// METHODS & TOOLS

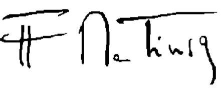
Practical knowledge for the software developer, tester and project manager **ISSN 1661-402X**

Winter 2012 (Volume 20 - number 4) www.methodsandtools.com

Checklists in Software Development: What to Do or What Should Have Been Done?

At the end of the past century, the approaches and methodologies (Information Engineering, RUP, CMMI) that were trying to improve software development processes and projects had a huge documentation on how to do things. Checklists were often used as gateways between project phases and you had to confirm that every goal of the phase has been addressed. Then the Agile Manifesto proposed a new approach preferring "individuals and interactions over processes and tools". This switched the software development process from picking a roadmap in an existing large body of proposed or imposed tasks to building upon a minimal framework, guided by general principles and with only few recommended practices. Imposed checklists had often a bad reputation. You would cross all items, just to move your code or project to the next level, even if you hadn't actually performed the required code inspection for instance. Checklists are however useful in software development and some approaches like lean recommend them to make the knowledge explicit. There is a difference between having checklists that are proactive tools to preserve, share and discuss knowledge and the checklists that are gateway items often responsible for habit sclerosis.

You should consider software development checklists like cooking recipes. When you start cooking, recipes give you a quick access to the knowledge of expert cooks. However, as long as you know only what to do but not why you do it, you might face disappointments if the size of your fish or your oven is not exactly the same than the one used to create the recipe. The more you master the art of cooking, the more you can modify existing recipes to your taste or what you have just found on the market. At the end you might even be able to create your own recipes. If you cook this special meal only once every 6 months or if you need to share it with a friend, then you have to write down the recipe. This is the same for software development good practices. Your checklist will naturally contain what to do, but it will be even better if you write and explain the why, how and why not of its content. If you are reading Methods & Tools, there are chances that you are convinced that you can share and keep knowledge with written material. You can do this inside your organization too. Just start to write one checklist for something that you don't do often or share knowledge with a new colleague. Your good resolution for 2013 could be to put writing checklists in your to-do list.



Inside

Testing Performance of Mobile Apps - Part 2: A Walk on the Server Side ... page 3

Agile Facilitation & Neuroscience: Transforming Information into Action ... page 14

Test Driven Traps ... page 28

Data Management Tools for Embedded Application Development ... page 46

JSHint JavaScript Code Quality Tool ... page 50

No More Mocks in NoSQL Applications with NoSQLUnit ... page 56

STARCANADA Conference - Click on ad to reach advertiser web site

Testing Performance of Mobile Apps - Part 2: A Walk on the Server Side

Alan Trefzger, www.xbosoft.com, blog.xbosoft.com

Introduction

We used to come to work, sit down in front of our computer and check the news, reply to emails, and do online shopping. But now we do all those things on our mobile device while commuting to work (hopefully while not driving). And our mobile Internet usage is projected to pass desktop Internet usage in 2014 [1]. The world has changed but our expectations have not. We expect the online experience to be as fast as our desktop experience, but it isn't, and that is a problem.

For mobile commerce, customers expect a page to load within 4-5 seconds.[2] In Web Performance Today, Nov 2011, a case study showed that "a 1 second delay in load times led to dramatic losses in bounce rate, page views, conversions, and cart size".[3]

Bounce rate	Page views	Conversions	Cart size
-8.3%	-9.4%	-3.5%	-2.1%

Figure 1: Results of 1 second delay in Load Times [4]

The problem is the same for information providers. Google found that a .5-1 second increase in page load time resulted in a 20% decrease in traffic and revenue. [5]

But how to ensure the same high quality user experience as your desktop is a difficult problem. Evaluating and testing the performance of a mobile application is not as straight forward as evaluating and testing the performance of traditional web-based solutions as there are several other variables such as application structure (browser versus native), network used (2G, 3G, 4G, etc.), payload structure, etc.

Performance testing of mobile devices and client applications was covered by Newman Yang of XBOSoft in his "Testing Performance of Mobile Apps – Part 1: How Fast Can Angry Birds Run?" But that is only a part of the performance problem.

Mobile browser-based application performance is usually heavily dependent on network and server application performance. Overall, mobile is much slower than desktop. "The median web page takes more than 11 seconds to load on both iOS and Android devices, compared to about seven seconds on the desktop."[6] Mobile is slower, but it doesn't have to be.

This article will address server performance for a mobile application. Since server issues are very similar for both mobile and full websites, we will look at testing the server which has already been tested and working for a desktop website. Once the server has proven to be stable and performing well for this narrow test, then we can continue onto performance testing for the network.

Differences between Mobile and Desktop usage

Fat vs. Thin data pipe: The mobile web is different than how you use the web on your desktop. When accessing an Internet site, a web page is delivered from a server directly to your desktop through a high speed, broadband data pipe. But for mobile, the data pipe is much thinner.

WebPerformance Load Testing Software & Services - Click on ad to reach advertiser web site

Because your web site should be in better shape than you are.

Training the NYC Marathon live website to be the fastest in the world for three years running.

New York Road Runners is the organization behind the world-famous New York City Marathon, which sends 40,000 runners across the city's five boroughs for 26.2 miles each November. Fans register to follow their favorite runners online, and these individual status pages are updated continuously until the exhausted and exhilarated runners cross the finish line, 20-30 participants per second.

For the past three years Web Performance has load tested the servers that provide live race information. We test and tune the capacity of the Athlete Tracking Website up to a level of 100,000 concurrent users, and the site handles the live race traffic without breaking a sweat.

Many Tower connections: For a mobile connection, not only is the pipe much smaller, but the data has to pass from tower to tower before it gets to your mobile device. This is one of the reasons your mobile user experience can be up to 2 times slower than the desktop experience.

Smaller, less powerful device: Your mobile device has a less powerful CPU, less RAM and a smaller screen resolution. The server needs to take this into account.

Navigation: Your finger is used to navigate so touches have to be translated into clicks.

Data Usage: As data plans charge by the data usage, a different philosophy is needed to speed up performance. Many websites anticipate what you might click on next and preloads those pages. Some websites preload all the next level pages. This can greatly speed up the user experience for the desktop, but for the mobile device, it takes up valuable bandwidth and can be costly.

These differences (and more) means that you have to approach mobile devices differently than when content is being served to the customer.

Proxy servers - Know what is going on behind the scenes

When you test, you need to know what you are testing. Sometimes the data coming back to your mobile device is not coming from the server of the URL that you requested, but from a proxy server that reformats the data and then sends it to the mobile device. Some Nokia devices and Opera mini browsers, both request web pages through proxy servers, which process and compress them before sending them to the client mobile phone, speeding up transfer by two to three times and dramatically reducing by up to 90% the amount of data transferred. Since many data plans charge by the amount downloaded, the savings can be considerable. This pre-processing by the proxy server increases compatibility with web pages not designed for mobile phones.

So when you are testing, you need to know whether or not you are going through a proxy server, like the servers that Opera Mini uses. For if you test through a proxy server, then your results will be dramatically different, and probably better than if you didn’t. Since most users will access your servers through a non-proxy server, (unless you require Nokia devices or Opera browsers) you must make sure you are testing in non-proxy server environments.

First step, a simple comparison test

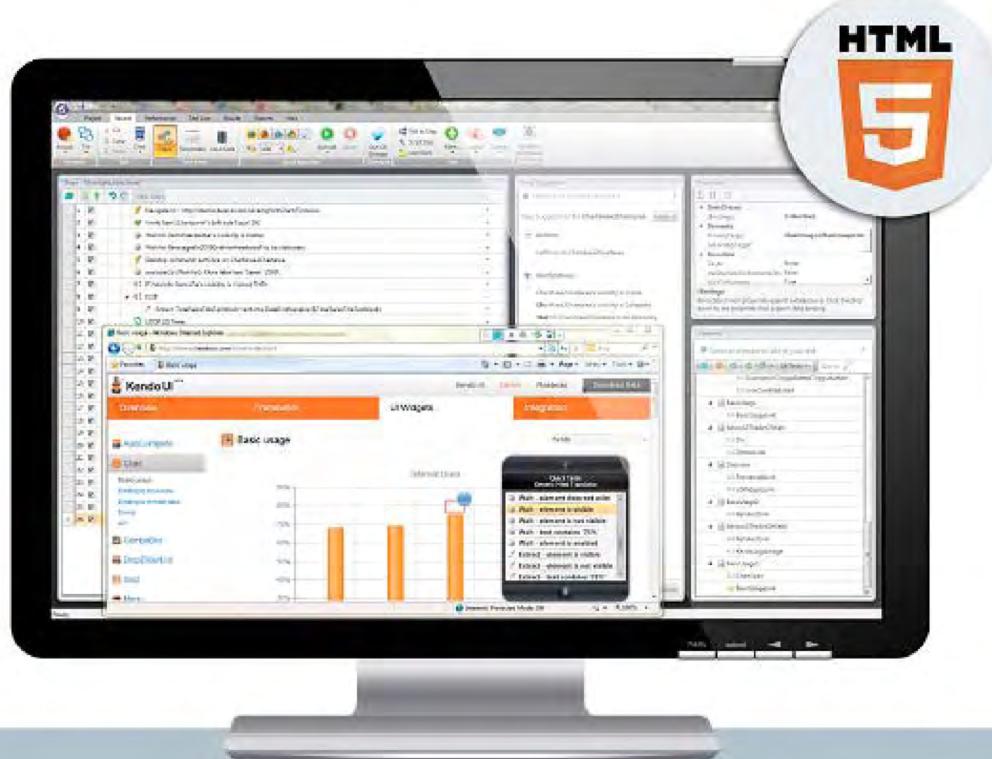
There are three approaches to making your websites accessible to mobile devices. The first and most common approach is to develop a mobile version, a m.site, of your full website designed for the desktop. The second is a mobile app and the third is to make your full website more mobile friendly. In all three cases, the first step is a simple comparison test.

Example: XBOSoft has a full version website, xbosoft.com, and has developed a mobile version, m.xbosoft.com. A performance test was executed to compare the times between the full website on a desktop, the full website on a mobile device and the m.site on a mobile device. IE8 was used for the desktop test. A Wi-Fi network was used for all three to eliminate network issues with 2g or 3g networks. An iOS 5.0 and an Android 2.3 phone were used and their results averaged. Both a first access (no cache) and a repeat access (caching) were used. Six tests were performed on each device and the results averaged.

Telerik Test Studio - Click on ad to reach advertiser web site

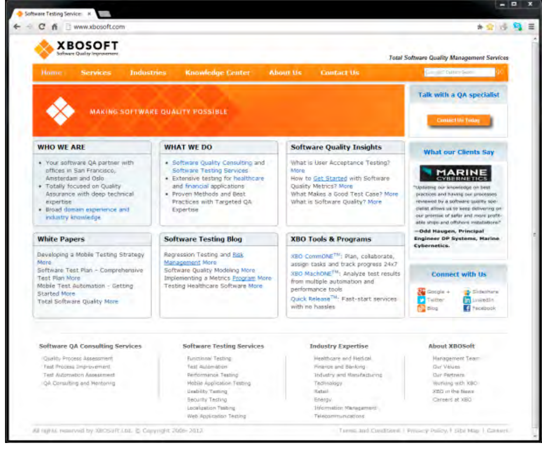
Test Studio

Easily record automated tests for your modern HTML5 apps



Test the reliability of your rich, interactive JavaScript apps with just a few clicks. Benefit from built-in translators for the new HTML5 controls, cross-browser support, JavaScript event handling, and codeless test automation of multimedia elements.

www.telerik.com/test-studio



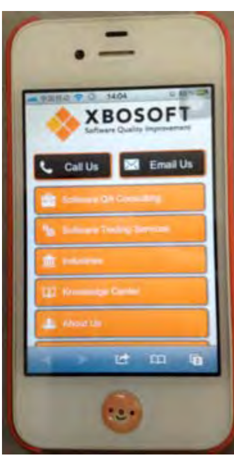


Figure 2-4: xbosoft.com on desktop - xbosoft.com on iPhone 4 - m.xbosoft.com on iPhone 4

Comparison of Load Times (in seconds)	First download	Repeat download
xbosoft.com on desktop	7.5	3.1
xbosoft.com on mobile device	14.8	4.4
m.xbosoft.com on mobile device	9.1	4.1

The development of m.xbosoft.com was considered a success for two reasons:

1. m.xbosoft.com was 40% faster than xbosoft.com on the mobile devices, and
2. m.xbosoft.com was easier to read than xbosoft.com on the mobile devices.

Analysis of Server Performance

You have decided on your approach, now you want to carefully analyze your servers' performance. This next step is critical for all three approaches. There are many good tools in the marketplace, but we would like to start with webpagetest.org.

Example: We will use our previous example of xbosoft.com and m.xbosoft.com and describe how we would go about improving the server performance.
First run a test on xbosoft.com and m.xbosoft.com
Go to webpagetest.org and enter:

- Website url - xboxoft.com and m.xbosoft.com
- Test location (pick one), and
- Browser - iPhone 4 iOS 5.1.

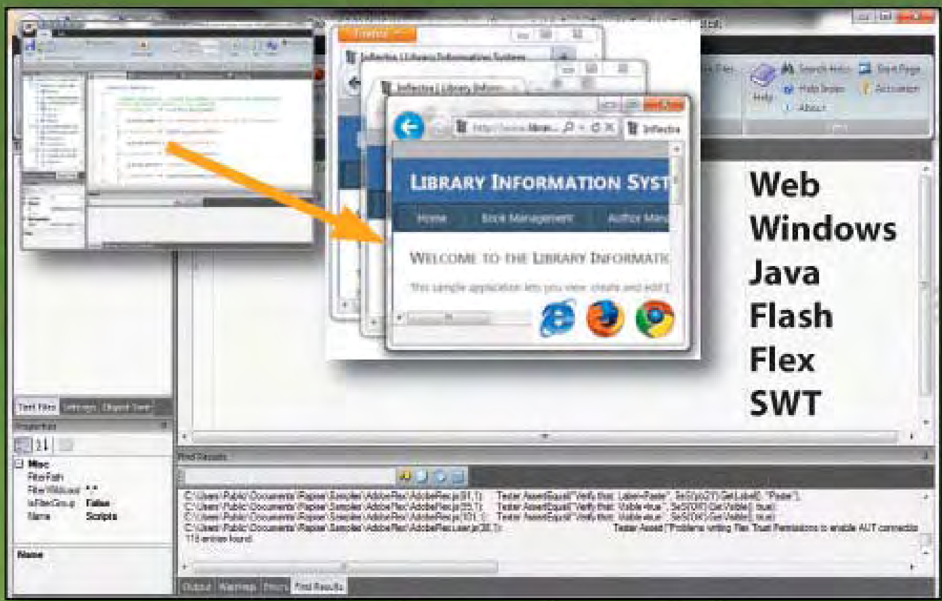
Results of webpagetest.org

Webpagetest.org gives a grade on six areas where significant performance can be made. For this test, significant improvement can be made in 5 of the six areas measured. If performance requirements are not being met, this is a quick way to identify what areas to work on to improve performance.

Rapise Rapid & Flexible Test Automation - Click on ad to reach advertiser web site

Need to Test Your Application on Multiple Environments?

Writing Test Scripts Too Slow?



It's time to try a better way.

Learn more at: inflectra.com/rapise

www.inflectra.com
sales@inflectra.com
+1 202-558-6885

Web Page Performance Test for
xbosoft.com

From: Dulles, VA - **iPhone 4 iOS 5.1**
Fri Nov 23 2012 17:28:39 GMT+0800 (China Standard Time)

Figure 5: Summary Scorecard for xbosoft.com on iPhone 4 iOS 5.1

- First Byte Time – time for receiving first byte for the page.
- Keep Alive Enabled – connection socket is kept open so that many objects can use it.
- Compress Transfer – compress object types "JavaScript" or "text".
- Compress Images – compress object types "image".
- Cache Static Content – controls expiration of objects in cache.
- CDN Detected – Content Delivery Network, Is a distributed system of servers being used?

