METHODS & TOOLS

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

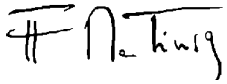
Winter 2011 (Volume 19 - number 4) www.methodsandtools.com

The Meaning of Agile Certification is Value

In their book " Practices for Scaling Lean & Agile Development", Craig Larman and Bas Vodde discuss about the quality of code and certification, mainly the CMMI approach. They refute the link between good code production and certification and wrote "Do not believe that an appraisal, rating, or certification in any process improvement model - including Scrum, agile methods and ISO certification - means much of anything, other than the ability to somehow pass an appraisal at least once." Although I usually agree with them, I would disagree on this point. Certification has its meaning. And this meaning is money. Certification has become an important business and this is why you have so many "independent" professional associations that now provide some type of certification. Just in the Agile project management world, you can be certified by the Scrum Alliance, Scrum.org or the Project Management Institute (PMI)

Do Larman and Vodde think that people would have made the efforts to build these programs if they were meaningless for them? Of course, training or coaching helps transitioning to Agile and there is value in being trained directly by great Agile minds like Jeff Sutherland or Mike Cohn. But there is also a difference between training and certification. Having followed ScrumMaster training doesn't mean that somebody will be a good ScrumMaster. When plane pilots are certified, they have to actually know how to fly and this certification is continuously monitored. The process is not based on their ability to answer once some questions about flight theory. Or at least it shouldn't. We often read the mantra about the difference between "Doing Agile" and "Being Agile". To me, certification belongs more to the "Doing Agile" side. This opinion is obviously easier to express if you are not trying to pay your bill giving Certified ScrumMaster courses. But didn't we say that Agile was about "Values"? Often if you want to understand the meaning of some professional behavior, you just need to follow the money...

Thanks for reading and supporting Methods & Tools in 2011 and I wish you all the best 2012.



Inside

How to Make Your Culture Work with Agile, Kanban & Software Craftsmanship

Michael Sahota, michael.sahota [at] agilitrix.com
Agilitrix, http://agilitrix.com/

Introduction

Culture is an often ignored, misunderstood and yet a key dynamic in company performance. In this article we will introduce a model of culture that is simple to understand and apply. The model will be used to show that Agile, Kanban and Software Craftsmanship have strong cultural biases that limit the scope of their applicability. Finally, an approach will be outlined to select approaches that work with the culture in your organization.

Culture is the #1 Challenge with Agile Adoption

The results of the VersionOne's State of Agile Development Survey [1] are astonishing. The #1 barrier to further Agile adoption at companies is *cultural change*. To further underscore the importance, this problem is reported by 51% of respondents. Even this number may be understated because cultural impacts are challenging to identify.

So, how important is company culture? Edgar Schein, a professor at MIT Sloan School of Management, says "If you do not manage culture, it manages you, and you may not even be aware of the extent to which this is happening."

Agile is not a Process - it Defines a Culture

But what does this have to do with Agile?

Well, what is Agile? The consensus definition is provided by the now 10 year-old Agile Manifesto [2]. Agile is an *idea* supported by as set of *values* and beliefs. In other words Agile defines a *target culture* for successful delivery of software. (More on Agile's cultural model later on in this article).

Agile is commonly described as a process or a family of processes. This is true, but a dangerous and incorrect abstraction. (Mea culpa, I have communicated this misleading message many times.) If Agile were just a process family, then we wouldn't be seeing culture as a prevalent problem.

Even worse, Agile is bought and sold as a *product*. Companies have problems such as too slow time to market or bad quality and want a solution. Agile benefits are touted and a project is kicked off with Agile as the solution. Dave Thomas, coined the concept of the *Agile Tooth Fairy* where Agile Coaches can swoop in and sprinkle magic dust on troubled projects to correct years of atrophy and neglect [3]. This is a myth: Agile is not a silver bullet.

Such thinking is not new. Tobias Mayer has written about how Scrum is much more about changing the way we think than it is a process [4, 5]. Bob Hartman has a great presentation on this topic - Doing Agile isn't the same as being Agile [6]. Mike Cottmeyer wrote a series of great posts on how companies are adopting Agile, not transforming to Agile [7].

Michael Spayd conducted a Culture Survey of Agile [8]. His landmark results not only show that Agile has a particular culture profile, but identified the key elements as *Collaboration* and *Cultivation*. (More on this in the next section.) Independently, Israel Gat was speaking about the relationship between Agile and culture in How we do things around here in order to succeed [9]. His observation was that Agile adoption will trigger conflict due to cultural mismatches.

In summary, to be successful, we need to start thinking about *Agile as a culture and not as a product or family of processes.*

In the next section I will introduce a model for culture that can be used to understand culture at your company. The following sections explain the unique cultures of Agile, Kanban, and Software Craftsmanship. In the last section, I provide a playbook so that you can assess how well a particular approach fits with your company's culture.

Understanding Culture through the Schneider Model

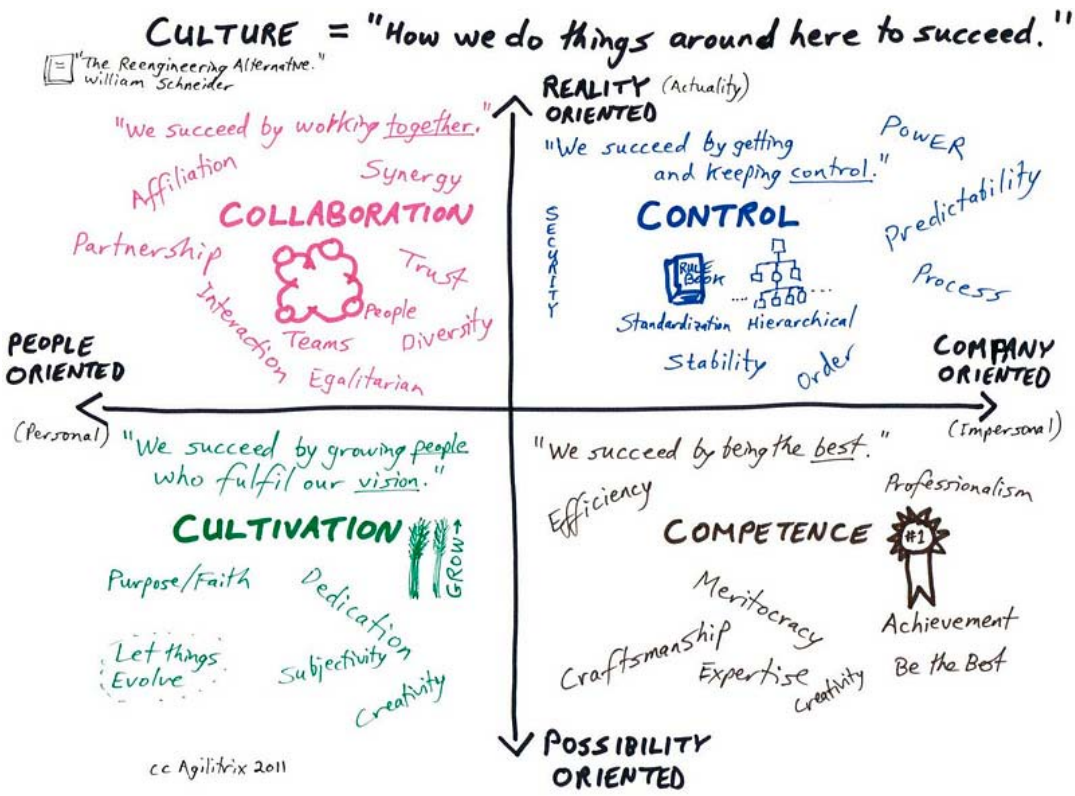
We need to define what we mean by culture before exploring Agile further. In this section, I will introduce the *Schneider Culture Model* based on William Schneider's book The Reengineering Alternative: A plan for making your current culture work [10]. Although there are many different ways of thinking about corporate culture, this model has been selected since it leads to actionable plans.

What is a culture model? A culture model tells us about the values and norms within a group or company. It identifies what is important as well as how people approach work and each other. For example, one culture may value stability and order. In this case clearly defined processes will be very import and there will be a strong expectation of conformance rather than innovation and creativity.

The Schneider Culture Model define four distinct cultures:

1. *Collaboration* culture is about working together.
2. *Control* culture is about getting and keeping control.
3. *Competence* culture is about being the best.
4. *Cultivation* culture is about learning and growing with a sense of purpose.

The diagram below summarizes the Schneider Culture Model. Each of the four cultures are depicted - one in each quadrant. Each has a name, a "descriptive quote", a picture, and some words that characterize that quadrant. Please take a moment to read through the diagram and get a sense of the model and where your company fits.



Another aspect of the Schneider model is the axes that indicates the focus of an organization:

1. Horizontal axis : People Oriented (Personal) vs. Company Oriented (Impersonal)
2. Vertical axis: Reality Oriented (Actuality) vs. Possibility Oriented

This provides a way to see relationships between the cultures. For example, Control culture is more compatible with Collaboration or Competence cultures than with Cultivation culture. In fact, Cultivation culture is the opposite of Control culture; learning and growing is opposite of security and structure. Similarly, Collaboration is the opposite of Competence. “All models are wrong, some are useful” - George Box, statistician. All models are an approximation of reality and it is important to remember that we are ignoring minor discrepancies so that we can perform analysis and have meaningful discourse. Also, we may wish to consider other models such as Spiral Dynamics [11] if we wanted to understand cultural evolution.

Key Points about Culture

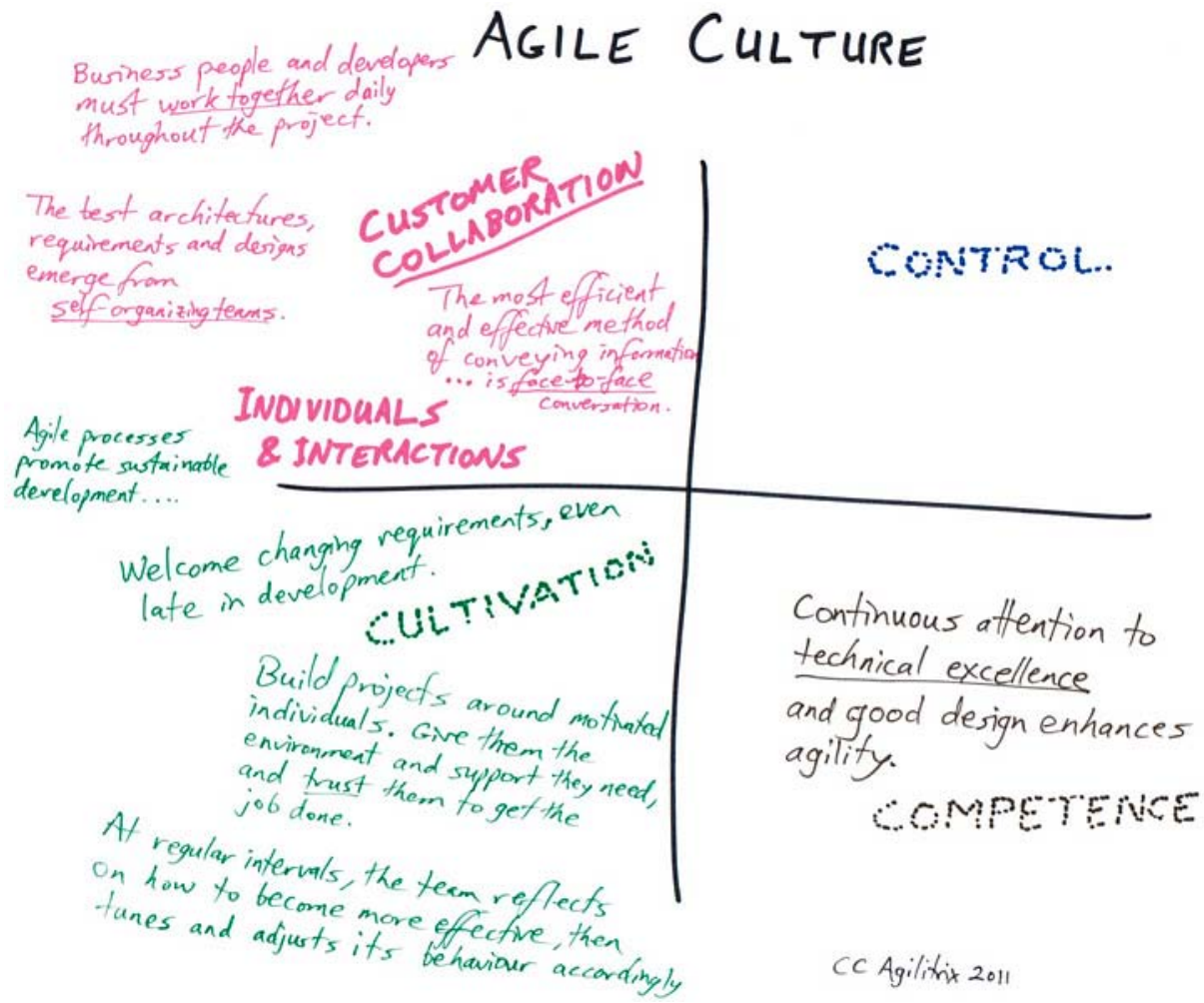
In the Schneider model, no one culture type is considered better than another. Please refer to the book for details the strengths and weaknesses of each. Depending on the type of work, one type of culture may be a better fit. Companies typically have a dominant culture with aspects from other cultures. This is fine as long as those aspects serve the dominant culture. Different departments or groups (e.g. development vs. operations) may have different cultures. Differences can lead to conflict.

