a final user can read them and understand them.
2. **Maintainability:** Writing automated test with c# (or other programming language) and Selenium is equivalent as writing application code, so we should follow coding best practice and OO principles.
3. **Robustness & Flexibility:** Robust tests that won't break with small changes, being able to do changes with reduced impact. Tests should be repeatable: I can run it repeatedly and it will pass or fail the same way each time.
4. **Collaboration & Team Work:** We want our tests structured in a way that allows easy collaboration and reuse between team members.

Summary of other Selenium projects

- **Selenium IDE** is a Firefox add-on that allows you to record and playback actions performed on a webpage. Also you can format recorded tests to port them to Selenium RC (C#, java, perl, php, python, ruby), when doing so modify them with maintainability considerations mentioned on the article. Using this is fine to start, but it quickly becomes faster coding the tests directly.
- **Selenium Grid** is a solution to scale Selenium RC tests, allowing running tests on parallel, different machines and environments.
- **Selenium 2** is the next version of Selenium RC, which is the result of merging WebDriver and Selenium RC. WebDriver is another tool for writing automated tests of websites but was designed to address some Selenium RC limitations like Same Origin Policy. The difference is that WebDriver controls the browser itself using native methods of the browser and operating system.
- Selenium 2 supports the WebDriver API and is backward compatible with Selenium RC, which means you can still run tests developed with this version.

References

Selenium client libraries: http://seleniumhq.org/docs/05_Selenium_rc.html#programming-your-test

Xpather: https://addons.mozilla.org/en-US/firefox/addon/xpather/

Firebug: https://addons.mozilla.org/es-es/firefox/addon/firebug/

Page Objects: http://code.google.com/p/Selenium/wiki/PageObjects

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