Next, the Load Time is given for viewing the page the first time (no caching) and for repeat viewing (with caching). For this test, the caching of the objects dramatically speeded up and improved the performance.

				Document Complete			Fully Loaded		
	Load Time	First Byte	Start Render	Time	Requests	Bytes In	Time	Requests	Bytes In
First View	12.654s	0.000s	4.074s	12.654s	49	842 KB	12.655s	49	842 KB
Repeat View	3.200s	0.000s	1.223s	3.200s	7	51 KB	3.201s	7	51 KB

Figure 6: Load Time Summary for First View download and Repeat View download

The Connection view shows each TCP socket and the requests that are retrieved over them. If there are too many connections, then make sure the 'keep alive' setting is enabled. You can also reduce the number of connections by combining js/css files, images, etc… In this example, connections were kept to a minimum, increasing performance.

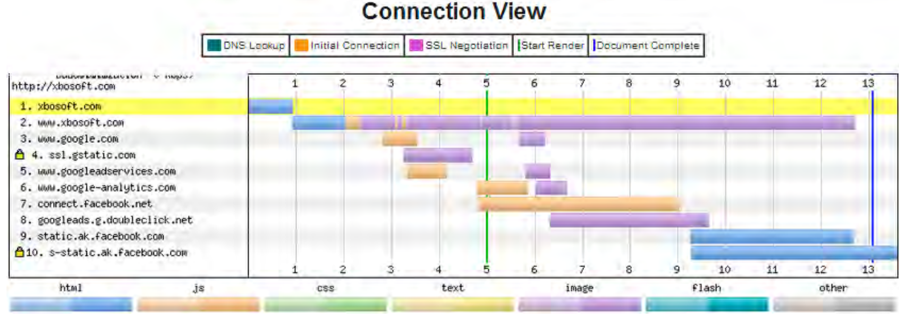


Figure 7: Connection View of each TCP socket opened

The Waterfall view gives the developer a detail look at each component, image, js/css file, etc… and how long each takes to load. A component that is taking too long can quickly be pinpointed and fixed (or eliminated).

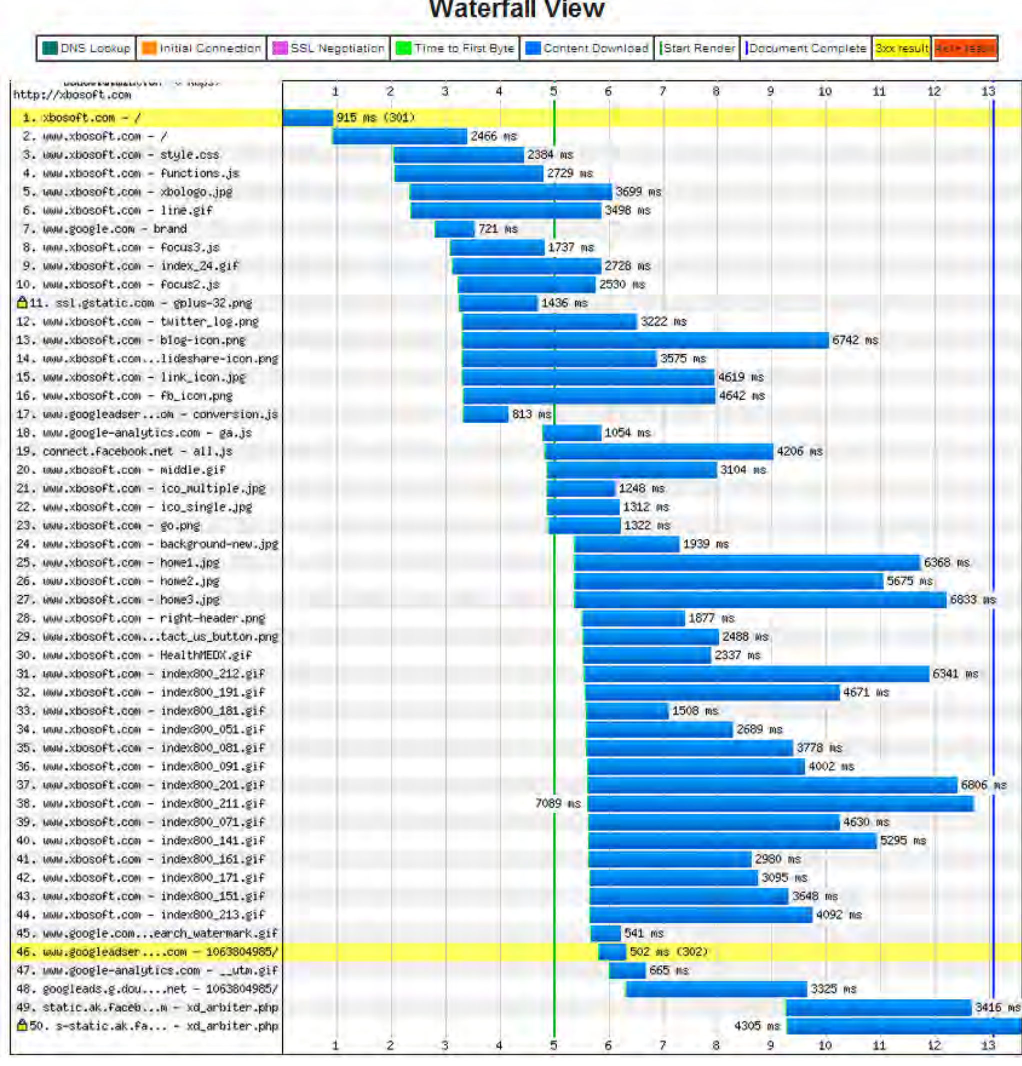


Figure 8: Waterfall View of time of each object downloaded

Next look at the repeat view when accessing the website with caching heavily used. Most of the page was successfully cached and resulted in a dramatic reduction in time.

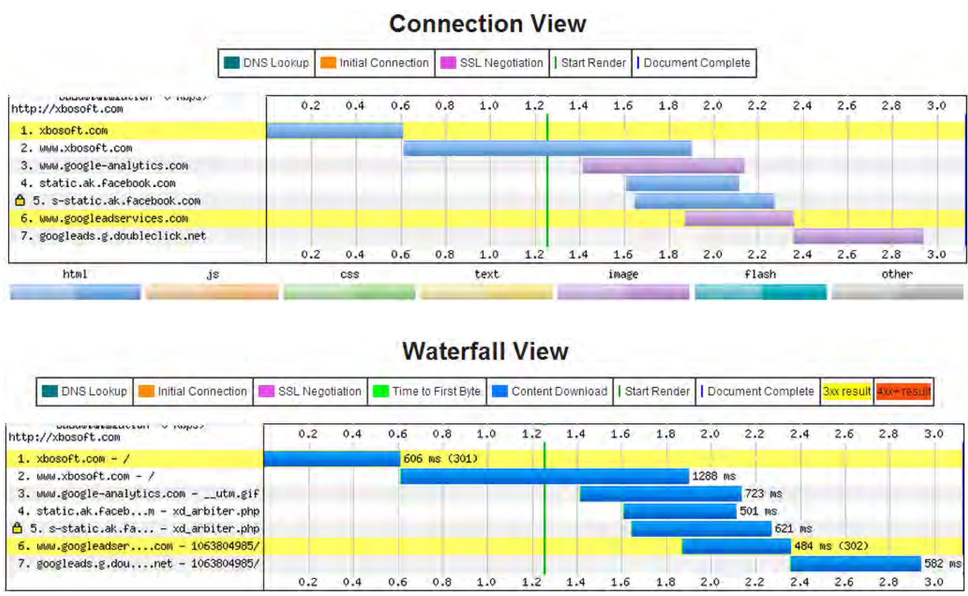


Figure 9: Connection and Waterfall View for repeat view download

Webpagetest.org provides a considerable amount more of detailed information to further pin point where performance enhances can be made.

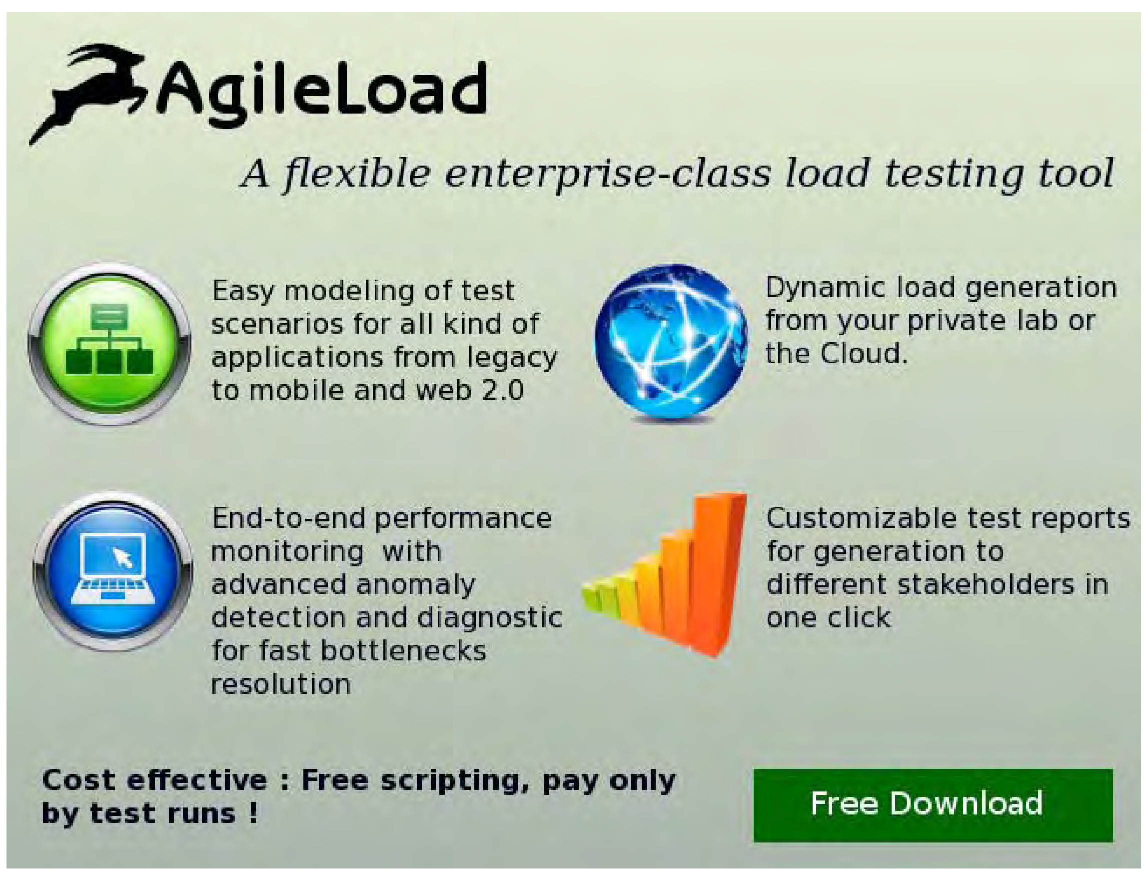
There are many good tools in the marketplace, but webpagetest.org is a good place to start in evaluating your server's performance. This quick and free test can bring immediate results for the most common and simplest problems.

Load Testing of Server

Your server has been load tested for your desktop website and now you are ready to bring your mobile app online. Some preliminary analysis of the mobile usage shows a lighter load on the server. This would be caused by smaller bandwidth demands on server connections and network latency which means that the server will receive fewer requests per second. Since the load on the server should be less for a mobile user than a desktop user, you feel fairly confident that the mobile rollout should go smoothly. This was the reasoning of the mobile project manager for an international bank who decided not to do any performance load testing before their mobile app went online.

This proved to be a disaster. The servers crashed as the mobile apps went online, and this was with only 5% of the users using mobile devices to access the server. The greater latency of the networks caused the server to hold open and extend sessions. This caused the backend server to use more memory and CPU time than the thresholds allowed, which caused the servers to crash. Performance load testing for mobile is sometimes seen as optional. This is a huge mistake as the mobile project manager for the international bank found out.

AgileLoad Flexible Enterprise Load Testing Tool - Click on ad to reach advertiser web site



To test this website for mobile access, you should start with a server that has been load tested, for example, 500 users for the desktop application. Then add one mobile app user at a time and monitor the effect on the server. In the above example, the transaction times doubled, degrading the system at just 10 mobile users, and the servers crashed when it approached 25 mobile users. [7]

Using a tool, for example HP LoadRunner along with Shunra vCat, you can simulate the 500 users and emulate mobile conditions for the mobile users.

Server Mobile Site Optimization Strategies: These strategies are applicable for both full and mobile site optimization, but are particularly applicable to the challenges of mobile performance.

Reducing the number of HTTP Requests: Round trips between the client and server cause the biggest degradation in page loading performance. Completing dozens of round trips to retrieve resources such as style sheets, scripts and images is especially damaging to performance for the mobile device because of the high latency of the network and relatively low bandwidth of the connection. Let’s look at a number of ways to reduce these requests.

Consolidating Resources: Consolidating JavaScript code and CSS styles into common file to be shared across multiple pages is standard practice. Images can also be consolidated, but you have to be careful to pay attention that your consolidated file isn’t too large for caching in local storage.

HTML5 Web Storage: Browser Caching, a standard optimization practice for desktops, doesn’t work as well for mobile devices because of the small size of local storage for caching. However HTML5 web storage can be an alternative. For all practical purposes, it is the same as caching, but much faster and more efficient.

Embed resources in html for first-time use: Inlining a resource, like an image, instead of a linked reference it can speed up the page load tremendously. However, again, you have to watch the size of your page so that it doesn’t become too large. This is commonly used with HTML5 web storage.

Unidirectional server updates: The server updates the client on a regular basis. For most applications, this does not require a bi-directional channel for back and forth communication with the client, but just an update at regular intervals. Of course, for messaging and gaming, bi-directional is required, but for information updates, a unidirectional channel is sufficient.

Reduce Payloads: Compress, resize and simplify. Sometimes the obvious is the best solution. To speed up payloads, smaller is better and faster. Not only does smaller reduce bandwidth consumption and latency, but it also can make the difference between a cacheable object and an uncacheable object. [8]

Compress: Compress on the server side and uncompress in the client. This can reduce the size up to 70% and tests have shown that this can significantly improve the performance.

Resize: Page loads the correct size image for the mobile device. Not only will this make the object smaller, but it saves processing so the client doesn’t have to resize the image. The small screen, many times, doesn’t require as high a resolution and finally, you can download a very low-resolution image at first, and then update it when the page has finished loading.

Simplify: HTML5 and CSS 3.0 provide many elements that used to require JavaScript. Updating your website to include these new standards will greatly simplify and decrease the size of your webpage.

Conclusion

Moore's Law for computer power and storage and Nielsen's Law for Internet bandwidth will ultimately solve mobile devices' current performance problems. This will lessen the need for mobile apps and m.sites. However, that could be well into the future, until then, choosing between making the full site mobile friendly, developing a mobile app, or promoting an m.site should be done carefully, being well aware of the tradeoffs.