Agile Culture is about Collaboration and Cultivation

As mentioned earlier, Michael Spayd has done the community a great service by undertaking a culture survey of Agilistas. The results are very striking: it shows that the two dominant cultures are *Collaboration* and *Cultivation*, with Competence a distant third and Control barely even on the map. The results suggest that Agile is all about the *people*. Interestingly, the survey included Scrum, XP, as well as Lean-Kanban folks.

The Agile Manifesto and Principles Define Agile Culture

The Agile Manifesto and twelve principles - even after ten years - are still the reference for what is considered Agile. Consider the following diagram, where the values and principles are mapped to the Schneider model.



It can be seen that there is high density of values and practices that are aligned with *Collaboration* and *Cultivation*. Note that there were no elements related to Control culture and only one related to Competence culture. This is strikingly similar to the result obtained by Michael Spayd in his survey of Agilistas.

Analysis Approach (For the Curious)

Some of you may be curious as to how I arrived at my result.

For each value or principle, I analyzed how well it was aligned with each of the cultures. If there was a strong affinity, I associated it with that culture. For example, Customer Collaboration was very easy since it identifies success through people working together. Some items seemed to be orthogonal to culture. For example, working software, didn't really seem to suggest one culture over another. Well, it may weakly suggest Competence culture, but only a bit. Other items were a best guess based on my current understanding.

These results have been partially validated through group workshops where participants performed the same activity after having an explanation of the culture model. [12]

Culture Model Lets Us Ask Useful Questions

Agile is about people. OK, not such a startling conclusion. Seems we all know that. What is interesting is that when we start thinking about Agile as a *specific culture* we can now use this for asking interesting questions:

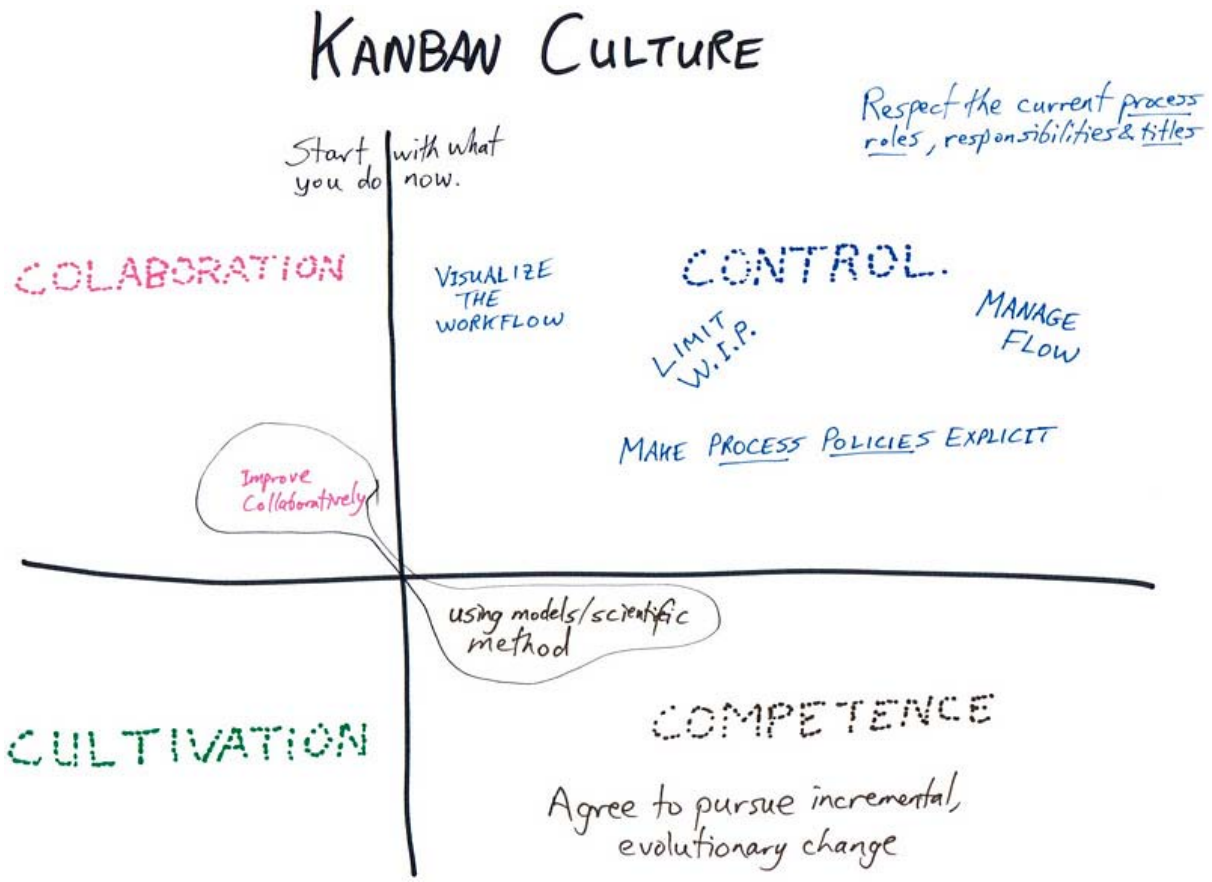
- What is the culture in my company now?
- How well is the culture aligned with Agile?
- What problems can I expect due to misalignment?

More on this in the section *Working with your Culture* below.

Kanban Culture is Aligned with Control

I am choosing an insightful post by David Anderson - The Principles of the Kanban Method [13] as the basis for my analysis. David is arguably the main leader of the Kanban/Software school with his book, very active mailing list and Lean Software and Systems Consortium.

As with the Agile manifesto, I have taken the Kanban principles and aligned them with the Schneider culture model. As can be seen in the following diagram, Kanban is largely aligned with Control culture.



Control cultures live and breathe policies and process. Kanban has this in spades. Control culture is also about creating a clear and orderly structure for managing the company which is exactly what Kanban does well. Control cultures focus on the company/system (vs. people) and current state (vs. future state). This is a good description for the starting place used with Kanban.

What is really interesting from a cultural analysis perspective is the principle: Improve collaboratively using models and scientific method. These two concepts really don't mix, so how can this work? According to Schneider, other cultural elements can be present as long as they support the core culture. So having some people focus is fine as long as it supports controlling the work. The notion of evolutionary or controlled change can also be compatible with a Control culture if it is used to maintain the existing organizational structure and hierarchy.

Wait a Minute - Kanban is Agile, isn't it?

Mike Burrows made a very influential post [14] where he argues that Kanban satisfies each of the Agile Principles. Now that I am studying this from the perspective of culture, I see that this is only weakly the case.

Agile and Kanban for sure share a common community, and many practices may be cross-adopted. However, they are fundamentally promoting different perspectives. Agile is first about people and Kanban is first about the system. Yes, people can be important in Kanban too, but this is secondary to the system. See CrystalBan [15] for ideas of how to integrate people into Kanban. So is Kanban Agile? I used to think so. I don't any more. I can see now how the belief - that Kanban is Agile - is harmful since the cultural biases are different.

Kanban is a Good Tool

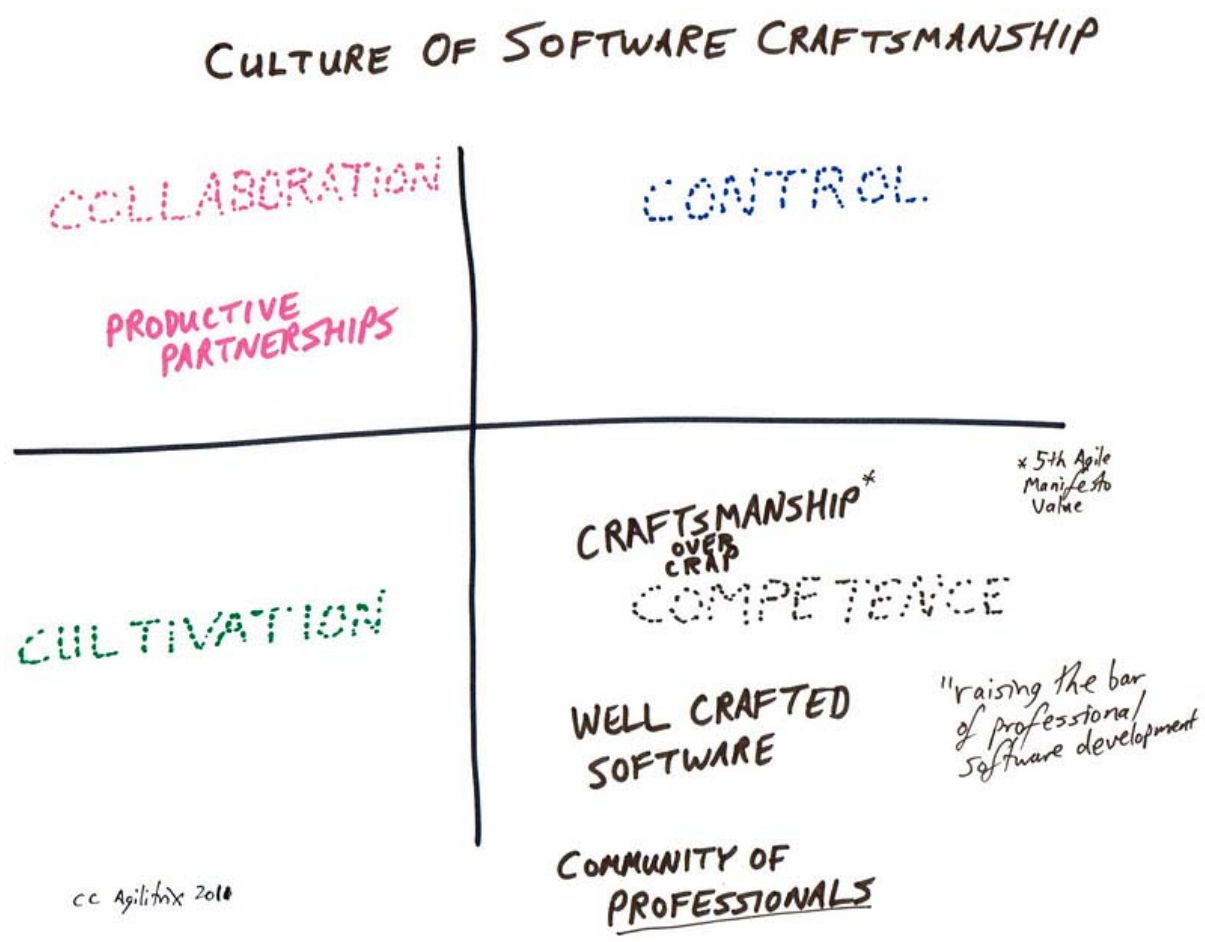
Sometimes when I share this analysis (where Kanban is linked to Control culture), I get a negative reaction since within the Agile community since *Control culture is anathema*. (Actually, it is anathema to knowledge work in general). To avoid any misunderstanding, I would like to clarify a few things:

1. I love Kanban and think it is great. We need more of it in the world. See my related blog post - Scrum or Kanban? Yes! [16] - where I argued that some situations are a better fit for Kanban vs. Scrum.
2. I am not saying people who use Kanban are control freaks or prefer command and control. What I am saying is that if your company has a Control culture, then Kanban is a good tool to help (vs. Scrum).

Software Craftsmanship as about Competence

The rise of anemic Scrum was noted to dismay among the Agile community and in particular by "Uncle Bob" Martin who coined the fifth Agile manifesto value of Craftsmanship over Crap (Execution) [17]. This gave rise to the much needed community of Software Craftsmanship [18].

I have already established that the Agile community is focused on Collaboration and Cultivation at the expense of Competence. We as a community of software professionals do need to pay attention to Competence and technical excellence for long term sustainability. For further information on this, see Uncle Bob's recent article The Land that Scrum Forgot [19]. The diagram below relates parts of the Software Craftsmanship Manifesto to cultures identified in the Schneider model.



Not surprisingly, there is a big focus on Competence Culture. This culture achieves success by being the best. And craftsmanship is about being the best software developers possible.

The value of *productive partnerships* stands alone. I am curious as to what purpose this supports as it is not directly related to writing quality software. I am wondering whether:

- this exists as a bridge to the Agile community?
- is related to the strong XP practice of pairing?
- is intended to appeal to the need for mentorship?

Why We Need to Care

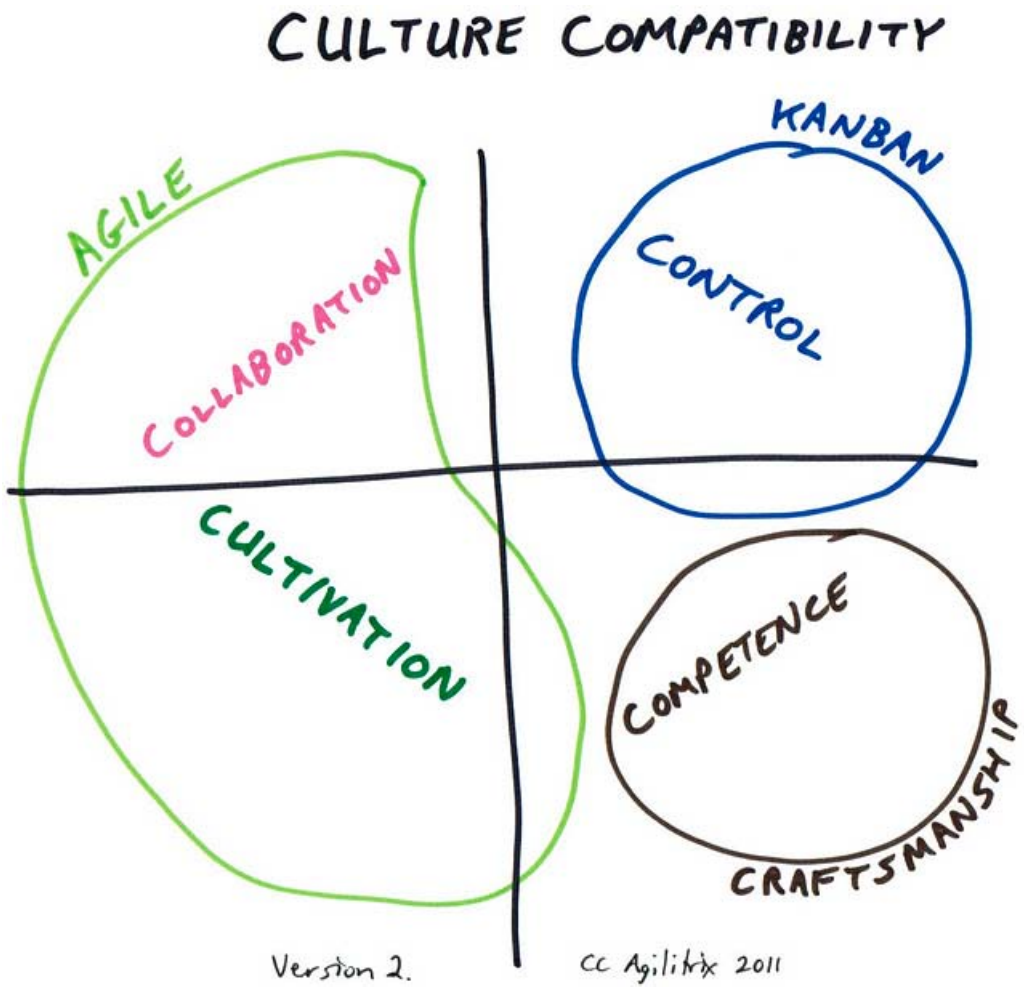
Craftsmanship *needs to exist* to make sure that the technical practices promoted by XP don't get lost in fluffy bunny Agile culture. Things like: refactor mercilessly, do the simplest thing that could possibly work, TDD, ATDD, continuous integration, continuous deployment, shared code ownership, clean code, etc.

The creation and existence of a separate movement to support Competence culture that exists outside of the Agile, supports the assessment of Agile culture as focused on Collaboration and Cultivation and not Competence.

As a final footnote before departing the culture of Software Craftsmanship, I would like to reflect that the manifesto does not accurately reflect a key aspect of the movement: a *deep commitment to learning and growth* (Cultivation culture). This is a value that exists to support the goal of excellence in software construction.

Working with Your Culture

Consider the following diagram illustrating how Agile, Kanban, and Craftsmanship principles align with various cultures. The shapes illustrate the dominant culture for each of Agile, Kanban and Software Craftsmanship based on the analysis earlier in earlier sections.



The diagram can be used as a *playbook* to determine what approach builds on the culture at your company.

- Control Culture -> Lead with Kanban
- Competence Culture -> Lead with Software Craftsmanship
- Collaboration or Cultivation Culture -> Lead with aspects of Agile that align with the organizations culture. e.g. Vision and Retrospectives for Cultivation Culture.

Understanding Culture

The starting point for making culture work is to understand it. Schneider's book describes a survey you can give to staff [20]. The book suggests using survey results as a starting point for culture workshops with a diverse group of staff.

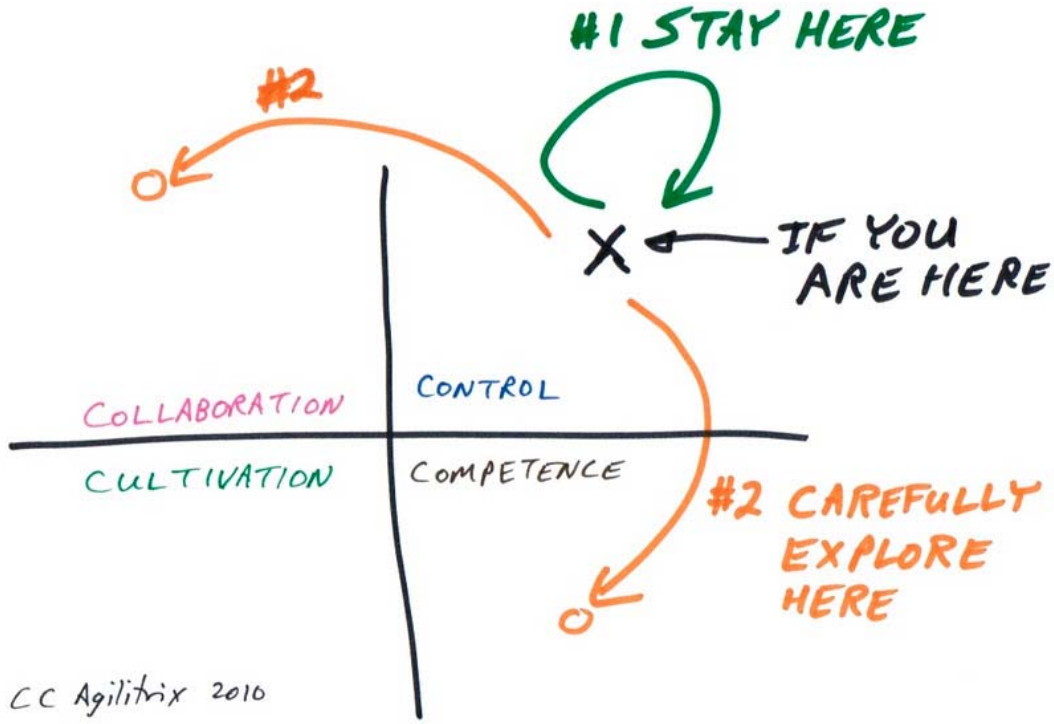
Management guru Peter Drucker says "*Culture … is singularly persistent …* In fact, changing behaviour works only if it is based on the existing 'culture'". The implication here is that it is not possible to just switch over from a Control culture to and Agile one.

A central premise of Schneider's book is that it is essential to work with the existing culture rather than oppose it. There are several suggestions for using cultural information to guide decision-making:

1. Evaluate key problems in the context of culture. Sometimes changes are needed to bring the culture into alignment with the core culture.
2. Sometimes the culture is too extreme (e.g. too much Cultivation without any controls - or vice versa!), and elements from other cultures are needed to bring it back into balance.
3. Consider the possibility of creating interfaces/adapters/facades to support mismatches between departments or groups.

What about Agile?

Consider the diagram below showing effective ways of working with culture



Option #1 illustrates that the easiest option is to work with the existing dominant culture (in this case Control). Option #2 is to carefully explore adjacent cultures in ways that support the core culture of the group. Choice of direction may be guided by what the secondary culture of the organization is. The idea here is to work with the culture, and not to fight it.

How to Change Culture is Another Story

Changing culture is difficult. See my series of blog posts on culture [21] for some details on how to change culture or wait for my upcoming eBook on InfoQ.

Conclusion

Congratulations! You now have the Schneider Culture Model - an easy-to-use tool for assessing culture in your company. Once you know your company's culture, you will be aware of how it is influencing many aspects of day to day work. More importantly, you can use the cultural fit model to decide what approach - Agile, Kanban, or Software Craftsmanship - will best fit with your organization.

References

[1] VersionOne, “State of Agile Development Survey”, http://www.versionone.com/state_of_agile_development_survey/10/

[2] Agile Manifesto, http://agilemanifesto.org/

[3] Dave Thomas, “Dave Thomas Unplugged - Agile 2010 Keynote”, http://agilitrix.com/2010/08/agile-2010-keynote-by-dave-thomas/

[4] Tobias Mayer, “The People’s Scrum”, http://agileanarchy.wordpress.com/scrum-a-new-way-of-thinking/

[5] Tobias Mayer, “Scrum: A New Way of Thinking“, http://agileanarchy.wordpress.com/scrum-a-new-way-of-thinking/

[6] Bob Hartmann, “Doing Agile isn’t the same as being Agile”, http://www.slideshare.net/lazygolfer/doing-agile-isnt-the-same-as-being-agile

[7] Mike Cottmeyer, “Untangling Adoption and Transformation”, http://www.leadingagile.com/2011/01/untangling-adoption-and-transformation/

[8] Michael Spayd, “Agile Culture”, http://collectiveedgecoaching.com/2010/07/agile__culture/

[9] Israel Gat, “How we do things around here in order to succeed”, http://www.agilitrix.com/2010/08/how-we-do-things-around-here-in-order-to-succeed/

[10] William Schneider, “The Reengineering Alternative: A plan for making your current culture work”

[11] Don Edward Beck, Christopher C. Cowan, “Spiral Dynamics : Mastering Values, Leadership, and Change”

[12] Michael Sahota - Workshop Results on Culture, http://agilitrix.com/2011/11/workshop-results-on-culture/

[13] David Anderson - The Principles of the Kanban Method, http://agilemanagement.net/index.php/Blog/the_principles_of_the_kanban_method/

[14] Mike Burrows, “Learning together: Kanban and the Twelve Principles of Agile Software”, http://positiveincline.com/?p=727

[15] Karl Scotland, “Crystallising Kanban with Properties, Strategies and Techniques”, http://availagility.co.uk/2011/08/03/crystallising-kanban-with-properties-strategies-and-techniques/

[16] Michael Sahota, “Scrum or Kanban? Yes!”, http://agilitrix.com/2010/05/scrum-or-kanban-yes/

[17] Robert Martin, “Quintessence: the Fifth Element for the Agile Manifesto”, http://blog.objectmentor.com/articles/2008/08/14/quintessence-the-fifth-element-for-the-agile-manifesto

[18] Manifesto for Software Craftsmanship, http://manifesto.softwarecraftsmanship.org/

[19] Robert Martin, “The Land that Scrum Forgot”, http://www.scrumalliance.org/articles/300-the-land-that-scrum-forgot

[20] Schneider Culture Survey, https://www.surveymonkey.com/s/VVNT5FB

[21] Michael Sahota, “Agile Culture Series Reading Guide”, http://agilitrix.com/2011/04/agile-culture-series-reading-guide/

How Software Architecture Learns

Christine Miyachi, Principle Systems Engineer and Architect, Xerox Corporation
http://home.comcast.net/~cmiyachi, Blog: http://abstractsoftware.blogspot.com/

Introduction

"All models are wrong, some are useful"
--generally attributed to the statistician George Box

"All *buildings* are predictions, and all *buildings* are wrong"
-Russell Brand, "*How Buildings Learn*" video
(http://video.google.com/videoplay?docid=2283224496826631552&emb=1&hl=en)

But maybe some buildings are useful. And if we replaced the word buildings with software, we come to the deduction that all software is wrong but some software is useful.

In the field of software design and software architecture, analogies are often made to building architecture. And these analogies are often criticized [1]. However, building architecture has been in practice for many hundreds of years where software architecture has only been around for less than fifty and only recently recognized by some as discipline. Software architects rarely go back and look at their work and how it has fared over time. (An exception to this is Grady Booch's work creating the "Handbook of Software Architecture" [2]) Russell Brand says in his book "*How Buildings Learn*"[3], that this is very true also with building architects. They build grand facades meant to stand out as monuments and almost never go back and study how their work has changed over time. And as Brand shows in his book, buildings change radically over time.

Software architects do not work like building architects. They don't design blue prints and then hand them off to builders. They typically work with an interdisciplinary team of software developers and business people. They interpret requirements into functional systems. They often still write code. In the past, they often got to see their working systems in the field and were then responsible for the rework and bug-fixing. Currently, much of the bug-fixing and changes requested by customers are delegated to off-shore teams leaving the original designers free to work on new projects but unable to learn how their original architecture changes over time and what could be done to make it better.

When studying building architecture, three things come to the surface:

1. Software designers can learn a lot from building designers
2. Software designers and building designers make the same mistakes
3. Building designers can learn a lot from software designers

In this paper we will examine all three learning opportunities to pave a way for better buildings and better software.

Defining Architecture

Software Architecture has a several definitions. The standard on software architecture - IEEE 1471-2000 says:
Software architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

In “A Taxonomy of Decomposition Strategies Based on Structures, Behaviors, and Goals” [4] , Koopman describes architecture as containing *structures*, *behaviors*, and *goals* along with a variety of decomposition strategies.

In relation to buildings, architecture is defined as [5] :
The planning, designing and constructing form, space and ambience that reflect functional, technical, social, environmental, and aesthetic considerations. It requires the creative manipulation and coordination of material, technology, light and shadow. Architecture also encompasses the pragmatic aspects of realizing buildings and structures, including scheduling, cost estimating and construction administration. As documentation produced by architects, typically drawings, plans and technical specifications, architecture defines the structure and/or behavior of a building or any other kind of system that is to be or has been constructed.

The Software Development Process

The software development process started out much like a building development process. This was dubbed “The Waterfall Method":

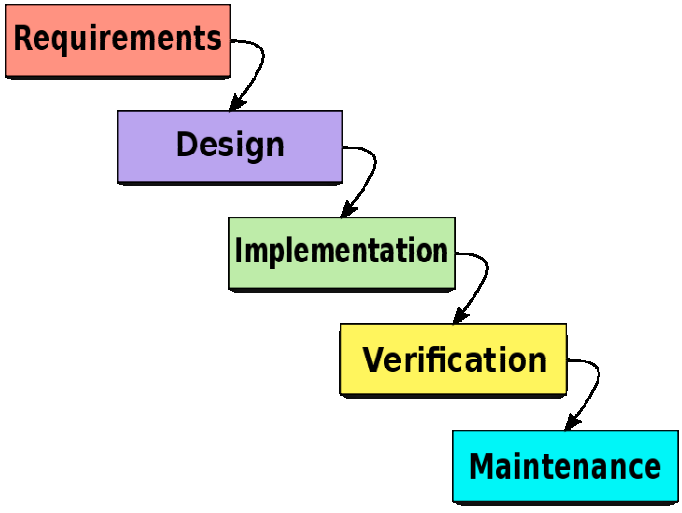


Figure 1 - The Waterfall Process

In this methodology, all requirement analysis, design, and architecture are done up front. As with building architecture, the blueprints are defined at the beginning of the process. But as Russell Brand points out in his book “How Buildings Learn”, what happens to buildings and software over time is that significant change occurs. Software designers have noticed this and have been working with a various software processes such as Scrum or Extreme Programming inspired by the “The Agile Manifesto”[6]. In these processes the change is embraced. Brand also points out projects where buildings are made with the requirement to be changeable by the owners.

In both these processes, change is done incrementally and a working system/building is produced. This approach to software development is called iterative:

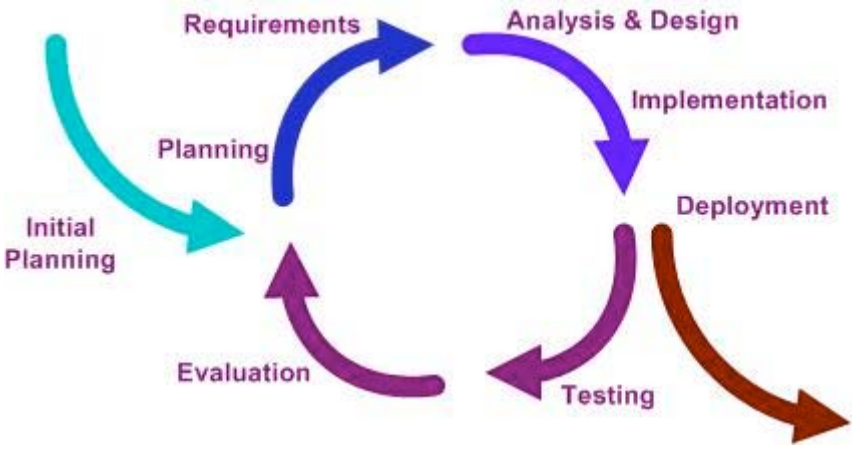


Figure 2 - The Iterative Process

In this process, requirements analysis, design, and architecture are done at each iteration. A final system is complete after several cycles.

Flow

"Flow, continual flow, continual change, continual transformation"
--Rina Swentzel, Peublo Indian architectural historian

"Responding to change over following a plan - focus on quick responses to change and continuous development. [6]
--The Agile Manifesto

When humans architect something, it is designed often not to adapt. After all, if the architecture is correct, it will meet all needs, now and in the future. It will serve as a monument, and work of art, and meet all the functional requirements. Unfortunately, with both buildings and with software this is often not the case. Buildings, as Russell Brand points out in his book *How Buildings Learn* are often made as works of art and sometimes don't even consider function. Software, while it is often designed with function in mind, does not take into account future considerations. As an example, Brand points to the MIT Media Lab.

Figure 3 - MIT Media Lab

This has a shared atrium and shared entrances but people are completely cut off from each other. Moreover, the building had a slew of problems when it was first opened. There was a terrible stench in the corridors, and doorknobs fell off, among other things. This building was a striking

new monument on campus but according to Brand, did little serve its inhabitants (As an MIT alumnus, I knew nothing of these problems and the media lab was always regarded as a revered building on campus. I never talked to the inhabitants though.) In time, the media lab was fixed and a new space was added that was significantly different.

Figure 4 - The new wing of the MIT Media Lab

In this building, the spaces were open, the building was open to sunlight and spaces were shared.

In software, I once created an interface to allow a user to configure changes in security. However over time, users wanted to configure more than just security items, so the interface changed to allow users to configure other aspects of the system. The name of the interface was still “Configure Security” and this may be a bit confusing to the user, because the function had changed. However, the interface and its underlying functionality had to change over time.