References

1. http://www.shunra.com/products/shunra-vcat-mobile
2. Mobile consumers expect speed greater than many retailers are providing, Internet Retailer, August 7, 2012.
3. Case Study: The impact of HTML delay on mobile business metrics, Web Performance Today, November 2011.
4. 2012 State of Mobile Performance, strangeloopsnetworks.com, p. 6.
5. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html
6. http://marketingland.com/mobile-browser-speed-battle-ios-devices-faster-on-3g-android-faster-on-lte-23364
7. Why Mobile Apps Fail after Deployment, webinar by Shundra on Apr 16, 2012.
8. 2012 State of Mobile Performance, strangeloopnetworks.com, p 10.

SoftwareTestingMagazine.com - Click on ad to reach advertiser web site

Follow the current trends and news in software testing on

Software Testing Magazine . com

Agile Facilitation & Neuroscience: Transforming Information into Action

Cara Turner, http://facilitatingagility.com/
Khanyisa Real Systems, http://www.krs.co.za/

The face of meetings has changed fundamentally since coaches and ScrumMasters started including facilitation techniques into agile meeting structures. Many of us have experienced the productivity benefits of these collaborative practices first hand, so they're hard to deny. Yet it's difficult to put our finger on why they should make so much difference to our software development process.

At the same time neuroscience research has become increasingly popular as we try to make sense of the mysterious process of creating ideas and knowledge. As more information becomes available, there seems to be enough overlap in the research to suggest that agile facilitation really does help us make better decisions faster.

"*Snake oil!*" I hear you cry, but it's all got to do with how we process information, and a lot to do with a part of the brain called the prefrontal cortex.

This article takes a look at how neuroscience supports Agile facilitation methods, how we transform information into action, and which facilitation activities we can use to do so.

Facilitation and agile thinking

From the mid 2000's a group of agile thinkers began to focus on collaborative practices that support self-organizing teams, with authors Jean Tabaka, Esther Derby and Diana Larsen leading the way. The facilitation field boasts a number of collaboration practices, and these have been increasingly applied to running agile meetings and Scrum 'events', with facilitation now being a recognized ScrumMaster role.

These practices include time-boxing meetings and each section within them, keeping the agenda visible, silent writing, consensus techniques, and structured collaborative methods which ensure a high level of participation throughout the meeting. Thanks to the specific agile focus on continuous improvement, a variety of activities have evolved to examine our processes and products from different perspectives.

The benefits we see

Complexity theory, Lean Thinking, Systems Thinking and even Improv Theatre all bear out the benefits of agile facilitation practices, which allow:

- In-depth exploration of all factors affecting our current situation
- Collective knowledge of and input into product decisions as they need to be made
- Early identification of high-benefit activities
- Frequent re-assessments to identify changes in our systems, and
- Increasing flexibility to respond to unexpected change

Dan North's work on Deliberate Discovery also provides insight into the agile meeting flow. North describes software development as a process of overcoming a current state of ignorance by discovering new information about our product as we need it – whether that's the feature set or how we'll store data [1].

TinyPM Smart Agile Collaboration Tool - Click on ad to reach advertiser web site

FREE 5-USER
Community Edition

DOWNLOAD
http://www.tinypm.com/download

Tiny Effort, Perfect Management

Web-based, lightweight and smart agile collaboration tool with product management, backlog, taskboard, user stories and wiki.

Team collaboration

tinyPM let you focus on your project and the team by making all boring mechanics invisible.

Customer engagement

Great adoption within business leads to better project awareness within the whole team. Translated into 16 lanugages.

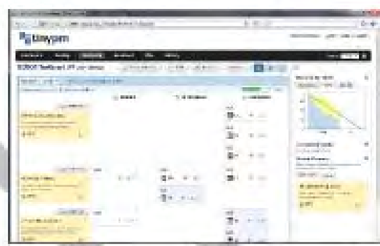
Agile management

tinyPM makes sure that all team members, clients, stakeholders feel comfortable with your agile process.

Integrations

tinyPM gets data from bug trackers, mail and more, so you can have all the information you need in one place.

Integrates with: Git, Mercurial SVN, Bitbucket, Github, JIRA, UserVoice, POP3, RSS/Atom



Now with Customizable Taskboard!
www.tinypm.com

Features

General
Local (on-site) installation and hosted edition
Multiple projects support
Advanced permission management
Timezones
Localized (16 languages)

Main functions
Dashboard with cross-project view
Sandbox (feature requests/ideas/bugs)
Backlog (Drag'n'Drop)
Taskboard (Drag'n'Drop)
Timesheet (time and budget tracking)
Wiki
Activity history

Release Planning
Grouping iterations into releases
Release delivery forecasts

Iterations
Iteration planning and tracking
Iteration goals

User stories
Estimation with customizable scale
MoSCoW priorities
Splitting user stories
Automatic work progress
Tags
Acceptance
Card colors
Changes history (versioning)
Attachments
Comments
Printing cards

Tasks
Progress
Converting tasks into stories
Moving tasks between stories
Multiple users assigned to a task
Estimation with customizable scale
Changes history (versioning)
Attachments
Comments

Metrics
Project burndown/burnup charts
Budget burndown charts
Iteration burndown charts (stories and tasks)
Velocity chart
Backlog progress display

Extensions
E-mail notifications
RSS feeds
REST API over HTTP
Plugins
Stories imports & export (CSV)

tinyPM is advanced, lightweight and smart tool for agile collaboration including product management, backlog, taskboard, user stories, wiki, integrations and REST API.

tinyPM goes beyond software development and encourages all team members (including clients, management and stakeholders) to actively participate in all your projects.

Contact us
support@tinypm.com

Follow us
twitter.com/tinypm

From this perspective we can see how the agile meeting flow supports the process of just-in-time decision-making. We explore the product needs and our available options through the various grooming activities, planning for releases and sprints, reviewing sprints, retrospecting and re-planning, to uncover the right amount of detail at the right time.

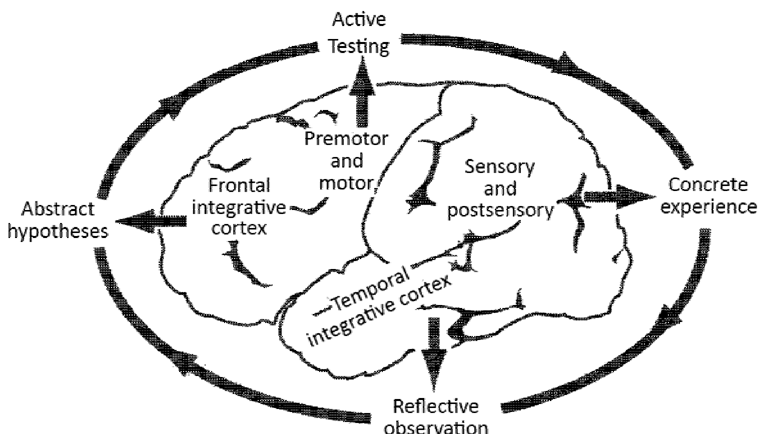
But none of this explains *why* facilitation methods work so well. The key seems to lie in the physical details of how we process information internally.

Agile Facilitation and Neuroscience

It's about how we learn

Prof. James Zull is a biologist and educator who became intrigued with the process by which we learn. Through exploring various theories of learning, he encountered David Kolb's work on Experiential Learning [2], and had the remarkable insight that from a biologist's viewpoint, the process described by Kolb maps directly to the physical makeup of the brain. [3]

In *The Art of Changing the Brain* Zull describes the major activities of experiential learning as they take place in various areas of the cortex (the soft grey outer layer of the brain), and their correlation to Experiential Learning.



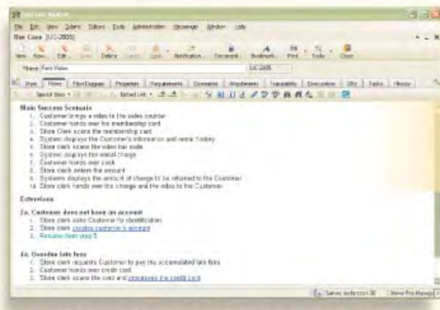
Information is received through the sensory and post-sensory areas of the cortex at the back of the brain (Kolb's Concrete Experience) which we process via the temporal integrative cortex by comparing it with knowledge we have previously acquired. We search our memories for associations or patterns that match our new data, and adjust our existing neural networks to accommodate this data (Reflective Observation).

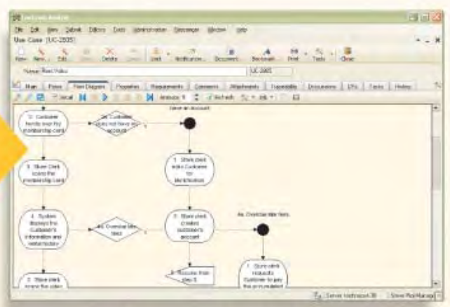
Once we've sifted through the data to understand it, our prefrontal cortex formulates an idea about the information we've received, why it's there, and what we can do with it (Abstract Hypotheses).

We then use our pre-motor and motor cortex to put our ideas into action (Active Testing). Zull makes the surprisingly obvious observation that eye movements, speech and writing are all motor activities and hence action. 'Progress' as we understand it is also deeply connected with moving – we move ahead or fall behind; we reach for new goals.

Having acted on our hypothesis, we follow this by sensing again, sifting the data we get back through our temporal integrative cortex filters, and make sense of it all anew.

Good Requirements = Better Software - Click on ad to reach advertiser web site





Take the pain out of writing Use Cases
with an advanced flow editor. . .

. . .then visualize your Use Case flow
with a single click. Automatically. Instantly!

TopTeam Analyst™

Good Requirements = Better Software™

TopTeamAnalyst enables you to capture requirements effectively, document clearly and communicate unambiguously. It will help you to build systems that meet your users' needs and reduce project risks. TopTeamAnalyst users have told us they love using it. Find out for yourself today!

There's much more online at...
www.TopTeamAnalyst.com

- View flash demos with screenshots and examples
- Download, unzip and run – no installation needed!
- Check out the price list and limited time offer

Click here to find out more

1-877-20-TOP-TEAM

Techno Solutions

Advanced Use Case Modeling

- Advanced flow editor with automatic renumbering and synchronization of Alternate Flows reduces tedious rework
- Automatic conversion of Use Case flow to diagram
- Wizards and Guidelines to help write Use Cases easily
- Integrated Use Case diagramming tool
- Single click output to MS Word document
- Release management to facilitate iterative development
- Integrated Change Management and Issue Tracking

Complete Requirements Capture and Management

- WYSIWYG hierarchical Requirements Document editor
- Rich text editor enables you to use Bullets, Tables, Images, OLE embedding to document Requirements effectively
- Advanced Traceability tools
- Complete set of diagramming tools – Context Diagram, Navigation Map Diagram, Screen Prototyping to help you express Requirements clearly
- Every artifact is stored in a multi-user, multi-time zone, versioned repository for distributed teams
- Advanced Notification, Threaded discussions and email integration for team collaboration

The correlations between this cycle and Deming's PDCA cycle of Plan, Do, Check, Act are noteworthy, fundamentally supporting the continuous improvement cycle baked into agile via regular sprintly retrospectives.

The term 'experiential learning' is also consistently applied to the way we learn agile practices and processes. We can't 'become agile' by reading books: we have to experience, assimilate, experiment and review in order to 'get it'.

But what about that prefrontal cortex?

In *Your Brain at Work* neuroscientist David Rock describes in detail the process of cognitive thought: the activities governed by the prefrontal cortex. This is the region we use when we're consciously thinking about anything, and its main functions are: Understanding, Deciding, Recalling, Memorizing and Inhibiting.[4]

The prefrontal cortex seems like our brain's equivalent of a computer's CPU: we use it to hold a thought in mind, to call up memories, to make comparisons and to make decisions. Like a CPU, the prefrontal cortex has a very limited processing capacity – but unlike a computer, we can't just add hardware to make it go faster. So we have to change the way we work to make it more effective.

Conceptual Challenges

One of the well-known constraints of our short-term memory is that we can recall around 7 simple concepts at one time. However, most of the concepts we deal with are complex, particularly in IT. Ideas map to further ideas and associations, and yet further memories and connections.

Calling up the idea of a smart phone could bring to mind: social connectivity, apps, mobile design, platform, compatibility, email, appointments, plans… not to mention phone calls.

According to Rock, when we're actively thinking about complex concepts, we can only hold one in mind without any degradation in quality. For comparisons, two are optimal, and for decision-making we can hold no more than four, but at this point the quality of recall has degraded sufficiently that we aren't able to focus on a whole picture.

If we think about agile planning, with systems made up of architectural layers and interconnecting applications, we can see that attempts to model new feature sets in our mind alone poses a significant challenge.

Another constraint of our internal CPU is that conscious work burns metabolic fuel fast. Because of all the connected associations, working with just one concept depletes our processing capacity, and activities like planning and prioritizing which require all of the prefrontal cortex activities, exhaust us quickly.

SpiraTeam Complete Agile ALM Suite - Click on ad to reach advertiser web site

It's time to try a better way.

The most complete yet affordable
Agile ALM suite on the market today.

Learn more at: inflectra.com/spiraTeam

www.inflectra.com
sales@inflectra.com
+1 202-558-6885

This may have something to do with why we tend to want to avoid them. Long and exhausting or frequently rescheduled grooming sessions are a sign that the deeper level system planning is missing. Keeping grooming at the 'appropriately detailed' level allows us to get the depth of knowledge we need as we need it, without wasting precious CPU capacity.

Focus!

We're more easily distracted when we're tired – which makes sense now that we know that *inhibiting* other thoughts is part of the prefrontal cortex's role. When everything is competing for our attention and we aren't able to filter and focus on the important connections, we lose the ability to hold all the relevant data in mind. We tend to make poor decisions not only about the work we're doing but also where to place our attention.

As a short-term fix we can replenish the sugars our mental activities deplete, but this is not sustainable and also has other health implications, so we need to schedule frequent breaks. Another way to recharge is simply to change our activities to give different areas of the brain a rest.

Moving from active thinking to more energy-efficient activities served by long-term memory (anything that we can run on 'autopilot' for a while) allows us to come back to the heavier work a bit later with more clarity.

These insights together explain why it helps to time-box activities, particularly intensive exercises such as grooming and prioritizing. Scheduling them early or after a short break when we're more alert will also help here.

Our ability to inhibit other thoughts is also what allows us to place deep attention on one thing. So doing things like setting aside specific time for planning and integrating activities such as grooming, sprint planning and retrospectives is another way we can improve the quality of our attention – ideally away from our desks and day-to-day distractions so that we don't have to inhibit these as well.

What we're good at

Fortunately, our prefrontal cortex has lots of strengths too, that help reduce load on the limited CPU.

Use our senses

To start with, molecular biologist John Medina explains that "Our senses evolved to work together – vision influencing hearing, for example – which means that we learn best if we stimulate several senses at once." Since each sensory input is recorded in different parts of the brain, by incorporating physical / spatial, visual, sound and even taste and smell elements, we create much more vivid associations, and hence have better memory recall. [5]

Medina also emphasizes the importance of movement in its own right. Exercise improves the flow of blood to our brains, bringing glucose for thinking, and oxygen to clean up the chemical waste produced by thinking (the brain uses the most glucose of any organ in the body). It also helps keep our neurons able to connect to each other. While a bit of movement in Sprint Planning is hardly a workout, experiments indicate that even a small amount of exertion shows improved results over no movement.

This helps to explain why interaction in meetings – even the simple steps of collaborative writing and getting up to stick post-its on a flipchart – makes a difference.

Get graphic

Zull, Rock and Medina all emphasize that we have an excellent capacity for processing visual imagery. Images and sketches are highly efficient means for communicating detailed information; it's much easier to convey a concept in a graphical representation showing relationships between people and objects than it is to explain it using language - as witnessed by the rise of the infographic.

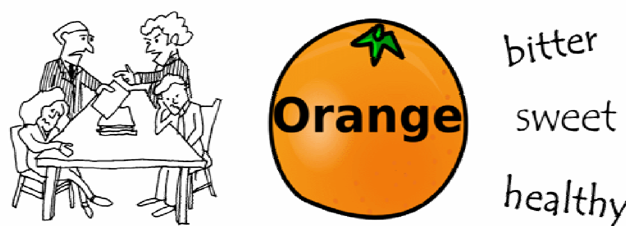
Going metaphorical

Combined with our pattern matching ability, our visual processing gives us the capacity to create and interpret metaphors and associations powerfully. Just as it's easier to communicate ideas via images than words, we find it easier to integrate information by creating connections with things we already know and understand.

Although our associations are often spoken or written, the effect is to create a mental image. Metaphors work by applying the range of connotations and associations we have with a familiar item to a different item or process to deepen our understanding.

Metaphors are particularly useful in agile facilitation to get us out of our jargon use, which is notoriously easy to misunderstand.

This process generates new connections that may be unconscious, hidden or surprising, which extend or strengthen our existing neural networks. This in turn helps us to remember, visualize and think creatively about an issue.



While it may seem counter-intuitive at first, thinking in metaphors is a communication skill we all utilize naturally, and one that teams become increasingly fluent with over time.

So apologies to the skeptics: there's good neuroscience for comparing your last sprint to a fruit …

Mental mapping

Pattern matching also gives us the ability to build up 'template' patterns when we've seen and done something many times. These templates seem to represent a specific state of a concept, building up our own shorthand for complex concepts.

The main advantage is to be able to visualize different routes to and from a current state, for better decision-making, but it also helps us to do another energy-saving tactic for thinking, which is to 'chunk' information. This is a form of mental mapping in which we consciously

identify 3 - 4 key associations with each idea, and use this summary form for decision-making. This keeps the information level rich and our active-memory effort low.

Write it down

Another efficiency that Rock describes is to get the ideas and concepts out of our head by writing them down so that we don't have to use energy remembering them.

If we think of planning and retrospective activities when we do this as a group, the notes that others make also prompt our own memories, which helps to flesh out a fuller picture. Instead of trying to recall all the relevant associations and retain focus on each item while making decisions, this process frees up our CPU to sort, theme, compare, weight and select without the concern of overlooking important data.

I sometimes wonder where agile methodologies would be without the humble post-it note...

Think deeply

One *disadvantage* of our pattern matching ability is that we are quick to draw connections using only short-term memory and easy-to-access information. Thinking through an issue thoroughly requires us to search through longer term memories. This takes a fair amount more energy for our brains. When we're rushed or tired we don't go to the effort of finding the deeper associations, and end up working from incomplete data which leads to poorer quality conclusions.

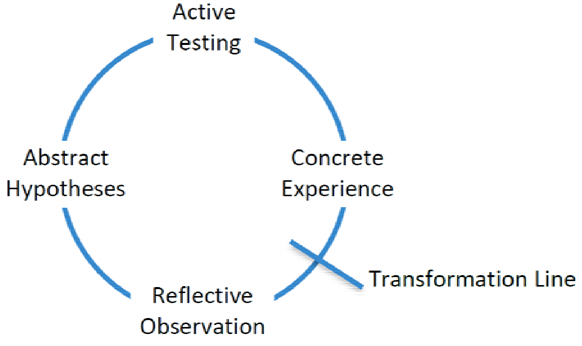
In terms of problem solving in agile, this explains the value of longer exercises that are intended to jog our memories - the 'Gathering Data' section in retrospectives, as well as root cause analyses such as '5 Whys' and Cause and Effect diagrams. Taking time to think more clearly about what we know leads to much higher quality of information – and now we know why.

Acting on our knowledge

Now that we understand more about *how* we think, let's look at turning information into active knowledge.

Coming back to the physicality of our brain, James Zull refers to this process as a "Transformation Line" – bridging the Reflective Observation function and Active Testing area of our brains.

Although his focus is university students, Zull's words echo almost exactly the purpose of retrospectives:



"Data enters learners through concrete experience where it is organized and rearranged through reflection. But it is still just data until learners begin to work with it. When learners convert this data into ideas, plans, and actions, they experience the transformation I have described. Things are now under their control, and they are free of the tyranny of information. They have created and are free to continually test their own knowledge." [3]

Agile retrospectives are our tool to make sure we truly do make this transformation: take our information and turn it into meaningful ways in which we can act to improve our world.

A range of Retrospective choices

There are many retrospective activities we can use to explore of our products and processes for improvements. But different formats will give us very different outcomes. Which game should we use, and why?

It helps that most retrospective activities fit into one of four categories:

	Analysis	Imagination
Visual	Collaborative Sorting	Generating Ideas
Immersive	Simulations	Experience Prototyping

Visual activities work at the idea level, helping us to get thoughts out of our head, to study them and make collaborative decisions.

Immersive games work at the experiential level and provide much deeper learning by engaging all our senses and bringing awareness to the interactions and decisions we make instinctively. This is the space of rapid learning.

Analytical games investigate our current situation, while *imaginative* games work to generate new insights and ideas.

The games mentioned below are all listed in the reference section.

Visual Activities

Collaborative Sorting

At the most accessible level, we have "games" that are simply collaborative sorting activities. These explore the relationships between events and facts about the sprint, and while they may make use of metaphor to create related sorting categories, they don't challenge our imagination.

Games such as *Liked, Lacked, Learned, Longed For* and learning matrixes create a framework that makes it is easier to identify the specific details that we can act on to improve right now, from the multitude of things we think and know.

Root cause formats like *Cause and Effect diagram* help us get beyond surface thinking by applying deep questioning; while *Speed Boat* type games start to bring in metaphor.

From what we know about brain science, this format trumps a Meeting to Discuss Improvements primarily by creating a visual focus – it's much easier to give valuable input when we are literally *focused* on it. The Gamestorming authors call this "meaningful space" – somewhere where we can juxtapose and contemplate the issues that require our attention.[6]

Generating Ideas

Visual imagination exercises help us capture ideas from new perspectives.

Brainstorming activities such as *Random Input* use association to drive new connections, while 'futurespecting' exercises such as *Pre-Mortem* look from a future date to imagine what could happen in our projects, as if they already have.

Drawing exercises such as *Draw the Problem* and *Poster Session* connect with different processes in our brains (thinking visually is different from communicating visually), leading to rich and frequently surprising insights.

Resistance to these games now makes sense. We may have concerns about our creative ability, but also thinking back to our brain functions, this kind of work can be very taxing. We need to change our frame of reference, inhibit distractions, generate concepts, and try to interpret them – all at the same time.

David Rock also notes that we need a certain amount of stress to reach an optimal state of creative flow. With too little stress we're unmotivated to exert ourselves, but too much shuts down our imagination as we go into survival mode. So it helps to have working agreements that create a sense of comfort while maintaining creative tension.

Quantity trumps quality here, so there are no bad questions and no ideas rejected. This helps us to get existing ideas out of our heads quickly, to free up space to search deeper for different ideas. 'Bad' ideas often spark conversation that produces excellent ones. Silly, funny ideas are welcome too, as humor has a relaxing effect, which eases our creative concerns and frees up our associative abilities.

Teams new to imaginative work may want to end by clarifying concrete steps to take further, so that the input feels channeled somewhere. As teams become more experienced, this becomes less important. Ideas can be left to surface organically which helps avoid 'anchoring' our thinking too soon, leading to far more profound insights.

Immersive activities

Immersive activities take us more deeply into the realm of play – the practice of learning by doing.

Simulations

Simulations like the *Kanban pamphlet game* and *Jenga testing game* allow us to learn behaviors relevant to a process, encounter tricks and traps, and discover solutions. The internalized experience of falling into a trap and finding our way out is the most powerful learning here – this is a true safe to fail environment.

Games have set objectives, repeated cycles with small variations, and of course twists that interrupt our carefully constructed plans. These help participants understand our behaviors under different systems. By incorporating the realities of software development, they also teach us the value of resilience.

Lego simulations take this further. The activity of building in a safe-to-fail environment teaches us the mechanics of collaborative projects. Being able to tear down structures easily, to share insights and work together to build new ones quickly, models the kind of ways we can interact while building software together.

This can also be taken into creative innovation sessions using activities like Gamestorming's *Make a World.* Design company IDEO's CEO Tim Brown calls it as playful building, or thinking with our hands.

Experience Prototyping

Experience Prototyping is the name IDEO gave to activities such as role playing and *Bodystorming*, which they and Nordström's Innovation Lab use to get out of traditional thinking.

The idea behind these is to 'act out' a service situation or application, with people taking the 'roles' of both interfaces and the users of the system, to unearth the unexpected behaviors and expectations consumers actually have.

In his TEDTalk "*Tales of Creativity and Play*" Brown emphasizes that, aside from embarrassment, our 'adult' distrust of play seems to be that we don't believe the insights generated from play will be valid in the 'real world'. [7]

IDEO's research into play challenges this: kids' play is not the unstructured free-for-all we imagine with our 'serious' workday minds. Instead, kids follow set scripts learnt from adults in order to create life-like situations in which they can 'try on' and practice future roles. Children are quick to note when the rules of the play have been broken.

In the same way, when we 'try on' an experience we also set up clear conditions so that it's easy to spot when we're not being authentic. In fact, we act with as much authenticity as our embarrassment will let us, and consequently the quality of the discoveries and insights is significant.

It's no surprise that the area in which these activities have found most traction is in the design phase, generating new and deeper insights into our products and customers, paving the way to better customer experiences as well as product innovation.

An Empathy Tool

Actual role-playing focuses more deeply on human interaction – Brown describes role-play as both a tool for prototyping experiences and an empathy tool.

You know how often you look back on a bad conversation and think "if only I had done that instead of this"? At one level our brains can't distinguish between fiction and reality – sections of our brain are dedicated to "theory of mind" where we seem to model in our own mind the things we see others doing, to try to understand their motivation. This is not only a mental process but also has a physiological effect so that we almost *are* experiencing what others are doing.

When we play another character, we become acutely more aware of the needs and incentives driving others, as well as highlighting our personal behaviors and biases. And simply by trying out different behaviors in a situation, we discover new ways to respond to them in our 'real life' scenarios. While these lessons may not help resolve a past situation, they may well help to understand it better, and create new possibilities for when we encounter similar scenarios.

It's not 'All or Nothing'

We still need to balance play and serious activities – Brown notes that kids naturally move between these two states, but as adults we tend to fear that they are absolute, we're either playful or serious.

Thinking about retrospectives, it makes sense that we include different kinds of kinds of activities for generating ideas than for acting on them. Divergent, exploratory activities are where we most need creative and playful input, to give us the widest possible set of options. We then bring our serious attention back for the convergent activities of selecting ideas and acting on them.

Putting it all together

As we have seen, our minds are powerful tools, but our active thinking capacity is limited, so we need to use it wisely, particularly when working with complex concepts.

Neuroscience sheds some light on how agile facilitation supports our thinking process:

- Working with the right amount of detail at the right time keeps our focus clear

- Moving to a separate space away from our normal details helps to create a mental break and focus on the work for each meeting
- Planning an agenda to include movement and variety improves our ability to think and focus
- Understanding our processes and looking for improvements needs particularly deep attention, for which collaborative facilitation activities are most beneficial
- Creative and playful exploration generates profoundly new insights, to create those innovative ideas that are really the same thing looked at from a different angle

All of which really do help us to reach the best decisions, fast.

References for: Agile Facilitation & Neuroscience

1. Deliberate Discovery: http://dannorth.net/2010/08/30/introducing-deliberate-discovery/
2. David Kolb's work on Experiential Learning: http://www.learning-theories.com/experiential-learning-kolb.html
3. James E. Zull The Art of Changing the Brain Illustrations: p 18 and 40
4. David Rock Your Brain at Work
5. John Medina Brain Rules: 12 Principles for Surviving and Thriving at Work, Home, and School
6. Dave Gray, Sunni Brown & James Macanufo Gamestorming: A Playbook for Innovators, Rulebreakers, and Changemakers
7. Tim Brown talk 'Tales of Creativity and Play': http://www.ted.com/talks/tim_brown_on_creativity_and_play.html

Facilitation Activities:

Collaborative Sorting:

8. 5 Whys: http://www.boxtheorygold.com/blog/bid/16814/Try-the-5-Whys-Problem-Solving-Tool
9. Cause and Effect Diagrams: http://www.crisp.se/file-uploads/cause-effect-diagrams.pdf
10. Liked, Lacked, Learned, Longed For: http://ebgconsulting.com/blog/the-4l%E2%80%99s-a-retrospective-technique/
11. Speed Boat: http://www.gogamestorm.com/?p=608

Idea Generating:

12. Random Input: http://members.optusnet.com.au/charles57/Creative/Techniques/random.htm
13. PreMortem: http://inevitablyagile.wordpress.com/2011/03/02/pre-mortem-exercise/
14. Draw the Problem: http://www.gogamestorm.com/?p=373
15. Poster Session: http://www.gogamestorm.com/?p=419

Simulations:

16. Kanban pamphlet game: http://www.agilistapm.com/learn-kanban-by-playing-a-game/
17. Jenga testing game: http://nandalankalapalli.wordpress.com/2011/09/15/game-test-small-test-often
18. Lego simulations: http://vinylbaustein.net/tag/lego/

Experience Prototyping:

19. Make a World: http://www.gogamestorm.com/?p=591
20. Bodystorming: http://www.gogamestorm.com/?p=344

Test Driven Traps

Jakub Nabrdalik, TouK, touk.pl
Jakubn [at] gmail.com, blog.solidcraft.eu

Have you ever been in a situation, where a simple change of code broke a few hundred tests? Have you ever had the idea that tests slow you down, inhibit your creativity, make you afraid to change the code? If you had, it means you've entered the Dungeon-of-very-bad-tests, the world of things that should not be.

I've been there. I've built one myself. And it collapsed killing me in the process. I've learned my lesson. So here is the story of a dead man. Learn from my faults or be doomed to repeat them.

The story

Test Driven Development, like all good games in the world, is simple to learn, hard to master. I've started in 2005, when a brilliant guy named Piotr Szarwas, gave me the book "Test Driven Development: By Example" [1], and one task: create a framework.

Those were the old times, when the technology we were using had no frameworks at all, and we wanted a cool one, like Spring, with Inversion-of-Control, Object-Relational Mapping, Model-View-Controller and all the good things we knew about. And so we created a framework. Then we built a Content Management System on top of it. Then we created a bunch of dedicated applications for different clients, Internet shops and what-not, on top of those two. We were doing well. We had 3000+ tests for the framework, 3000+ tests for the CMS, and another few thousand for every dedicated application. We were looking at our work, and we were happy, safe, and secure. Those were good times.

And then, as our code base grew, we came to the point, where a simple anemic model we had, was not good enough anymore. I had not read the other important book of that time: "Domain Driven Design" [2], you see. I didn't know yet, that you could only get so far with an anemic model.

But we were safe. We had tons of tests. We could change anything.

Or so I thought.

I spent a week trying to introduce some changes in the architecture. Simple things really: moving methods around, switching collaborators, such things. Only to be overwhelmed by the number of tests I had to fix. That was TDD, I started my change with writing a test, and when I was finally done with the code under the test, I'd find another few hundred tests completely broken by my change. And when I got them fixed, introducing some more changes in the process, I'd find another few thousand broken. That was a butterfly effect, a chain reaction caused by a very small change.

It took me a week to figure out, that I'm not even half done in here. The refactoring had no visible end. And at no point my code base was stable, deployment-ready. I had my branch in the repository, one I've renamed "Lasciate ogni speranza, voi ch'entrate". [3]

We had tons and tons of tests. Of very bad tests. Tests that would pour concrete over our code, so that we could do nothing.