In 1896, Louis Sullivan, a Chicago high rise designer, wrote “Form ever follows function”. However, architects can rarely anticipate function both in buildings and software. Brand says “Function reforms form, perpetually.” [7] In software design, continuous change has become the standard as most software use Agile or Iterative process. Three of the principles of Agile development are focused on change:

- Customer satisfaction by rapid delivery of useful software
- Welcome changing requirements, even late in development
- Working software is delivered frequently (weeks rather than months)

The engineers behind the Agile Manifesto noticed that much of the software being developed did not meet customer needs. A study done of DOD projects showed many of the features implemented were rarely used or never used:

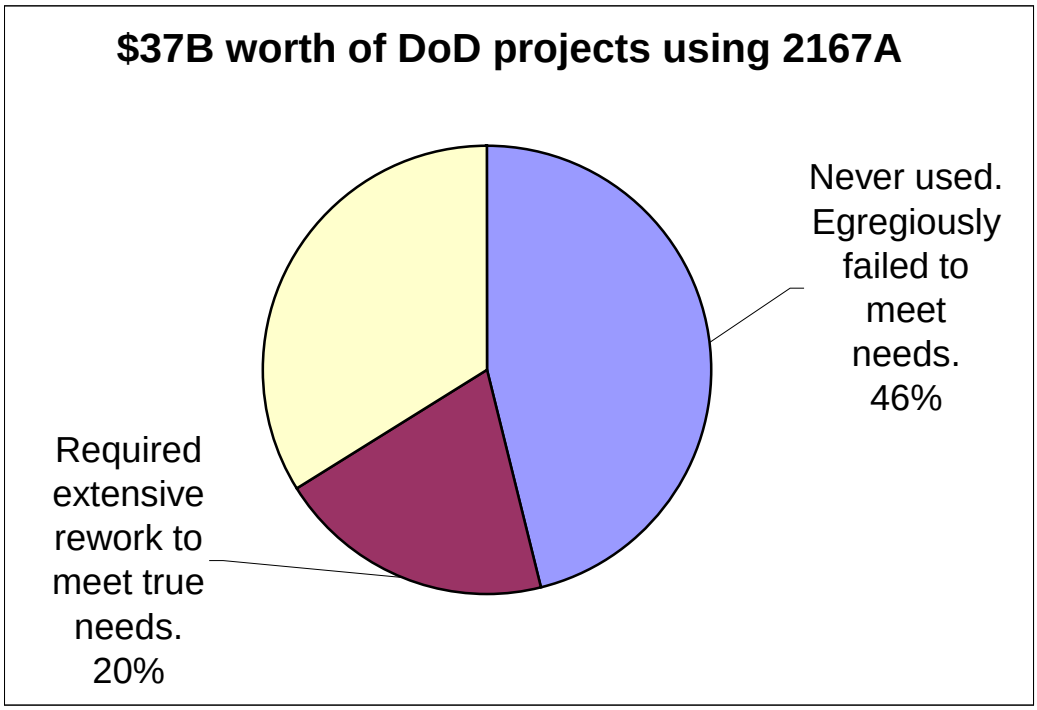


Figure 5 - Requirements that didn't meet the needs of the user

Brand also noticed that buildings were built to look impressive and that many elements were not thought through for functionality. The architects looked at the building from the outside first, and considered the inhabitants too late or never. To rectify this, the Agile creators advocated for bringing in the user early on in the development process. By delivering incremental changes to the user, the user can see if the software meets their needs. They may even change the requirements during this time. The Agile creators felt this change should be embraced rather than trying to freeze requirements. Changing requirements after all are part of the business world and if not embraced, the software will be more useful and thus more successful.

The best buildings, Brand says in his book, are buildings that could be changed over time. Winston Churchill said “We shape our buildings, and afterwards our buildings shape us.”[9] But Brand then says this process goes on: “First we shape our buildings, then they shape us, then we shape then again, ad infinitum”. Working on an Agile process is much the same. We deliver a software function, the user tries it, and changes it, we rework it and deliver it again. At some point we do have to deliver that functionality, but it incrementally changes until it satisfies the customer need, which may in fact change over this process.

Building for Change

Despite the Agile Manifesto call for continuous improvement, in practice, once a feature is delivered it is considered “done”. Because the functionality to be delivered is the main focus of an Agile team, does the designer / architect consider how the software will need to change in the future? Does he or she have time for that? What about future maintenance?

Brand discusses in his book that buildings are rarely built for maintenance. Buildings will decay if not cared for. Software, also will have architectural decay or turn into spaghetti code, if not cleaned out from time to time. When a building decays, Brand says it can either be reworked or torn down. Software also can be rewritten from scratch or reworked. In large software projects, despite software's malleability, managers are often reticent to wipe out an entire software module and rewrite it. Similarly, some buildings are hard to wipe out completely, particularly if they are in a crowded area.

Neglecting maintenance in buildings or maintainability in software has enormous costs. Brand says that if building designers just spent 5% of building design on maintenance issues, cost savings could be achieved. Theoretically in software, if maintainability was considered in the architecture, maintenance problems could be also reduced.

But one of the common practices when doing Agile development is: do not work on future requirements. The idea is to work on required functionality now and if future requirements need a reworked architecture, then architects must have the courage and fortitude to do it. This is easier said than done in an environment clouded with tight schedules and market pressure. Designing software with future change in mind is happening less and less in the industry as Agile processes become more wide spread.

One tradeoff in software would be to look at what areas rapidly change and design an architecture that can handle high change in that area of the system. In large software systems, this can be in areas where a new input is required or a new output is required. It tends to be around interfaces. Sometimes infrastructure changes with technology updates. For example, in the networking area of the software, the design could be made to handle changes for new protocols. In buildings says change happens from slowest to fastest with the following concepts: Site, structure, skin, services, space plan, and stuff. Site changes the slowest and stuff inside the building changes the fastest.

In software, this corresponds to the location of the software in the larger system, the structure of the software modules, the interfaces, what service the software provides, the layout of the interfaces, and then the data that flows in and out of the software module. All these areas can change, but designers can focus on the areas of most rapid change to have an impact on making a system adapt to change.

The Façade: A Building / Software Comparison

Design patterns in software architecture define particular structure and behavior commonly used. One such pattern is The Façade. The **facade pattern** is a software engineering design pattern commonly used with Object-oriented programming. The name is by analogy to an architectural facade.

A facade is an object that provides a simplified interface to a larger body of code, such as a class library. A facade can:

- make code that uses the library more readable, for the same reason;
- reduce dependencies of outside code on the inner workings of a library, since most code uses the facade, thus allowing more flexibility in developing the system; [8]

Buildings also have facades and although the façade does not change, the building inside changes drastically. An example is the Boston Athenaeum. These two pictures show the front of the building in 1896 and then in modern times. The exterior of the building is virtually

unchanged. The façade becomes loved over time and is a secure point. The building is cherished as unchanging despite the huge changes going on inside the building. Similarly in software the façade protects the user from the underlying churn of the implementation.

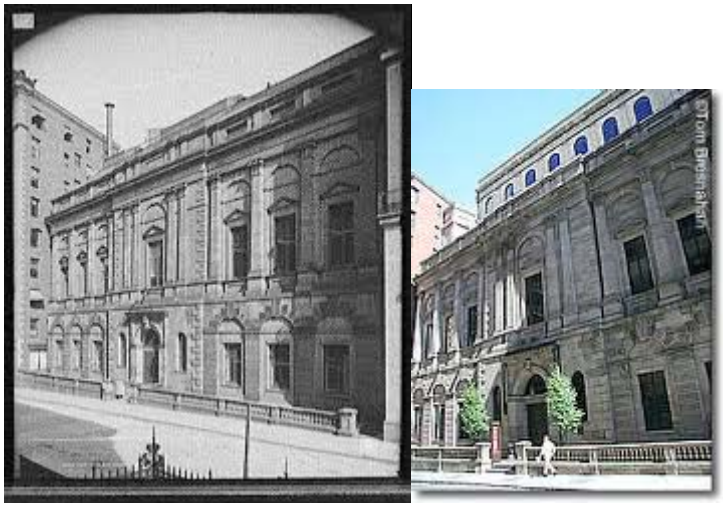


Figure 6 - The Boston Athenaeum

Conclusion

Brand says “Buildings live in time. In time we learn.” He points out that buildings change dramatically with fashion. Software design and processes also follow certain trends. Waterfall was popular in the early days of software development and Agile process are the common trend today. But are we building software that can change over time? Building designers can learn from Agile techniques. Brand points out buildings that are made as a shell, where the occupants are brought in and allowed to mold the interiors to their needs. Software developers can learn about how buildings change over time that software should be designed for change *and* for specific functionality. Both disciplines can learn that doing some upfront work on maintainability and extensibility will pay big dividends in the future of the building or product.

References

1. (http://bexhuff.com/software-architecture-is-not-building-architecture , http://triebert.nl/downloads/072005_swarm_concluding_paper.pdf)
2. (http://www.handbookofsoftwarearchitecture.com/index.jsp?page=Main) where he creates a reference to the design of software intensive systems.
3. Brand, Russell, *How Buildings Learn: What happens after they’re built*, Penguin Books, 1994
4. (DE-Vol. 83, 1995 Design Engineering Technical Conferences, Volume 2, ASME 1995)
5. http://en.wikipedia.org/wiki/Architecture
6. http://Agilemanifesto.org/principles.html
7. *Why Buildings Fail*, p. 3
8. http://en.wikipedia.org/wiki/Facade_pattern
9. *How Buildings Learn*, Chapter 1

Understanding of Burndown Chart

Dusan Kocurek, ScrumDesk, www.scrumdesk.com

Being a startup is a great period in the company's lifetime. Developing a product in which you believe with a small and highly motivated team is remarkable experience that you hope will stay forever. Successful products grow quickly in complexity terms that lead to more complex processes and metrics just to measure the progress ending up in non-usable results. The software industry is recognizable by high failures rate, late delivery, low quality, but still producing each day products used by millions of people.

Back to Basics

In the last ten years we have learned how to create products in an agile environment. *Agility* in software industry is more accepted these days because it brings visibility, intensive collaboration and simplicity. The clients appreciate radiated progress and insight into 'kitchen'. Agile packs everything we might remember from the beginning of the startup.

People tend to search for good agile metrics that provide this visibility. They expect to have a set of charts giving a deep understanding of the product status and the effort made and remaining. Reality, however, shows that such complexity is not necessary. Especially for the team, there are only few charts that team should use on daily basis. The *Burndown chart* is such a fundamental metric.

The Burndown chart is very simple. It is easy to explain, easy to understand. But there are pitfalls observed in many agile workshops and adoptions.

People tend to think that the Burndown chart is so simple that they do not give appropriate attention to understand what it says.

Burndown Chart

As a definition of this chart we can say that the Burndown chart displays the *remaining effort for a given period of time*.

When they track product development using the Burndown chart, teams can use a sprint Burndown chart and a release Burndown chart. This article concentrates on the sprint Burndown chart as it is used on daily basis.

Sprint Burndown Chart

Teams use the sprint Burndown chart to track the product development *effort remaining in a sprint*.

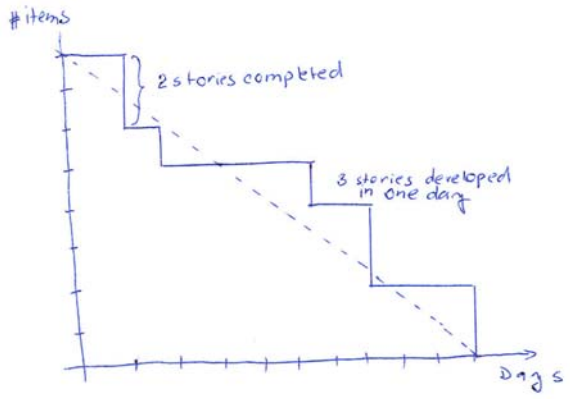
General speaking the Burndown chart should consist of:

- X axis to display working days
- Y axis to display remaining effort
- Ideal effort as a guideline
- Real progress of effort

Companies use different attributes on the Y axis. All of them have benefits and drawbacks.

The number of stories

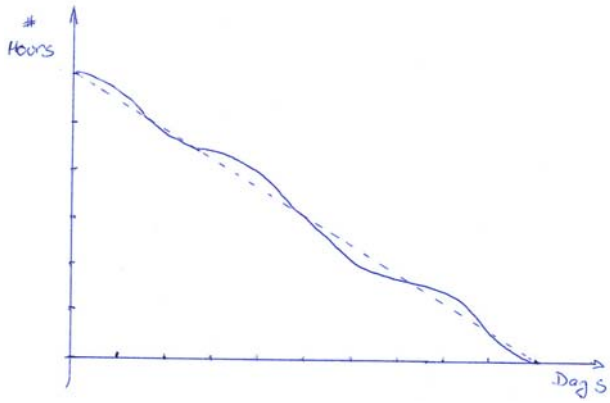
New agile teams used to start with the Burndown chart that displays the number of items on the Y axis.



It is a *big mistake* if they are allowed to continue with this approach. The reason is simple; stories require different effort to be completed. The first completed story can be two times bigger than the second one, but *the chart does not explain the real iteration status*.

Time unit

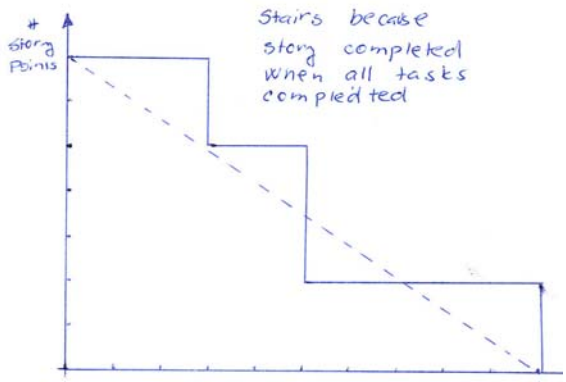
Time unit (hours or days) represents *remaining time necessary for completion*.



The benefit of displaying time is that it provides *more granular view* of progress, but *it leads to micromanagement* as well. It is typically suggested for *new agile teams*. Teams are often asked to carefully track time spent and remaining time, but all those who ask their team to do so should understand that the team is *hired to develop a real product*, not to spend amounts of time. Therefore most agile teams track a *remaining size*.

Remaining size

The chart displays the remaining *size of all stories in a sprint* backlog that needs to be done, using story points.



Stories are typically bigger items than tasks. Stories are also considered as done only when all tasks are completed. This leads to *stairs* in the Burndown chart.

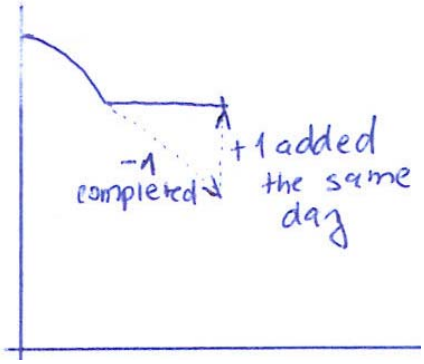
Such stairs are usually not evaluated correctly. Especially management reads them as *'no progress'* while they mean *'effort continues, it has not been completed yet'*. Steps could be smaller if stories are broken down in an appropriate way.

More experienced teams work with stories (level of what, not how), that are enough for a daily synchronization. It is less comprehensive, but sufficient. It allows to have shorter daily meetings that are more focused on 'what we solved, what remains'.

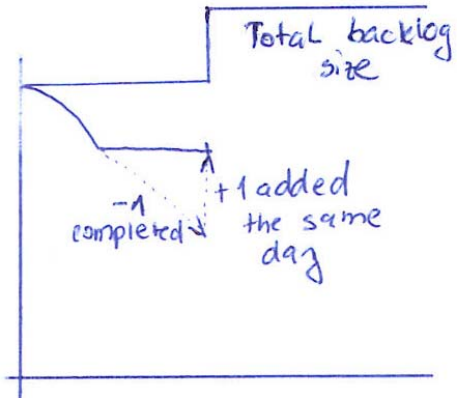
Bring an Understanding

Simplicity of the chart does not help if process gaps need to be identified. In this case, two lines are not enough as they simply display a summary of work for all team members and the gaps need to be identified on a task board.

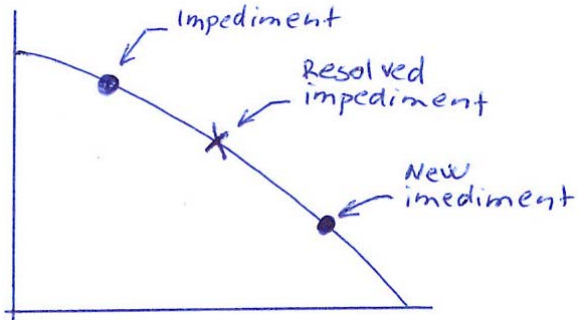
Usually it is unclear if a team is too late or somebody added additional work. Especially, in the case that an equal amount of work has been completed, there will be *no progress indicated*.



In this situation, viewing the total size of the sprint backlog should be helpful. Any change in the total size provides a clear explanation for the actual line issue.



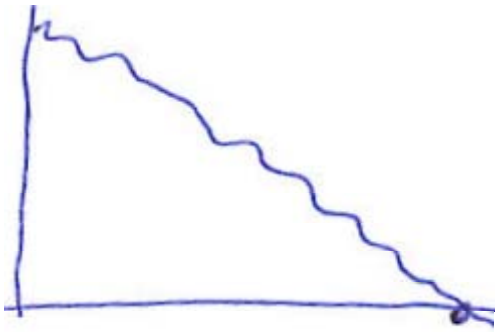
It might be helpful to display *impediment indicators* as well. I think that impediments are a pain that should be visible. Having them on a Burndown chart is a good way to combine them with the progress giving the sprint overview.



What a Burndown Chart Can Say

There are only two lines drawn in Burndown chart, but the situation they describe might have different reasons. In workshops, I can see that even certified Scrum masters are not able to explain situations described by Burndown charts correctly.

Ideal Team



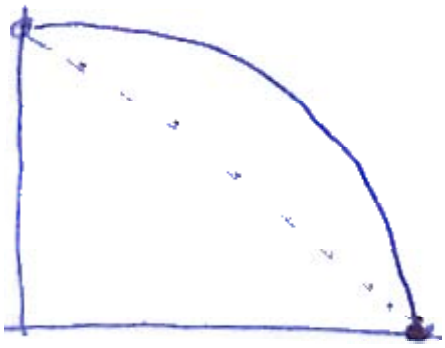
Such diagram indicates *the great team* able to organize itself. It indicates a *great product owner* who understands the reason for a locked sprint backlog and a *great Scrum master* able to help the team. The team is not over-committing and finished the spring backlog on time. The team is also able to estimate capacity correctly. No corrective action is necessary in such case.

Great Team

Such progress might be observed on charts of experienced teams. The team has completed work on time and met the sprint goal. They also have applied the principle of getting things done, but the most important is they have *adapted a scope of the sprint backlog* to complete the sprint. At the end the team has a possibility to complete some additional work.

In the retrospective, the team should discuss the reasons of late progress in the first half of the sprint and solve issues so they are better in the next sprint. The team should also consider the capacity that they are able to complete.

Nice Team



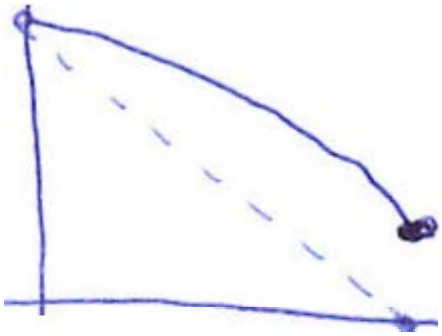
This is a *typical progress* that can be observed in many experienced agile teams.

The chart says again that the team was able to complete their commitment on time. They

adapted the scope or worked harder to complete the sprint. The team is self-reflecting.

The team should discuss change of plan immediately as they see the progress has been slowing down from the beginning of the sprint. Typically it is suggested to move a low priority item from the sprint backlog to the next sprint or back to the product backlog.

Boom. It Is Too Late.



This burndown chart says: “You have not completed your commitment”.

The team has been late for the entire sprint. The team did not adapt the sprint scope to appropriate level.

It shows that the team has not completed stories that should have been split or moved to the next sprint. In such situation the capacity of the next sprint should be lowered. If this happens again, corrective actions should be taken after a few days when slower progress is observed. Typically, lower priority story should be moved to the next sprint or back to the product backlog.

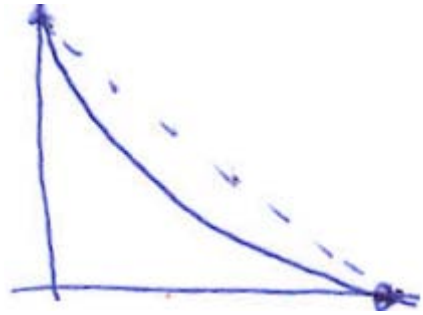
Boom. Too Early.

The team finishes its work sooner than expected. The stories were implemented, but the team didn't work on additional stories even it had the capacity to do it.

The stories were probably overestimated, therefore the team finished them earlier. Also the velocity of the team has not been probably estimated correctly.

The Scrum Master must be more proactive in either getting the team to fix estimation or ensure additional stories are ready to be added into the current sprint.

Let's Have a Rest

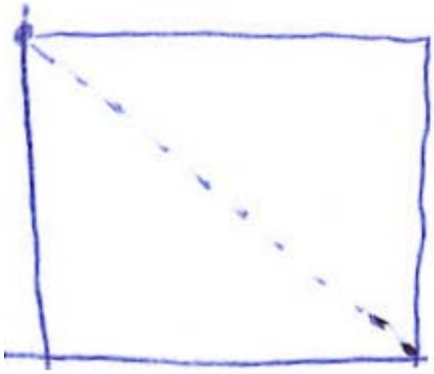


The team with such progress has a problem. The problem is either the team committed to less than they are able to complete or the product owner does not provide enough stories for the sprint.

The reason might be also an over-estimation of complexity, which ends up in completion earlier than expected at the beginning of the sprint.

The Scrum Master should identify this problem earlier and ask the product owner to provide the team with more work. Even if stories are over-estimated, the team should at least continue with stories from the next, already preplanned, sprint.

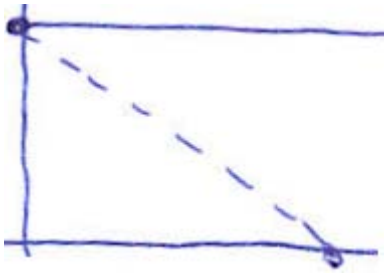
Oh, Management Is Coming!



The team is probably doing some work, but maybe it does not update its progress accordingly. Another reason might be that the product owner has added the same amount of work that was already completed, therefore the line is straight.

The team is not able to predict the end of the sprint or even to provide the status of the current sprint. The Scrum Master should improve it Scrum masterships and coach the team on why it is necessary to track the progress and how to track it. Such team should be stopped after two or three days that shows a flat the line of progress and should immediately apply corrective actions.

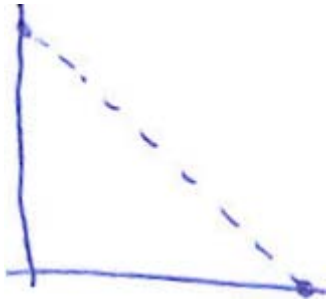
Do Your Duties



The team is non-functional on many levels. The Scrum Master of this team is not able to coach the team why it is necessary to track progress on daily basis. The product owner does not care about development progress either.

To fix this situation the team should restart. *Restart from scratch* by training and do a retrospective to figure out why this is happening.

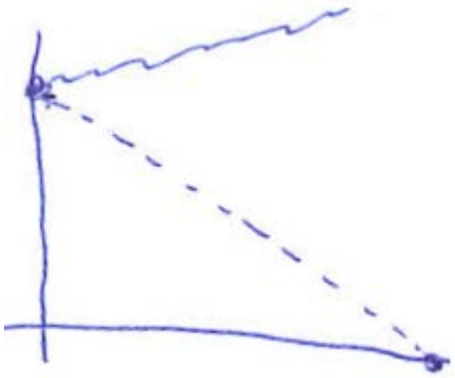
Zero Effort



A chart like this indicates that stories or tasks were not estimated during the sprint planning meeting and the sprint has not officially started yet.