NeoLoad Performance Testing Solution - Click on ad to reach advertiser web site

The Load & Performance Testing Solution for all web & mobile applications

“ We were able to identify the system bottleneck before it was a real problem. ”

Chris McCarthy - Terremark

NeoLoad is a load & performance testing solution for web and mobile applications that improves testing effectiveness. It enables faster tests, provides pertinent analysis and supports the newest technologies.

Test your Web application's performance easily and ensure trouble-free deployment thanks to NeoLoad!

Support for HTTP, AJAX, Flex, GWT, Silverlight, Java serialization, Push technologies, Oracle Forms, Siebel...

Generate Virtual Users from:

Your internal infrastucture

The Neotys Cloud Platform

Download Free Trial

www.neotys.com

E-mail: sales@neotys.com

The only real options were: either to leave it be, or delete all tests, and write everything from scratch again. I didn't want to work with the code if we were to go for the first option, and the management would not find financial rationale for the second. So I quit.

That was the Dungeon I built, only to find myself defeated by its monsters.

I went back to the book, and found everything I did wrong in there. Outlined. Marked out. How could I skip that? How could I not notice? Turns out, sometimes, you need to be of age and experience, to truly understand the things you learn.

Even the best of tools, when used poorly, can turn against you. And the easier the tool, the easier it seems to use it, the easier it is to fall into the trap of I-know-how-it-works thinking. And then BAM! You're gone.

The truth

Test Driven Development and tests, are two completely different things. Tests are only a byproduct of TDD, nothing more. What is the point of TDD? What does TDD brings? Why do we do TDD?

Because of those three reasons.

1. To find the best design, by putting ourselves into the user's shoes.

By starting with “how do I want to use it” way of thinking, we discover the most useful and friendly design. Always good, quite often that's the best design out there. Otherwise, what we get may be theoretically correct, but terribly hard to use.

Do you remember EJB 1 or 2? CORBA? Any technology designed by a committee, is going to be “correct”, but overengineered and painful to use. And you don't want that.

The best design is discovered by the one who has to use it.

2. To manage our fear.

It takes balls, to make a ground change in a large code-base without tests, and say “it's done” without introducing bugs in the process, doesn't it? Well, the truth is, if you say “it's done”, most of the time you are either ignorant, reckless, or just plain stupid. It's like with concurrency: everybody knows it, nobody can do it well.

Smart people are scared of such changes. Unless they have good tests, with high code coverage.

TDD allows to manage our fears, by giving us proof, that things work as they should. TDD gives us safety

3. To have fast feedback.

How long can you code, without running the app? How long can you code without knowing whether your code works as you think it should?

Jama Contour - Click on ad to reach advertiser web site

Leverage your collective genius.
Deliver successful projects.

At Jama, we believe collaboration is the key to success. Teams developing complex systems, software and other products use Contour, the powerful Web-based requirements management software, to manage the detailed scope of projects through the development lifecycle. People love Contour because it keeps everyone connected, fits any process and is elegantly simple to use.

Try Contour free for 30 days.

No installation needed. Sign up now for your Contour hosted trial.

www.jamasoftware.com
Email us: info@jamasoftware.com
Call us: 1 (800) 679-3058

Feedback in tests is important. Less so for front-end programming, where you can just run the shit up, and see for yourselves. More for coding in the backend. Even more, if your technology stack requires compilation, deployment, and starting up.

Time is money, and I'd rather earn it than wait for the deployment and click through my changes each time I make them.

And that's it. There are no more reasons for TDD whatsoever. We want Good Design, Safety, and fast Feedback. Good tests are those, which give us that [4] Ok, I'm lying a bit in here. Authors like Kent Beck and Tomasz Kaczanowski mention more reasons. For example: using TDD in communication (Behavior Driven Development and user stories are a great example). In my practice, though, I've found only those three to be of a great importance. Others were easily replaced with different (sometimes better) tools and methods than TDD. Your mileage may vary, so find out what's best for you..

Bad tests?

All the other tests are bad.

The bad practice

So how does a typical, bad test, look like? The one I see over and over, in close to every project, created by somebody who has yet to learn how NOT to build an ugly dungeon, how not to pour concrete over your code base. The one I'd write myself in 2005.

This will be a Spock sample, written in groovy, testing a Grails controller. But don't worry if you don't know those technologies. I bet you'll understand what's going on in there without problems. Yes, it's that simple. I'll explain all the not-so-obvious parts.

```
def "should show outlet"() {
  given:
    def outlet = OutletFactory.createAndSaveOutlet(merchant: merchant)
    injectParamsToController(id: outlet.id)
  when:
    controller.show()
  then:
    response.redirectUrl == null
}
```

So we have a controller. It's an outlet controller. And we have a test. What's wrong with this test?

The name of the test is “should show outlet”. What should a test with such a name check? Whether we show the outlet, right? And what does it check? Whether we are redirected. Brilliant? Useless.

It's simple, but I see it all around. People forget, that we need to:

Trick 1: Verify the right thing

I bet that test was written after the code. Not in test-first fashion.

But verifying the right thing is not enough. Let's have another example. Same controller, different expectation. The name is: "should create outlet insert command with valid params with new account"

Quite complex, isn't it? If you need an explanation, the name is wrong. But you don't know the domain, so let me put some light on it: when we give the controller good parameters, we want it to create a new OutletInsertCommand, and the account of that one, should be new.

The name doesn't say what 'new' is, but we should be able to see it in the code.

Have a look at the test:
```
def "should create outlet insert command with valid params with new account"() {
  given:
    def defaultParams = OutletFactory.validOutletParams
    defaultParams.remove('mobileMoneyAccountNumber')
    defaultParams.remove('accountType')
    defaultParams.put('merchant.id', merchant.id)
    controller.params.putAll(defaultParams)
  when:
    controller.save()
  then:
    1 * securityServiceMock.getCurrentlyLoggedUser() >> user
    1 * commandNotificationServiceMock.notifyAccepters(_)
    0 * _._
    Outlet.count() == 0
    OutletInsertCommand.count() == 1
    def savedCommand = OutletInsertCommand.get(1)
    savedCommand.mobileMoneyAccountNumber == '1000000000000'
    savedCommand.accountType == CyclosAccountType.NOT_AGENT
    controller.flash.message != null
    response.redirectedUrl == '/outlet/list'
}
```

If you are new to Spock: n*mock.whatever(), means that the method “whatever” of the mock object, should be called exactly n times. No more no less. The underscore “_” means “everything” or “anything”. And the >> sign, instructs the test framework to return the right side argument when the method is called.

So what's wrong with this test? Pretty much everything. Let's go from the start of “then” part, mercifully skipping the over-verbose set-up in the “given”.

```
1 * securityServiceMock.getCurrentlyLoggedUser() >> user
```

The first line verifies whether some security service was asked for a logged user, and returns the user. And it was asked EXACTLY one time. No more, no less.

Wait, what? How come we have a security service in here? The name of the test doesn't say anything about security or users, why do we check it?

Well, it's the first mistake. This part is not, what we want to verify. This is probably required by the controller, but it only means it should be in the “given”. And it should not verify that it's called “exactly once”. It's a stub for God's sake. The user is either logged in or not. There is no

sense in making him “logged in, but you can ask only once”.
Then, there is the second line.

```
1 * commandNotificationServiceMock.notifyAccepters(_)
```

It verifies that some notification service is called exactly once. And it may be ok, the business logic may require that, but then... why is it not stated clearly in the name of the test? Ah, I know, the name would be too long. Well, that's also a suggestion. You need to make another test, something like “should notify about newly created outlet insert command”.

And then, it's the third line.

```
0 * _._
```

My favorite one. If the code is Han Solo, this line is Jabba the Hut. It wants Hans Solo frozen in solid concrete. Or dead. Or both.

This line, if you haven't deducted yet, is “You shall not make any other interactions with any mock, or stubs, or anything, Amen!”.

That's the most stupid thing I've seen in a while. Why would a sane programmer ever put it here? That's beyond my imagination.

No it isn't. Been there, done that. The reason why I wrote such a thing is to make sure, that I covered all the interactions. That I didn't forget about anything. Tests are good, what's wrong in having more good?

I forgot about sanity. That line is stupid, and it will have it's vengeance. It will bite you in the ass, some day. And while it may be small, because there are hundreds of lines like this, someday you gonna get bitten pretty well. You may as well not survive.

And then, another line.

```
Outlet.count() == 0
```

This verifies whether we don't have any outlets in the database. Do you know why? You don't. I do. I do, because I know the business logic of this domain. You don't because this tests sucks at informing you, what it should.

Then there is the part, that actually makes sense.

```
  OutletInsertCommand.count() == 1
   def savedCommand = OutletInsertCommand.get(1)
   savedCommand.mobileMoneyAccountNumber == '1000000000000'
   savedCommand.accountType == CyclosAccountType.NOT_AGENT
```

We expect the object we've created in the database, and then we verify whether it's account is “new”. And we know, that the “new” means a specific account number and type. Though it screams for being extracted into another method.
And then...

```
   controller.flash.message != null
   response.redirectedUrl == '/outlet/list'
```

Then we have some flash message not set. And a redirection. And I ask God, why the hell are we testing this? Not because the name of the test says so, that's for sure. The truth is, that looking at the test, I can recreate the method under test, line by line.

Isn't it brilliant? This test represents every single line of a not so simple method. But try to change the method, try to change a single line, and you have big chance to blow this thing up. And when those kinds of tests are in the hundreds, you have concrete all over you code. You'll be able to refactor nothing.

So here's another lesson. It's not enough to verify the right thing. You need to

Trick 2: Verify only the right thing

Never ever verify the algorithm of a method step by step. Verify the outcomes of the algorithm. You should be free to change the method, as long as the outcome, the real thing you expect, is not changed.

Imagine a sorting problem. Would you verify it's internal algorithm? What for? It's got to work and it's got to work well. Remember, you want good design and security. Apart from this, it should be free to change. Your tests should not stay in the way.

How should the test above look like? First, let’s notice, that there are three potential business rules worth testing in here.

First, when we create outlet, we should have an OutletInsertCommand in our database, but we shouldn’t have an Outlet yet (the change has to be accepted by a supervisor). Those two lines check that:

```
Outlet.count() == 0
OutletInsertCommand.count() == 1
```

Second, the supervisor (called acceptor in here) should be notified. And this is verified as well:

```
1 * commandNotificationServiceMock.notifyAcceptors(_)
```

Third, we can see, that the command in question, should have a “new account”, and this state is represented like that:

```
def savedCommand = OutletInsertCommand.get(1)
savedCommand.mobileMoneyAccountNumber == '1000000000000'
savedCommand.accountType == CyclosAccountType.NOT_AGENT
```

Everything else is either irrelevant to the business goal, or just complete rubbish. We should be able to change everything else, as long as those three rules are not broken.

So what should we do? Is it enough to delete everything else from the test and just leave the lines mentioned above? Not really.

If we started with a test first, we would have those three requirements represented as three different tests, because we start with a name of the test only. And that’s a much better solution, because every time we break a business expectation, we will know exactly which rule we broke. In other words, it’s better for a test, to have just a single business reason to fail.

So let’s write those test right.

```
def setup() {
  userIsLoggedIn()
}
```

```
private void userIsLoggedIn() {
  securityServiceMock.getCurrentlyLoggedUser() >> user
}

private void setValidOutletInsertCommandParameters(def controller) {
  def validParams = OutletFactory.validOutletParams
  validParams.remove('mobileMoneyAccountNumber')
  validParams.remove('accountType')
  validParams.put('merchant.id', merchant.id)
  controller.params.putAll(validParams)
}

def "created outlet insert command should have new account"() {
  given:
    setValidOutletInsertCommandParameters(controller)
  when:
    controller.save()
  then:
   outletInsertCommandHasNewAccount()
}

private boolean outletInsertCommandHasNewAccount() {
  def savedCommand = OutletInsertCommand.get(1)
  savedCommand.mobileMoneyAccountNumber == '1000000000000' &&

savedCommand.accountType == CyclosAccountType.NOT_AGENT
}

def "should not create outlet, when creating outlet insert command"() {
  given:
    setValidOutletInsertCommandParameters(controller)
  when:
    controller.save()
  then:
    Outlet.count() == 0
    OutletInsertCommand.count() == 1
}

def "should notify acceptors when creating outlet insert command"() {
  given:
    setValidOutletInsertCommandParameters(controller)
  when:
    controller.save()
  then:
    1 * commandNotificationServiceMock.notifyAccepters(_)
}
```

Now the test breaks only when it has to, we get the correct feedback, and we are free to do whatever else we want. Here’s the trick to remember

Trick 3: Verify only one right thing at a time

Now for another horrible example.

```
@Unroll("test merchant constraints field #field for #error")
def "test merchant all constraints"() {
  when:
    def obj = new Merchant((field): val)

  then:
    validateConstraints(obj, field, error)

  where:
    field                       | val                                      | error
    'name'                      | null                                     | 'nullable'
    'name'                      | ''                                       | 'blank'
    'name'                      | 'ABC'                                    | 'valid'
    'contactInfo'               | null                                     | 'nullable'
    'contactInfo'               | new ContactInfo()                        | 'validator'
    'contactInfo'               | ContactInfoFactory.createContactInfo()   | 'valid'
    'businessSegment'           | null                                     | 'nullable'
    'businessSegment'           | new MerchantBusinessSegment()            | 'valid'
    'finacleAccountNumber'      | null                                     | 'nullable'
    'finacleAccountNumber'      | ''                                       | 'blank'
    'finacleAccountNumber'      | 'ABC'                                    | 'valid'
    'principalContactPerson'    | null                                     | 'nullable'
    'principalContactPerson'    | ''                                       | 'blank'
    'principalContactPerson'    | 'ABC'                                    | 'valid'
    'principalContactInfo'      | null                                     | 'nullable'
    'principalContactInfo'      | new ContactInfo()                        | 'validator'
    'principalContactInfo'      | ContactInfoFactory.createContactInfo()   | 'valid'
    'feeCalculator'             | null                                     | 'nullable'
    'feeCalculator'             | new FixedFeeCalculator(value: 0)         | 'valid'
    'chain'                     | null                                     | 'nullable'
    'chain'                     | new Chain()                              | 'valid'
    'customerWhiteListEnable'   | null                                     | 'nullable'
    'customerWhiteListEnable'   | true                                     | 'valid'
    'enabled'                   | null                                     | 'nullable'
    'enabled'                   | true                                     | 'valid'
}
```

Do you understand what's going on? If you haven't seen it before, you may very well not. The “where” part, is a beautiful Spock solution for parameterized tests. The headers of those columns are the names of variables, used BEFORE, in the first line. It's sort of a declaration after the usage. The test is going to be fired many times, once for each line in the “where” part. And it's all possible thanks to Groovy's Abstract Syntaxt Tree Transofrmation. We are talking about interpreting and changing the code during the compilation. Cool stuff.

So what this test is doing?

Nothing.

Let me show you the code under test.

```
static constraints = {
  name(blank: false)
  contactInfo(nullable: false, validator: { it?.validate() })
  businessSegment(nullable: false)
  finacleAccountNumber(blank: false)
  principalContactPerson(blank: false)
  principalContactInfo(nullable: false, validator: { it?.validate() })
  feeCalculator(nullable: false)
  customerWhiteListEnable(nullable: false)
}
```

This static closure, is telling Grails, what kind of validation we expect on the object and database level. In Java, these would most probably be annotations.

And you do not test annotations. You also do not test static fields. Or closures without any sensible code, without any behavior. And you don't test whether the framework below (Grails/GORM in here) works the way it works.