To fix this situation, team should immediately arrange a planning meeting, estimate the user stories, include them in the sprint according to their velocity and start the sprint.

Up to the Sky



The first sprint typically looks like that. It is good source of knowledge because the team has failed significantly and it is highly visible.

Stories or tasks were added into the sprint backlog everyday without any progress recorded. Another reason might be that tasks were re-estimated constantly during the sprint.

The mistake is that the team did not identify the problem that the chart displays.

The sprint backlog should be reevaluated and rearranged immediately. The coach might be helpful, as an experienced Scrum master and product owner should often facilitate this situation.

Bump on the Road

The team *has not started sprint correctly*. They added stories after the sprint had started. It is positive that they recognized that planning is missing planning, it is however too late. The team should be careful about the capacity estimation as planning happened in the middle of the sprint, not before it.

In such case it is suggested to restart the sprint, even within a shorter timeframe.

Progress from Long-Run Perspective

Burndown charts described in previous paragraphs are description of iteration. But does a nice Burndown chart indicate a *great team*? Maybe if your team indicates great progress for more than one iteration. Does the team believe in such success? I would be personally careful. We all know about changes coming every minute. Maybe the team provides conservative estimation for their safety.

Management usually takes care about the improvement of velocity, sprint by sprint. Please, do not expect that. Velocity is not an indicator of the team. *Velocity is not a KPI* by which you should measure your team. *Velocity is just capacity planning tool*. Nothing more, nothing less. Asking people to accomplish more story points in iterations will result in stories that have more story points estimated without real reasons. It could be name as *"Story points inflation"*.

Finally

I've started with a description of the startup environment in which the team spirit is crucial for success. Agile is probably the best way to develop product in startups because it allows focusing on product. The product often needs to get to market as soon as possible without losing competitive advantage. But even such agile environment needs a focus. *Burndown charts are easy way how to radiate visibility* on your path, but they should be read and understood correctly without early conclusions. I hope this article will provide you a guideline on how to help your teams.

The Psychology of UX - Part 2

Vanessa Carey, Caplin Systems, http://www.caplin.com/

Welcome back to The Psychology of UX series! This article continues to discuss the "10 Things to About Human Psychology That Should Inform UX Design" proposed by Susan Weinschenk. The first part of this article was published in the Fall 2011 issue of Methods & Tools.

6. People are Easily Distracted

How do you get someone's attention? What do people notice? What distracts people? These are some of the questions we'll be exploring. Let's get started.

People pay attention to anything new and different, particularly movement.

If you think back to our Palaeolithic days, we needed to be survivalists. Any movements around us could be potential threats and we had to be alert and aware in order to exist. Similarly, our "old brain" (the first part of our brain to evolve) uncontrollably notices danger, food and sex. It may seem quite brute by today's standards, but once again it's all about survival. We need to eat, we need to procreate, and we need to protect ourselves so that we can continue to do those things.

But what does this mean in regards to design? *When appropriate*, placement of images of food, attractive people or potential danger/violence will focus our attention quickly on that area of the page. If this is suitable to your design aims, then by all means quickly grab the attention of the old brain. However this has been abused in recent times with companies trying to get easy looks with cheap usage of these types of imagery. My favourite example of this is a company called 'Nila Foods', who without any restraint apply gratuitous imagery to their trucks. But hey, it worked or I wouldn't remember their name!

We notice faces

The second most common thing that people pay attention to is faces. We can't help but look at faces. Be it because we wanted to see another person's emotions to develop relationships or to look for potential threats, faces capture and keep our attention; so much so that our focus will remain on the same part of the page even after the face has disappeared. When humans look at faces, we tend to look at the eyes first, so showing an image of a person looking directly into the camera is most powerful. Similarly, if we see the person in the photo looking in the direction of something, we'll look in the same direction. Usability specialist, James Breeze's screenshot of an eye-tracking experiment demonstrates this perfectly.

Use the senses to grab our attention

As we are limited to the visual and auditory senses with computers, we should attract those two senses using colour, size, relationship/position, noise and animation. If an entire site is designed in gentle shades of blues and then you place a bright red button in the middle, it's going to get attention. Similarly if you reading an article on a website, your eyes will most likely scan large headings, quotes and bold text more than the body of the text. We notice anything different. Why do you think I bold specific words in my blog posts so often?

However since people are easily distracted, **don't** use lots of sensory imagery and advertisements all over your website if you want a person to focus on one thing at a time.

But that doesn't mean they always notice…

People are prone to something called "change blindness". Depending on what we focus on, we can be completely oblivious to changes around us. Interesting how we only really notice what we are focused on at the time. Taking this into consideration for web design, don't assume that because you've made a small change to a webpage (for instance after a click to a new page) that people will notice. Make things obviously different and bold so the user will easily recognise the difference.

7. People Crave Information

You can't deny we live in an age where the average person obsessively checks their email and Facebook all day long. Endless Google queries. Combing through the surplus of tweets on Twitter. Random Wikipedia search sessions. Be it in front of their desktop, or staring intently into their mobile if they are on the go, we all exhibit these behaviours. We've come to think of it as normal. And in a sense… it is. The web taps into this natural inclination of ours: the desire to constantly seek out new information.

But what is the reason for this? Why are we so driven to learn new things?

It's simple, kids. It is called dopamine. Dopamine is a chemical released in the brain that makes us seek out things, such as love, food, sex, or even information. The simple act of aspiring to do something and the anticipation of doing it unleashes dopamine in our minds and creates a pleasurable mental state.

From an evolutionary perspective, we can see that the more you are actively seeking out these things, the more likely you are to survive. Without an appetite for food, you won't eat. Without a desire for the other sex, you won't procreate. Without a fervor for new information, you won't learn things (things that could potentially better or save your life). That's why learning is dopaminergic (meaning it causes dopamine to be made). Seeking out new information helps us survive.

Interestingly, people will often crave more information than they could realistically process at any given time because it makes them feel they have more options and thus more control over their lives, which all goes back to survival. Ever searched for an answer on Google, found it, but you continue to look for more answers to validate your question? This is another reason that most people are in some sense addicted to the web; it offers absolutely endless information on any subject you can imagine, if you know where to find it. Some of the most popular websites on the internet are ones extremely rich in content: namely Facebook, Google and Twitter.

So what does this mean in regards to web design?

Because dopamine is released during the stages of **searching** for information, **not** during receiving information, it isn't quite so simple as to say that if you give your user all the information they could possibly want and more, they'll be pleased. It's the activity of allowing the user to find information, revealing that information or surprising them with information that gives them enjoyment. In fact too much information on a screen can distract from other key information, thus frustrating the user. For this reason we need to make the information clear, clean and easy to navigate.

Choice = Power

The user will still want to feel that they have many choices or access to information. For this reason, the essential information needed to help them complete their task should be at their fingertips, as well as the ability to seek out more, with more details views, click-throughs, or tooltips with extra information.

"Ask and ye shall receive."

How can we give our users satisfaction in the journey of seeking out new information without making it unobvious or tedious? One way we can do this is by providing information to the user when the user asks for it. This also gives them a sense of control, which will heighten their experience. This could be in the form of expandable items, rollovers, contextual hovers, or click-throughs.

Feedback

Because humans like to be aware and knowledgeable, it is equally important to keep them informed of what is going on. Give them feedback as to what is happening behind the scenes. And importantly to do so in a human language. The computer doesn't need to inform the user that it is requesting file 1458xj via the server. The computer needs to tell the user that their file will be there and when. Progress bars, status updates, live help are just a few ways to do this.

8. Most Mental Processing is Unconscious

Most mental processing occurs unconsciously

The brain often acts without our conscious knowledge. The reason for this is that we have three brains. The old brain, the mid brain (emotional), and the new brain.

The old brain makes most of our decisions

…based on food, sex and danger. These things grab our attention because they determine our possibilities for survival. The old brain was the first to be developed in the evolutionary history of animals and humans. It is the part of the brain that constantly, unconsciously, assesses your environment, deciding what is safe and what isn't. It controls automatic unconscious processes like breathing and digestion.

As the old brains is concerned with survival above all, nothing is more important that 'YOU' to your old brain.As soon as something relates to you, or the word 'you' is used, your old brain switches its focus to that thing. Susan Weinschenk gives this example in her book 'Neuro Web Design' where she demonstrates the power of using the word 'you' to sway people towards a product.

"First product description: "This software has many built-in features that allow for photos to be uploaded, organised and stored. Photos can be searched for with only a few steps."

Now read this paragraph for the same product: "You can upload your photos quickly, organise them any way you want to and then store them so that they are easy to share with your friends. You can find any photo with only a few steps." Which product would you buy? You'd likely buy the one that says "you" and "your". *This is not a conscious decision. Your non-conscious brain will tell you that the second product is better for you.*"

In addition, the old brain is always looking for potential threats, food or opportunities for sex and therefore is very skilled at noticing change. As mentioned above, it is constantly scanning its environment observing changes. And there are a **lot** of changes in your environment. The estimate is 11 million piece of information every second. Of that, only 40 are conscious. The unconscious mind lets us process all incoming data and evaluate what is good or bad.

"The mind operates most efficiently by relegating a good deal of high-level, sophisticated thinking to the unconscious, just as a modern jet liner is able to fly on automatic pilot with little or no input from the human, 'conscious' pilot. The adaptive *unconscious does an excellent job of sizing up the world, warning people of danger, setting goals, and initiating action* in a sophisticated and efficient manner." -Timothy D. Wilson

It is a hugely efficient tool that shows us what to pay attention to consciously while skimming through the rest. As you might remember from my previous section, multitasking is impossible - we can only focus on one thing at a time - so we need to make sure it's worth our conscious attention. That's why it's such a successful system.

"The only way that human beings could ever have survived as a species for as long as we have is that we've developed another kind of decision-making apparatus that's capable of making very quick judgements based on very little information." - Malcolm Gladwell

The emotional brain is impulsive.

The emotional brain is (obviously) where all emotions are processed, and it is the root of impulses. Because of this it makes a big impact on our decision-making. The old brain and the emotional brain are very connected in the sense that if the old brain is highly aroused (by fear, or desire) the emotional brain deeply processes this information and etches it in our memory.

Because we are natural visual processors, we respond to pictures and imagery the most. Changes in visuals are easily picked up. Similarly when we think of stories or read, we break the ideas into images in our minds. These images arouse emotions in us. Imagine a news story of a plane crash with a front-page cover of burning, twisted metal shrapnel. That's going to affect your emotional brain quite a bit. Similarly photos of a sexual nature, food or potentially dangerous scenarios will grab our attention with the old brain and sway our emotions with the midbrain.

Our behaviour is affected by things we aren't consciously aware of.

"Unconscious processing can give rise to feelings, thoughts, perceptions, skills, habits, automatic reactions, complexes, hidden phobias and concealed desires." - Wikipedia

One way that scientists have observed this is in the instance of 'framing'. In 'framing', your old brain and new brain receive these unconscious messages and you act upon them. In one study, they saw that using the words "retired", "Florida" and "tired" actually made people walk slower. Amazingly, a great portion of people's behavior is driven by factors that they aren't even aware of. Both brains act without our knowledge. Rational reasoning is normally not the deciding factor. Both the old brain and the emotional brain act without our conscious knowledge. People will always assume they made a rational and conscious decision, but in reality our decisions always start from our old- and mid-brains, and sometimes finish there too. Some decisions may come from your new brain (rational), but most are based on the subliminal messages our other brains give us based on things we react to in our environment.