Oh, you may test that for the first time you are using it. Just because you want to know how and if it works. You want to be safe, after all. But then, you should probably delete that test, and for sure, not repeat it for every single domain class out there.

This test doesn't even verify that, by the way. It's a unit test, working on a mock of a database. It's not testing the real GORM (Groovy Object-Relational Mapping, an adapter on top of Hibernate). It's testing the mock of the real GORM.

So if TDD gives us safety, design and feedback, what does this test provide? Absolutely nothing. So why do we put it here? Because our brain says: tests are good. More tests are better.

Well, I've got news for you. Every single test which does not provide us safety and good design is bad. Period. Those which provide only feedback, may be thrown away the moment you stop refactoring your code under the test. Assuming you ever stop. I tend to use the Boy-Scout Rule ("Always leave the code you visit cleaner than you found it"), so I never stop refactoring. I just do it "by the way".

So here's my lesson number four:

Trick 4: Provide safety and good design, or be gone

That was the example of things gone wrong. What should we do about it?

The answer: delete it.

But I yet have to see a programmer who removes his tests. Even so shitty as this one. We feel very personal about our code, I guess. I definitely do. So in case you are hesitating, let me remind you what Kent Beck wrote in his book about TDD:
The first criterion for your tests is confidence. Never delete a test if it reduces your confidence in the behavior of the system.
The second criterion is communication. If you have two tests that exercise the same path through the code, but they speak to different scenarios for a readers, leave them alone. [1]

Now you know, it's safe to delete it.

Units of measurement

Imagine this: you have a booking system for a small conference room in a small company. By some strange reason, it has to deal with off-line booking. People post their booking requests to some front-end, and once a week you get a text file with working hours of the company, and all the bookings (for what day, for how long, by whom, submitted at what point it time) in random order. Your system should produce a calendar for the room, according to some business rules (first come, first served, only in office business hours, that sort of things).

As part of the analysis, we have a clearly defined input data, and expected outcomes, with examples. Beautiful case for TDD, really. Something that sadly never happens in the real life.

Our sample test data looks like this:

```
class TestData {
   static final String INPUT_FIRST_LINE = "0900 1730\n";
   static final String FIRST_BOOKING    = "2011-03-17 10:17:06 EMP001\n" +
                                "2011-03-21 09:00 2\n";
   static final String SECOND_BOOKING   = "2011-03-16 12:34:56 EMP002\n" +
                                "2011-03-21 09:00 2\n";
   static final String THIRD_BOOKING    = "2011-03-16 09:28:23 EMP003\n" +
                                "2011-03-22 14:00 2\n";
   static final String FOURTH_BOOKING   = "2011-03-17 10:17:06 EMP004\n" +
                                "2011-03-22 16:00 1\n";
   static final String FIFTH_BOOKING    = "2011-03-15 17:29:12 EMP005\n" +
                                "2011-03-21 16:00 3";

   static final String INPUT_BOOKING_LINES =
                                 FIRST_BOOKING +
                                 SECOND_BOOKING +
                                 THIRD_BOOKING +
                                 FOURTH_BOOKING +
                                 FIFTH_BOOKING;

   static final String CORRECT_INPUT = INPUT_FIRST_LINE +
INPUT_BOOKING_LINES;

   static final String CORRECT_OUTPUT = "2011-03-21\n" +
                               "09:00 11:00 EMP002\n" +
                               "2011-03-22\n" +
                               "14:00 16:00 EMP003\n" +
                               "16:00 17:00 EMP004\n" +
                               "";
}
```

So now we start with a positive test:

```
BookingCalendarGenerator bookingCalendarGenerator =  new BookingCalendarGenerator();

@Test
```

```
public void shouldPrepareBookingCalendar() {
   //when
   String calendar = bookingCalendarGenerator.generate(TestData.CORRECT_INPUT);

   //then
   assertEquals(TestData.CORRECT_OUTPUT, calendar);
}
```

It looks like we have designed a BookingCalendarGenerator with a “generate” method. Fair enough. Lets add some more tests. Tests for the business rules. We get something like this:

```
  @Test
   public void noPartOfMeetingMayFallOutsideOfficeHours() {
      //given
      String tooEarlyBooking = "2011-03-16 12:34:56 EMP002\n" +
                    "2011-03-21 06:00 2\n";

      String tooLateBooking = "2011-03-16 12:34:56 EMP002\n" +
                    "2011-03-21 20:00 2\n";

      //when
      String calendar = bookingCalendarGenerator.generate(TestData.INPUT_FIRST_LINE +
tooEarlyBooking + tooLateBooking);

      //then
      assertTrue(calendar.isEmpty());
   }

   @Test
   public void meetingsMayNotOverlap() {
      //given
      String firstMeeting = "2011-03-10 12:34:56 EMP002\n" +
                    "2011-03-21 16:00 1\n";

      String secondMeeting = "2011-03-16 12:34:56 EMP002\n" +
                    "2011-03-21 15:00 2\n";

      //when
      String calendar = bookingCalendarGenerator.generate(TestData.INPUT_FIRST_LINE +
firstMeeting + secondMeeting);

      //then
      assertEquals("2011-03-21\n" +
              "16:00 17:00 EMP002\n", calendar);
   }

   @Test
   public void bookingsMustBeProcessedInSubmitOrder() {
      //given
      String firstMeeting = "2011-03-17 12:34:56 EMP002\n" +
                    "2011-03-21 16:00 1\n";
```

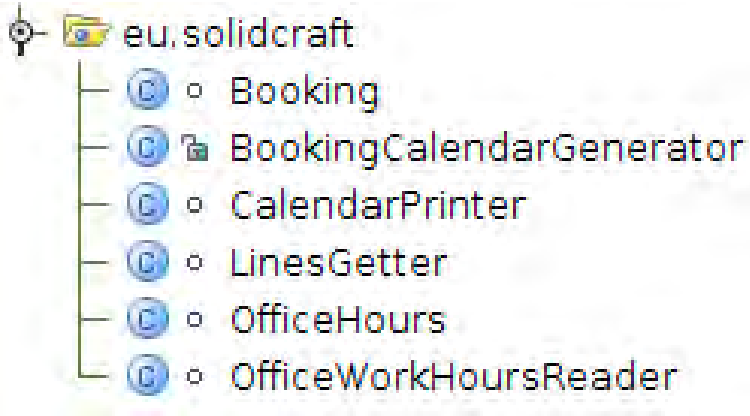
```
        String secondMeeting = "2011-03-16 12:34:56 EMP002\n" +
                "2011-03-21 15:00 2\n";

        //when
        String calendar = bookingCalendarGenerator.generate(TestData.INPUT_FIRST_LINE +
firstMeeting + secondMeeting);

        //then
        assertEquals("2011-03-21\n15:00 17:00 EMP002\n", calendar);
    }

    @Test
    public void orderingOfBookingSubmissionShouldNotAffectOutcome() {
        //given
        List<String> shuffledBookings = newArrayList(TestData.FIRST_BOOKING,
TestData.SECOND_BOOKING,
                TestData.THIRD_BOOKING, TestData.FOURTH_BOOKING,
TestData.FIFTH_BOOKING);
        shuffle(shuffledBookings);
        String inputBookingLines = Joiner.on("\n").join(shuffledBookings);

        //when
        String calendar = bookingCalendarGenerator.generate(TestData.INPUT_FIRST_LINE +
inputBookingLines);

        //then
        assertEquals(TestData.CORRECT_OUTPUT, calendar);
    }
```

That's pretty much all. But what if we get some rubbish as the input. Or if we get an empty string? Let's design for that:

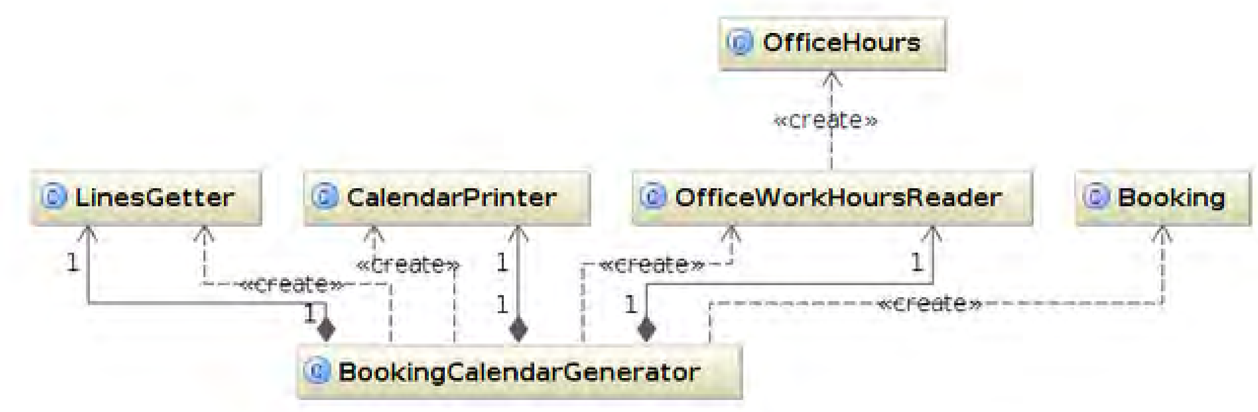
```
   @Test(expected = IllegalArgumentException.class)
    public void rubbishInputDataShouldEndWithException() {
        //when
        String calendar = bookingCalendarGenerator.generate("rubbish");

        //then exception is thrown
    }

    @Test(expected = IllegalArgumentException.class)
    public void emptyInputDataShouldEndWithException() {
        //when
        String calendar = bookingCalendarGenerator.generate("");

        //then exception is thrown
    }
```

IllegalArgumentException is fair enough. We don't need to handle it in a more fancy way. We are done for now. Let's finally write the class under the test: BookingCalendarGenerator.

And so we do. And it comes out, that the whole thing is a little big for a single method. So we use the power of Extract Method pattern. We group code fragments into different methods. We group methods and data those operate on, into classes. We use the power of Object Oriented programming, we use Single Responsibility Principle, we use composition (or decomposition, to be precise) and we end up with a package like this:



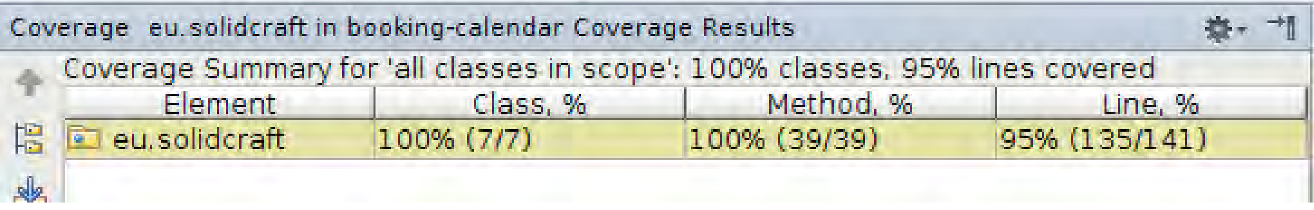
We have one public class, and several package-scope classes. Those package scope classes clearly belong to the public one. Here's a class diagram for clarity:



Those aren't stupid data-objects. Those are full-fledged classes. With behavior, responsibility, encapsulation. And here's a thing that may come to our Test Driven minds: we have no tests for those classes. We have only for the public class. That's bad, right? Having no tests must be bad. Very bad. Right?

Wrong.

We do have tests. We fire up our code coverage tool and we see: 100% methods and classes. 95% lines. Not bad.


Coverage eu.solidcraft in booking-calendar Coverage Results

Coverage Summary for 'all classes in scope': 100% classes, 95% lines covered

Element	Class, %	Method, %	Line, %
eu.solidcraft	100% (7/7)	100% (39/39)	95% (135/141)

But we have only a single unit test class. Is that good?

Well, let me put some emphasis, to point the answer out:

Trick 5: It's a UNIT test. It's called a UNIT test for a reason!

The unit does not have to be a single class. The unit does not have to be a single package. The unit is up to you to decide. It's a general name, because your sanity, your common sense, should tell you where to stop.

So we have six classes as a unit, what's the big deal? How about if somebody wants to use one of those classes, apart from the rest. He would have no tests for it, right?

Wrong. Those classes are package-scope, apart from the one that's actually called in the test. This package-scope thing tells you: “Back off. Don't touch me, I belong to this package. Don't try to use me separately, I was design to be here!”.

So yeah, if a programmer takes one of those out, or makes it public, he would probably know, that all the guarantees are voided. Write your own tests, man.

How about if somebody wants to add some behavior to one of those classes, I've been asked. How would he know he's not breaking something?

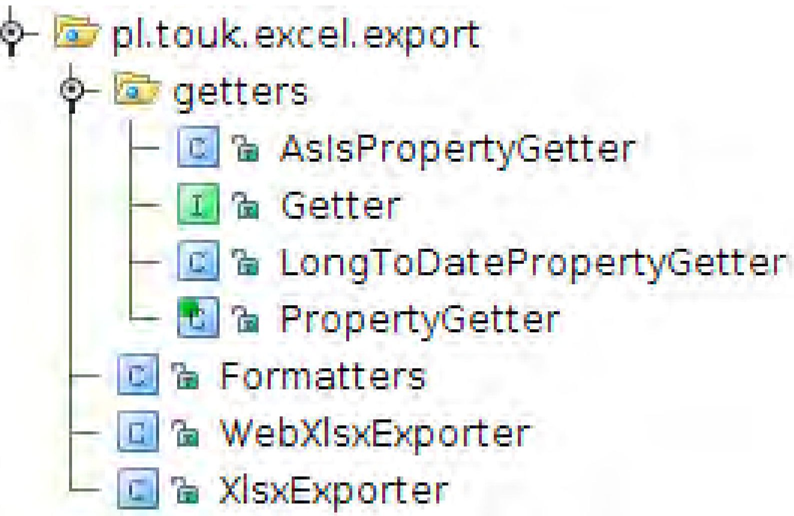
Well, he would start with a test, right? It's TDD, right? If you have a change of requirements, you code this change as a test, and then, and only then, you start messing with the code. So you are safe and secure.

I see people writing test-per-class blindly, without giving any thought to it, and it makes me cry. I do a lot of pair-programming lately, and you know what I've found? Java programmers in general do not use package-scope. Java programmers in general do not know, that protected means: for me, all my descendants, and EVERYONE in the same package. That's right, protected is more than package-scope, not less a single bit. So if Java programmers do not know what a package-scope really is, and that's, contrary to Groovy, is the default, how could they understand what a Unit is?

AdminiTrack Issue & Defect Tracking - Click on ad to reach advertiser web site

How high can I get?

Now here's an interesting thought: if we can have a single test for a package, we could have a single test for a package tree. You know, something like this:



We all know that packages in Java are not really tree-like, that the only thing those have with the directory structure is by a very old convention, and we know that the directory structure is there only to solve the collision-of-names problem, but nevertheless, we tend to use packages, like if the name.after.the.dot had some meaning. Like if we could hide one package inside another. Or build layers of lasagne with them.

So is it O.K. to have a single test class for a tree of packages?

Yes it is.

But if so, where is the end to that? Can we go all the way up in the package tree, to the entry point of our application? Those... those would be integration tests, or functional tests, perhaps. Could we do that? Would that be good?

The answer is: it would. In a perfect world, it would be just fine. In our shitty, hanging-on-the-edge-of-a-knife, world, it would be insane. Why? Because functional, end-to-end test are slow. So slow. So horribly slow, that it makes you want to throw them away and go some place where you would not have to be always waiting for something. A place of total creativity, constant feedback, and lightning fast safety.

And you're back to unit testing.

There are even some more reasons. The first one is that it's hard to test all flows of the application, testing it end-to-end. You should probably do that for all the major flows, but what about errors, bad connections, all those tricky logic parts that may throw up at one point or another. No, sometimes it would be just too hard, to set up the environment for integration test like that, so you end up testing it with unit tests anyway.

The second reason is that though functional tests do not pour concrete over your code, do not inhibit your creativity by repeating you algorithm in the test case, they also give no safety for refactoring. When you had a package with a single public class, it was quite obvious what someone can safely do, and what he cannot. When you have something enclosed in a library, or

a plugin, it's still obvious. But if you have thousands of public classes, and you are implementing a new feature, you are probably going to use some of them, and you would like to know that they are fine.

So, no, in our world, it doesn't make sense to go with functional tests only. Sorry. But it also doesn't make sense to create a test per class.

It's called the UNIT test, for a reason.

Use that.

Conclusion

Test Driven Development is easy to learn, hard to master. Like with all powerful tools, one can do a lot of harm, when using it improperly. And with a tool so simple to use, that happens more often than not. It’s when you hear people cursing and blaming TDD for being slow and unhelpful.

TDD is just a method. It’s a way, not the destination. Whenever you seem to have a problem with TDD, or tests in general, it really helps, to recall what the goal is. Whenever someone gets angry at tests in a project, whenever people try to rediscover the wheel by using integration or functional test only, it’s usually because they used TDD in a very wrong way.

My goal is to create cool software in an easy and pleasant way. I love my job. There are times, when I hate it as well. I’ve found that proper TDD helps me reduce the number of times, when I hate it. But improper TDD has exactly the opposite effect. What works in one context, may not work in another. What works for one person, may not have the same effect on somebody else. It’s crucial to be pragmatic and reflect on the methods and tools, instead of using them blindly. Drugs, guns and replicants [4] can be pretty dangerous, when used improperly. Let’s share the best practices for TDD, before they make it illegal.

References

1. Kent Beck, Test Driven Development: By Example, Addison-Wesley Longman, 2002

2. Eric Evans, Domain-Driven Design- Tackling Complexity in the Heart of Software, Addison-Wesley, 2004

3. Abandon all hope, ye who enter here - Divine Comedy, Dante Alighieri

4. Blade Runner

Data Management Tools for Embedded Application Development

Sasan Montaseri, ITTIA, http://www.ittia.com/

When it comes to embedded systems and mobile devices, developers face a choice between flat file and embedded database, as any future change can be very risky and expensive for their application, in a demanding market where shortcomings can result in failure.

For stand-alone applications that store little data, custom flat file formats are a straightforward method to save information. Plain text files use a human-readable format, such as INI, CSV, XML, or JSON that is parsed into application-specific data structures. Similarly, flat binary files are created by writing those data structures directly to disk. However, the simplicity of flat files come with a number of limitations, such as:

- Although easy to write and create, even the slightest change requires the entire file to be rewritten, which in turn makes files susceptible to data loss.
- The extra wear caused by repeated erasing and writing of a large text file to flash memory will eventually wear out and thus reduce the total lifetime of flash media.
- Since the entire file must be read into memory to efficiently search for data, they cannot be very large because of the time required to write changes.
- Flat files also have a limited data management life cycle offering, as devices and embedded systems are becoming more aware of each other and other systems.

Even with binary files, some common problems are:

- Binary files can be divided into smaller pieces used independently, but special tools must be developed to use these files outside the application.
- Binary files provide some protection against corruption of the entire file, but can still lose data when changes are only partially written before an unexpected power failure.
- Binary files make it more difficult to store variable-width data.
- If data is copied directly from memory to a binary flat file, it is not easy to open the file on another architecture.

The need to store data in a software library takes developers of embedded systems and mobile devices to choose an embedded database. Databases can be implemented in many different ways, and the choice of algorithms in a particular database product has a profound impact on performance characteristics and what features are available. Areas most impacted by the implementation of the database include:

- Frequency of costly disk or flash media I/O operations.
- Time required to recover from a crash or power loss.
- Ability to manage large amounts of data without severe performance degradation.
- Performance impact of sharing data between tasks and other applications.
- Relative performance of read and write operations.
- Portability of the storage format and application code.
- Effort required to integrate database technology into the application.

All embedded databases serve three main purposes for data management:

- Reliable storage
- Efficient queries
- Safe shared access

Balancing these three areas of functionality is difficult because strong guarantees in one area can weaken the capabilities of the other two areas.

SQLite is an example of open source database available on the market. We will now study some of the pros and cons SQLite as a database:

1. **Atomic Transactions:** SQLite supports atomic commit, but creates a rollback journal for each transaction that remembers the original value before each change is made. So when SQLite commits a transaction, it must write all modified pages to disk in their entirety and then wait for the operation to finish. This is extremely costly, but necessary to support recovery with only a rollback journal.
2. **Concurrency and Shared Access:** SQLite uses the locking mechanism built-in to the file system to protect database files during modification, which allows any process on the device with access to the file to use the database. This approach relies on the stability of the file system locking framework, and therefore cannot be used with files that are shared over a network.
3. **Data Typing and Type Safety:** SQLite uses dynamic run-time typing, which means that data types are only checked when a value is read from the database and used. This is useful in prototyping, before the application's requirements are fully formed, because it allows data to be written to the database without much regard for how it will be used later. Production code, however, must be carefully audited to ensure that type mismatches are not possible or can be dealt with in a reasonable way.
4. **In-Memory Storage:** While SQLite supports memory databases, they store data in the same format that is used for disk tables. SQLite memory databases are private to each connection, and cannot be shared between different tasks.
5. **High Availability:** The only high availability offered by SQLite is a limited backup solution. SQLite only provides basic recovery logging and limited backup, which are important tools for self- stabilization but hardly a complete solution.
6. **Replication:** SQLite provides no replication functionality.
7. **Device Notification:** SQLite applications can register callback functions to be notified when a row is changed. However, this feature only monitors changes made by the current connection, and cannot be used to monitor other connections to the database.

To overcome SQLite's shortcomings, developers of embedded systems and mobile devices should look for a database offering the following features:

1. **Performance:** Whether database access is limited to one connection at a time, or divided between several concurrent tasks, it must show a significant performance advantage for database writes.
2. **Low-level access to the engine:** Each embedded application has unique requirements for data management and storage. Selecting the right tools requires a careful problem analysis and a thorough understanding of database technology. Using a technology with the right features makes a significant impact on the performance, maintainability, and extensibility of the application.
3. **Concurrency:** Database should greatly simplify data management for applications on embedded systems and devices, but without the complexity of a back-end database server.

4. **Replication:** Database should support built-in replication that duplicates changes across multiple databases, even if connectivity is not always available.
5. **ANSI Standards:** Database should follow the ANSI SQL standards.
6. **Data typing and type safety:** Database should offer greater protection with static typing rather than the manifest typing in SQLite.
7. **True in-memory storage engine:** Database should use algorithms that are optimized for both read and write performance in RAM.
8. **High availability:** The combination of built-in replication, client/server communications, online backup, and high-performance recovery should give developers a complete set of tools to achieve fault tolerance through both self-stabilization and duplication.

A feature-level comparison between open source and proprietary databases cannot be done accurately, as there is too much variation within each camp. However, the following table shows how a database combines the best qualities of both plain text and binary file formats, while also overcoming several important limitations:

	Plain Text Files	**Flat Binary Files**	**Databases**
Human-readable format	Yes	No	No
Convenient for configuration files, any editor can view and save a plain text file.			
Compatible with existing tools and APIs	Yes	No	Yes
Standard tools make it easy to convert to a spreadsheet or generate reports.			
Platform-independence	Yes	No	Yes
Variations in data structure alignment and padding tie flat binary formats to one platform.			
Variable-width fields	Yes	No	Yes
Compact storage formats rely on dynamic buffers, not fixed-width structures.			
Model hierarchical data structures	Yes	No	Yes
Nodes in a tree map readily to XML elements and database tables alike.			
Minimum overhead	No	Yes	No
Records in a binary file are copied directly between memory and disk.			
Efficient write throughput	No	Yes	Yes
In-place updates avoid rewriting an entire file after updating or deleting a record.			
Model relational data structures	No	No	Yes
Organize records with multiple unique and primary keys.			
Transaction logging	No	No	Yes
Crash recovery protects data from unexpected power loss and similar failures.			
Isolated shared access	No	No	Yes
Concurrent tasks must coordinate to access data safely.			
Scalable sorting and searching	No	No	Yes
To find records quickly, indexes must be continuously maintained over key fields.			
Standard query language (SQL)	No	No	Yes
Express complex queries with little or no application code.			
High availability	No	No	Yes
Replication and online backup mitigate media and communication failure risks.			

Final Thought

Relational embedded databases offer solution to those typical problems. When you embed your application with an open source database like SQLite, you decide to become a database company and manage your own updates, upgrades, and patches. Choosing a proprietary database connects you to a dedicated team that will be available to take care of your needs from accountability, assistance in managing the danger of intellectual property and legal risks, and prediction of uncertainties. This helps application development to be simplified, and rather than building separate solutions for each of these problems, the application developer uses a standard framework for accessing and modifying data. Furthermore, reconciling these features with each other in a single application is not trivial and can become a great distraction to developing the business logic of the device.

Database Software Development Videos and Tutorials- Click on ad to reach advertiser web site

Improve your knowledge of the current trends in data management and database systems: Data Modeling, MySQL, Oracle, PostgreSQL, SQL Server, NoSQL, Object Relational Mapping

DatabaseTube.com

JSHint—a JavaScript Code Quality Tool

Anton Kovalyov, Mozilla, http://anton.kovalyov.net

JSHint is a static analysis tool written in JavaScript that helps developers detect potential bugs in their JavaScript code and enforce their development team's JavaScript coding conventions. JSHint scans your program's source code and reports about commonly made mistakes and potential bugs. The potential problem could be a syntax error, a bug due to implicit type conversion, a leaking variable, or any of the other 150+ problems that JSHint looks for.

JSHint is flexible enough so you can adjust it to your particular coding style. This flexibility doesn't prevent JSHint from spotting many mistakes and potential problems in your code, before you deploy them live.

Since JSHint is written in JavaScript, it can be used as a web application, command-line tool or a Node.JS module.

Web Site: http://jshint.com/
Version Tested: JSHint 0.9.1 (r12)
System Requirements: Any environment that can run modern JavaScript (ES5): browsers, Node.JS, Rhino, Windows Script Host and so on.
License and Pricing: Open Source
Support: Project page on GitHub (https://github.com/jshint/jshint/) and user mailing list (http://groups.google.com/group/jshint).

Why do we need static code analysis tools?

Any code base eventually becomes huge at some point, and simple mistakes - that would not show themselves when written - can become showstoppers and waste hours of debugging. This problem is particularly severe in JavaScript since its rushed development (JavaScript was created under an extremely tight deadline. Brendan Eich - a developer at Netscape - had only **ten days** to write the language) allowed for quite a few pitfalls to creep into the language. For example, consider the following snippet:

```
function add(x, y) {
        return
        x + y;
}

add(1, 2);
```

A programmer familiar with any other C-like language will probably expect the result of this expression to be 3 and will be very surprised to learn that the actual result here is *undefined*. This happens because the rules behind statement termination are not as simple in JavaScript as they should be. In this case, your JavaScript engine will automatically insert a semicolon after *return* because it thinks that *return* and *x + y* are two separate statements. So, basically, your JavaScript engine sees your program like this:

```
function add(x, y) {
        return;
        x + y;
}
```

Another common source of bugs is browser incompatibility. Most web developers use only modern browsers so it's easy for them to unknowingly introduce a bug like the one that's in this code snippet:

```
var episodes = [
	"An Unearthly Child",
	"Blink",
	"Angels Take Manhattan",
];
```

This code is a perfectly valid JavaScript according to the ECMAScript 5 (ECMAScript is a formalized version of JavaScript. There were a few reasons behind its name and why is it different from JavaScript but they are irrelevant for this discussion. All our readers need to know right now is that ECMAScript 5 is the latest standard for the JavaScript programming language.) specification and it will work just fine in all modern browsers. However, legacy browsers - such as Internet Explorer 6 or 7 - will fail to parse a trailing comma before the ending bracket and throw a syntax error. And unless you run all your tests in all supported browsers, this bug will most likely slip through the cracks and into the production. There is at least one case of a startup whose launch was almost ruined by this bug!

This is just a couple examples out of dozens different pitfalls that await anyone who is doing any kind of serious JavaScript development. This is where static code analysis tools come into play and help developers to spot these and similar problems. But before we dive into more examples, let's look at how to install and configure JSHint.

Installation

The simplest way to use JSHint is through your browser. Just go to http://jshint.com/, paste your JavaScript code and press *Lint.* JSHint will scan your code and generate a report on the same page. See Figure 1.

Using the website works well for smaller chunks of code but it becomes increasingly annoying to be constantly copying and pasting your code. In addition to that, it's impossible to automate your "linting" process and integrate JSHint into your development workflow through the website.

That's why we have a command-line version of JSHint, distributed as a Node.JS module. Assuming that you already have Node.JS installed all you need to do is run the following command:

```
$ npm install jshint -g
```

Once it finishes installing the package you will have a *jshint* program available to you:

```
$ jshint myfile.js

myfile.js: line 2, col 9, Missing semicolon.
myfile.js: line 5, col 6, Extra comma.
```

If you point JSHint at a directory it will scan and check all JavaScript files inside of it.

The JSHint website and our Node.JS package are not the only ways to use JSHint. You can also use JSHint as a plugin for your editor to lint your code while typing. Check out http://jshint.com/platforms/ to see if there's already a plugin for your favorite editor/IDE, and if not, consider writing a plugin yourself.

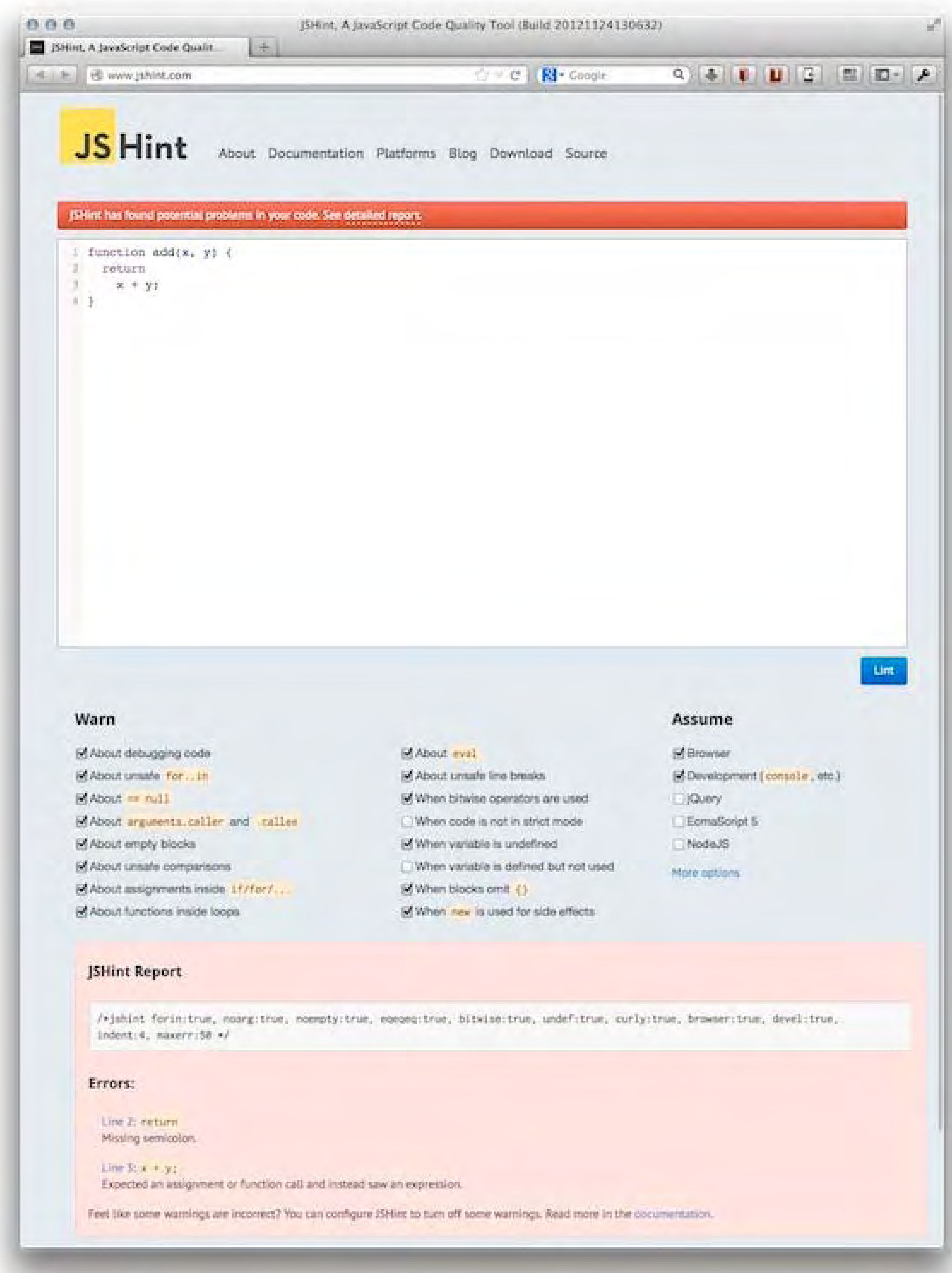


Figure 1. JSHint.com Report

Configuration

JavaScript is a flexible language, and there isn't one set of coding conventions that everyone follows. For that reason, JSHint is highly configurable, so that you can enable or disable options to suit you and your team's needs. For example, if you want JSHint to report about any variable that was declared but never used you will need to enable the *unused* option. If you want to make sure that all your code always runs in strict mode, you will need to enable the *strict* option.

There are two ways to set JSHint options: you can either create a configuration file or use special JavaScript comments. Let's start with a configuration file.

Before JSHint starts parsing your files it looks for a special file named *.jshintrc*. First, it looks in the current working directory and then goes all the way up until it finds the file or hits the root. Then, if the file hasn't been found yet, JSHint looks for it in your home directory.

The structure of *.jshintrc* is simple: it is a JSON file with option names as keys. For example, the following file will enable two options: *strict* and *unused*.

```
$ cat ~/.jshintrc
{
        "strict": true,
        "unused": true
}
```

Some options, such as *undef*, need to know whether you're using global variables defined somewhere outside of the current file. The globals can be provided to JSHint via the special *globals* option in the configuration file. For example, the following snippet will tell JSHint that, in addition to built-in JavaScript globals, you're also using a global variable named *MyLib*. And because its value is *false* JSHint will warn if you try to overwrite it in your file.

```
$ cat ~/.jshintrc
{
        "strict": true,
        "unused": true.
        "globals": {
        "MyLib": false
        }
}
```

Sometimes, however, you might want to set options per-file. For example, let's say you have a legacy file that doesn't adhere to your project guidelines but you still want to lint it. You can either extract this file into a separate directory and create a specialized *.jshintrc* next to it or you can use special comments to configure JSHint just for this file:

```
/*jshint expr: true, newcap: false */
/*global MyLegacyLib: false */
```

As you can see there are two type of JSHint comments. The first one, /*jshint, allows you to set options just like we did above in the *.jshintrc* file. And the second one, /*global, allows you to tell JSHint about a global variable *MyLegacyLib*.

And this is how you can configure JSHint to your own taste. You can find a complete list of options, together with short descriptions, on http://jshint.com/docs.

Leaking variables

Now that we know how to install and configure JSHint, let's go over a simple usage example. This example was simplified to not to bore you with long code listings but imagine that it is a part of a file with 1000 lines of code.

```
$.post("ajax/test", function (data) {
	var height = 0;
	var parsed = JSON.parse(data);
	if (parsed.height) {
	heigth = parsed.height;
	}
	changeHeight(height);
});
```

The code above has a rather small typo that can lead to very nasty bugs. On line 6 we accidentally misspelled *height* as *heigth* and by doing so we created a new variable *heigth*. Because that assignment didn't start with *var*, JavaScript puts that new variable in the global scope, and we've now leaked what was meant to be a harmless local variable. This example is simple but in a big enough project, such leaks can result in either unexpected behavior or increasing memory consumption of your application. Fortunately, JSHint was designed to catch such mistakes.

We will start by enabling an option *undef* which tells JSHint to check for variables that were used but never defined. Adding the following line to the beginning of the file will do the trick:

```
/*jshint undef:true */
```

Now if you run JSHint over your file it'll produce two warnings:

```
myfile.js: line 1, col 1, '$' is not defined.
myfile.js: line 6, col 9, 'heigth' is not defined.
```

An astute reader can tell that only one of these warnings is valid. Variable *$* might not be defined in this file but it is defined elsewhere in our project so let's inform JSHint about that by adding another comment:

```
/*global $:false */
```

Now we have only one warning and it is about *heigth* being undefined. We found our leak, can very easily fix it and verify the fix by running JSHint over that file again.

Summary

Now you know how to install and configure JSHint. You also learned how to use the tool to find leaking variables. But finding leaks is just one possible use of JSHint - and there are many more. Go over our documentation and find out how JSHint can help you in spending less time looking for missed semicolons and forgotten *var* keywords, and more time writing code.

Also, JSHint is written in JavaScript so it can be used almost anywhere. It comes with a safe set of default options so more novice programmers don't have to start by configuring it. And more experienced programmers can fine-tune JSHint to their own taste.

Documentation and Literature

The general documentation of JSHint is available on the project homepage: http://jshint.com/. And if you want to use JSHint as a JavaScript module, more technical documentation is available on the projects GitHub page: https://github.com/jshint/jshint/.

There is also a great book "Maintainable JavaScript" by Nicholas Zakas that teaches how to write maintainable code and use tools such as JSHint to keep your team on track: http://shop.oreilly.com/product/0636920025245.do

No More Mocks in NoSQL Applications with NoSQLUnit

Alex Soto, http://www.lordofthejars.com/

NoSQLUnit is an open source JUnit extension for writing tests of Java applications that use NoSQL databases. The goal of NoSQLUnit is to manage the lifecycle of NoSQL engines. It helps you to maintain the databases under test into known state and standardize the way we write tests for NoSQL applications. Current version supports the following engines: MongoDB, Cassandra, HBase, Redis and Neo4j.

Web Site: https://github.com/lordofthejars/nosql-unit
Version Tested: NoSQLUnit 0.7.0 on Linux Ubuntu 12.04
License & Pricing: Open Source (Apache License 2.0)
Support: https://github.com/lordofthejars/nosql-unit/issues

Overview

Software tests in general and unit tests in particular should follow the FIRST rules:

- Fast. Tests should be fast to execute. If they are fast you will run them more often.
- Isolated. The execution of one test should not condition the result of another test.
- Repeatable. Without any change, your tests should always return the same result.
- Self-validated. Tests should be treated as production code. Among other things, this implies that the tests should be written in a clean way. Anyone should understand quickly and exactly the purpose of that test.
- Timely. Test should be written before coding the production code.

There is an exception for high level tests (integration tests, acceptance tests, smoke tests, …) where tests can be slow because of external dependencies with database servers, filesystem access or network communication.

The isolation rule is the most broken one when people write tests for the persistence layer. Let's suppose that we have a piece of code that contains two methods: one that inserts an item into database and another one that counts the number of items in the database. These functions would have at least two unit tests, one for each method. This is the problem: the count test would return different result depending on the order of the tests execution. Clearly these tests are not running in a free environment and are not isolated. DBUnit solves this problem for relational database and NoSQLUnit does the same for NoSQL systems.

NoSQLUnit

NoSQLUnit is a JUnit extension that help you to:

- manage lifecycle of NoSQL servers.
- maintain the database under test into known state, fixing the isolation problem.
- standardize the way we write tests for NoSQL databases.

NoSQLUnit is composed by two groups of JUnit Rules and two annotations.

The *first group* starts and stops the database servers used within the tests. At least each supported database implements one rule of this kind, but it can contains more than one depending on start up modes provided by each backend. Almost all supported engines provide two ways to initialize the NoSQL system.

- Embedded mode: This mode takes care of starting and stopping the database engine inside the same JVM. The database could be " in-memory " or not, this is determined by the vendor. This mode will be typically used during the unit testing phase.
- Managed mode. This mode is in charge of starting NoSQL database from its installation directory and stopping it. This will typically be used during the integration tests.

The *next group* contains one JUnit rule for each supported engine, and can be understood as a connection to the NoSQL database. This connection will be used for maintaining the database into a controlled state.

The *first annotation* (@UsingDataSet) is used for populating database with contents of defined data set files.

The *second annotation* (@ShouldMatchDataset) is an optional annotation, which compares the content of the database to a reference data set file. This annotation is useful if asserting the database state directly from the test code may imply a huge amount of work.

Note that in both annotations, data set files are required and their format will vary according to NoSQL vendor.

Let's summarize the lifecycle of NoSQLUnit.

First of all, the required engines for executing the tests are started. Then the databases are cleaned and a test dataset is inserted into each database. Then the test is executed. An optional step verifies if the databases contain the expected data. Finally, NoSQLUnit stops the running engines.

Installation

NoSQLUnit is published to Maven Central Repository, so all you have to do is add the dependency, which will be different depending on the database engine, into your Maven, Gradle or Ivy dependencies file.

In the following example, we are going to write a test for an application that uses MongoDB as backend. Our pom file is:

```
<dependency>
    <groupId>com.lordofthejars</groupId>
    <artifactId>nosqlunit-mongodb</artifactId>
    <version>0.7.0</version>
</dependency>
```

Example

This example shows how to use NoSQLUnit when you use MongoDB as a database. This simple Java class inserts a new book into a MongoDB collection.

```
private DBCollection booksCollection;

public BookManager(DBCollection booksCollection) {
      this.booksCollection = booksCollection;
}

public void create(Book book) {
      DBObject dbObject = MONGO_DB_BOOK_CONVERTER.convert(book);
      booksCollection.insert(dbObject);
}
```

This NoSQLUnit test code verifies that the *create* method inserts a new book into database:

```
import static
com.lordofthejars.nosqlunit.mongodb.ManagedMongoDb.MongoServerRuleBuilder.new
ManagedMongoDbRule;

import static
com.lordofthejars.nosqlunit.mongodb.MongoDbRule.MongoDbRuleBuilder.newMongoDb
Rule;
public class WhenANewBookIsCreated {

      @ClassRule
      public static ManagedMongoDb managedMongoDb =
newManagedMongoDbRule().mongodPath("/opt/mongo").build(); [1]

      @Rule
      public MongoDbRule managedMongoDbRule =
newMongoDbRule().defaultManagedMongoDb("test"); [2]

      @Test
      @UsingDataSet(locations="initialData.json",
loadStrategy=LoadStrategyEnum.CLEAN_INSERT) [3]
      @ShouldMatchDataSet(location="expectedData.json") [4]
      public void book_should_be_inserted_into_repository() {


      BookManager bookManager = new
BookManager(MongoDbUtil.getCollection(Book.class.getSimpleName()));

      Book book = new Book("The Lord Of The Rings", 1299);

      bookManager.create(book); [5]

      }

}
```

This code contains all the typical phases of a NoSQLUnit test. We start by creating a managed MongoDB instance. [1]

Then we configure a connection to maintain MongoDB collections into a known state. [2] We define a connection to a managed MongoDB server using default parameters (localhost:27017) and using *test* database. This can be obviously changed by calling methods of `MongoDbConfiguration` class provided by `newMongoDbRule()` method.

After configuring the connection, it is time for setting which data is inserted in the database [3]. For this we use `@UsingDataSet`, which cleans the database and populate it with the content of the initialData.json file shown below.

```
{
	"Book":
	[
	{"title":"The Hobbit","numberOfPages":293}
	]
}
```

The first element ("*Book*") is the name of the collection, and then we simply must set the json documents belonging to that collection.

We also need a way to verify that the inserted data is what we expect [4]. We could establish a connection to server and assert this manually, but we can also use the `@ShouldMatchDataSet` annotation. And finally at [5] the code under test is called which saves a new book into MongoDB *Book* collection. See https://github.com/lordofthejars/nosql-unit/tree/master/nosqlunit-demo for a complete example.

Supported Databases

This table shows the supported engines:

Engine	Embedded Rule	Managed Rule	File Format
MongoDB	InMemoryMongoDB	ManagedMongoDB	json
Cassandra	EmbeddedCassandra	ManagedCassandra	cassandra-unit json
HBase	EmbeddedHBase	ManagedHBase	json
Redis	EmbeddedRedis	ManagedRedis	json
Neo4j	EmbeddedNeo4j	ManagedNeoServer	graphML xml
	InMemoryNeo4j	ManagedWrappingNeoServer	

Advanced Features

NoSQLUnit offers some additional features related to the NoSQL world:

- Tests can be run for applications deployed in the cloud. You should not define any lifecycle rule and configure the connection rule accordingly.
- NoSQL applications might use a polyglot persistence approach. Your shopping cart could for example be implemented using Redis and the reports can be stored into MongoDB. NoSQLUnit supports this approach by providing the `@Selective` annotation.
- Partial support to JSR-330, so underlying connection used for maintaining database into known state can be used in your test. This is very useful when an In-Memory embedded lifecycle approach is used.
- Testing replication.

Conclusion

Writing tests for persistence layer can be a really hard work. You need to create a lot of data set files and every data set file might involve the creation of a huge amount of data. Although this is not an easy task, don't let your persistence tests ruin your internal quality.

NoSQLUnit will continue its improvement by supporting more NoSQL systems and integration with Maven and Arquillian.

Although NoSQLUnit does not implement a plugin system to add support for additional engines, it is easy to add them by extending an abstract class and an interface.

Don't hesitate to ask support for any other NoSQL database or any other capability. You can open an enhancement issue at https://github.com/lordofthejars/nosql-unit/issues.

Further Readings

NoSQLUnit documentation: https://github.com/lordofthejars/nosql-unit
Cassandra site: http://cassandra.apache.org/
Cassandra-Unit site: https://github.com/jsevellec/cassandra-unit
HBase site: http://hbase.apache.org/
MongoDB site: http://www.mongodb.org/
Redis site: http://redis.io/
Neo4j site: http://neo4j.org/
NoSQL Distilled book: http://martinfowler.com/books/nosql.html

STARCANADA – Register Now with Your Remaining 2012 Budget!

Register now for the inaugural STARCANADA conference April 7-11, 2013 in Toronto, Canada! STARCANADA will offer the same up-to-date information, tools, and technologies as our other world-renowned STAR events. The full conference program is now available online so take a moment to explore what the premier conference in the software testing industry has in store. Register by January 18 to receive a $50 Amazon.com Gift Card PLUS up to $300 off your conference registration fees!

Go to http://sqe.com/go?sc13mt1204

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

METHODS & TOOLS is published by **Martinig & Associates,** Rue des Marronniers 25,
CH-1800 Vevey, Switzerland Tel. +41 21 922 13 00 Fax +41 21 921 23 53 www.martinig.ch
Editor: Franco Martinig ISSN 1661-402X
Free subscription on : http://www.methodsandtools.com/forms/submt.php
The content of this publication cannot be reproduced without prior written consent of the publisher
Copyright © 2012, Martinig & Associates