How does this affect web design?

When a website addresses all three brains, then we click and engage with the site. If a site is visually arousing, we'll pay attention. If it seems to address 'our' needs and relates to 'you', we'll pay attention. If there are a lot of changes, such as movement, carousels, videos, banner ads… it will grab our attention (even if we don't like it).

By tending to our old and mid-brain triggers (food, sex, danger, movement, change, visuals/imagery, and focus on 'you') with appropriate web design decisions, users won't stand a chance at resisting clicking around a bit.

9. People Create Mental Models

We are going to be talking about mental models. You might have a thought about what a mental model is already, but let me show you what it actually is. Oh wait, I just did. A mental model is simply a representation in the mind of a real or imaginary thing. It is the way you imagine something to be.

For example, when I say **cake**… this is what I think of in my mind:

Maybe not quite so tall, but this sort of airy, fresh, feminine vibe. But you might envision a cake looking different. That's because our mental models are different. If I were to bake you a cake, I might make one similar to my mental model because it would please me and satisfy my idea of how that thing should be. However you might be slightly disappointed if I gave you the pink fluffy yellow cake when your mental model of a cake was a rich dark chocolate cake smothered in ganache. As the cake baker (sub: designer), I would not satisfy you, the eater (sub: user) because I wasn't aware of your mental model before I baked the cake (sub: designed it). You dig?

The term "mental model" has existed for around 25 years or so and first came about from K. J. W. Craik, a philosopher and psychologist, in his book The Nature of Explanation, where he proposed that the mind forms models of reality and uses them to predict similar future events.

Since then the subject of mental models has been explored extensively by psychologists, and this definition, featured in a cognitive science article, sums mental models up nicely:
"A mental model represents *a person's thought process for how something works.* Mental models are based on incomplete facts, past experiences, and even intuitive perceptions. They help shape actions and behaviour, influence what people pay attention to in complicated situations, and define how people approach and solve problems."- Susan Carey, Cognitive Science and Science Education, 1986.

How are mental models formed?

It would be impossible for any person to know and imagine the entire world in their head. There are far too many things and too many variations on things for a person to be able to do that. So the human mind makes representations (images) of the world around them that they store in their mind. These numbered concepts exist in their mind and they can create relationships between the concepts to make sense of them and to represent the world around them. Some of the places from which mental models form are:

- Prior experience with similar sites or products
- Direct experience with the product
- Assumptions users have
- Intuitive perceptions
- Things users have heard from other people

And let's take into consideration that mental models are subject to change once the user has more experience or other assumptions arise about the thing.

How do mental models affect design?

First let's look at a different type of model, conceptual models, to better understand. A conceptual model is the actual model given to a user interface of the product by the designer. It is the concrete screens, buttons and interactions of the interface that the people who created it intended it to have. So a conceptual model is the actual look and behaviour of the thing, and a mental model is the idea the user has of that thing prior to or during interaction.

When a user's mental model and the conceptual model don't match up, you get a "bad" user experience. Bad in the sense that the user won't know how to use the thing, it will be hard to learn or they may not accept the thing altogether.

If they don't match, the system will be confusing to the user

Let's look at the example of an elderly person is used to reading paper books because that was commonplace for the majority of their lifetime. Hand them a Kindle, tell them it's a book, and the experience may be too jarring as it is completely different from their established mental model. In addition, a mismatched mental model may lead to user errors or require too much of the user for them to figure out how to use it.

"Many systems place too many demands on the humans that use them. Users are often required to adjust the way they work to accommodate the computer. Sometimes the result is a minor frustration or inconvenience, such as changes not being saved to a file. *Inaccurate mental models* of more complex systems, such as an airplane or nuclear reactor, *can lead to disastrous accidents."* Reason, 1990

Mismatched models are a common occurrence because they can arise from so many reasons. For example:

- If the designers thought they knew who the users would be (and their level of expertise), based the design on these assumptions, and then later found out the assumptions were wrong.
- If there are several user types for the interface and the designers only designed for one mental model. The other users' mental models would not match the design.
- Or maybe the designers don't design for ANY mental models of the users. The conceptual model thus represents the designer's mental model.
- Similarly, if the system isn't designed at all and is just a reflection of the underlying technology/hardware, the mental model matches that of developers (who use or made the technology/hardware)

But how do we match our conceptual models (designs) with the users' mental models?

"Typically, the burden is on the user to learn how a software application works. The burden should be increasingly on the system designers to *analyze and capture the user's expectations* and build that into the system design." Donald Norman, 1988

We can understand the users' mental models and design for that.

In order to understand the users' mental models we have to do research. We can do this through interviews, observation, or competitor analysis (pre-existing mental models the user might have). It's really quite simple; just talk to the user and understand where they are coming from, what their perspective is and how they expect to interact with the thing you are designing. Then incorporate these insights into the design, rather than designing from the designer's or developer's mental models of how the thing should be.

One good example is if you buy a new DVD Player. The player comes with a 50-page manual on how to connect the hardware, how to install it and how to get started. In order to fully use all the features on your new player, you will have to read the entire manual to understand how to use it. But if the company that designed the player would've put the focus on the user and their mental model of how a DVD player should work, they would've re-designed and re-engineered the product so that the user could instantly hook it up and play a DVD without any hassle of installing and setting up functions. Then the instruction manual could be reduced to a one page illustration, rather than 50-page bible of instructions.

OR we can teach the users a new mental model and prepare them for the conceptual model.

If you are creating an entirely new product or you are creating a product that revolutionises the way we think about an old product (like the Kindle did for the book), then it is okay to create a new conceptual model that changes people's mental models. But as we saw before, people's mental models affect the way they do or don't accept new conceptual models. Because of this, you need to prepare the user for the new conceptual model and in the process change their mental model before they have direct experience with the product.

You can do this through training and exposure, such as creating videos featuring the new product, advertisements, examples of usage or how-to's, and by generally putting your name and product out there in the public eye so that people can get used to it. Think: Steve Jobs unveiling every new Apple product before it reaches the public's hands. It worked!

One great example of an introduction video to a new conceptual model is 'Mint' who show you how their product works and how easy it is to use, while slowly convincing you to change your mental model about how a person should bank online. Yet if you notice, they also maintain certain mental models to help you get comfortable with the idea of Mint, like the visuals of spreadsheets, charts and graphs in the beginning of the video which lead you towards the new interface. By refreshing your memory on what your mental models of banking are, they show you they get it and then introduce you to their new conceptual model. The new interface is dotted with icons of typical financial symbols as well as more graphs and pie charts to monitor your money which not only hint, but shout of ideas of banking.

It's also helpful if you can compare the new conceptual model to an older one, or use metaphors, so that people can more quickly become comfortable with the idea of it. For instance, "Typing on a keyboard is just like typing on a typewriter, only you can undo your mistakes!"

When it comes to web design, as the web is a fairly new concept in regards to the length of human history, it's very important to link virtual concepts with real world examples through the use of metaphors. We can do this by using icons, buttons and visual affordances, like drop shadows to suggest depth of field, light or weight, similar to objects in real life. A really great article on metaphors on the web can be found here.

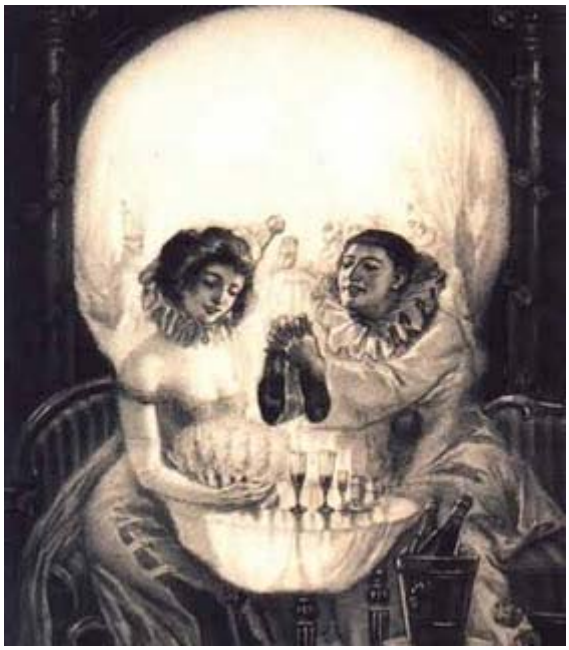
10. People Understand Visual Systems

One of the most famous theories on how humans visually perceive of things is the Gestalt principle. The Gestalt principle was first introduced as a psychology concept in Germany in the late 1800′s. The term "gestalt" literally means "form", and this represents the idea that the brain first sees the overall form of something and then begins to pick out the details. Let's look at an example:

It's Albert Einstein... or some guys with a lance.

At first you might see an old man with white crazy hair. But if you look closer at the details you'll pick out the horse, the lances, the two men in their armour and the windmill (that looks aflame).

Here's another example:



The face of death!

First we see the skull, but then we notice the couple drinking (a lot) of alcohol and….warming their hands over it?… Hey, I didn't say the details needed to make **sense**.. but they are there if we look.

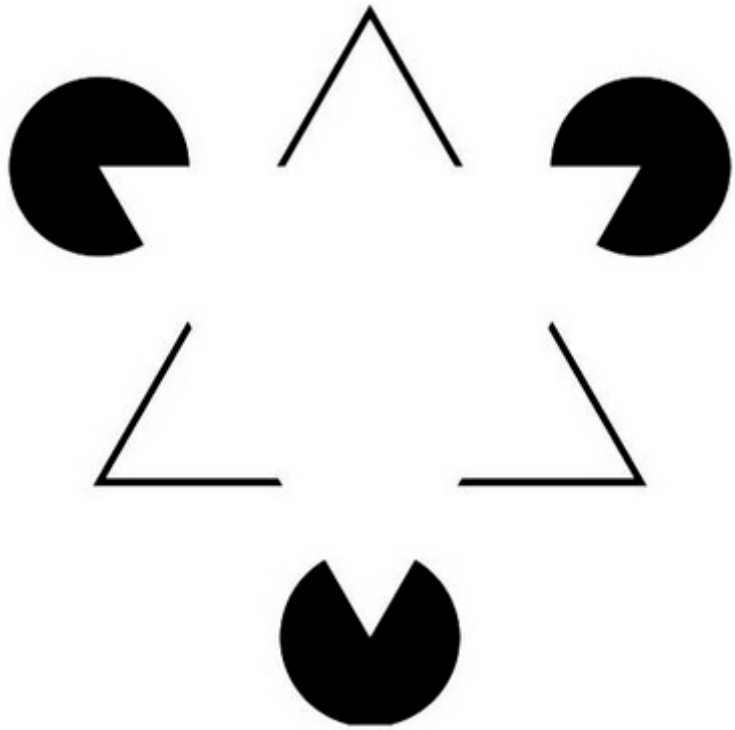
The point is: we notice the bigger picture first, then we hone in on the details. This idea has been expressed in the well-known phrase, "The whole is greater than the sum of the parts." Gestalt psychologists believe that there are inherent mental laws which dictate how we perceive of objects. Here are some of those laws explained:

The Laws of the Gestalt Principle

The Law of Closure

Our mind tries to close shapes and create objects even if only a bit of the shape is implied. We do this by ignoring gaps and completing contour lines. This was forms and shapes that can be imagined in the mind, even if not drawn out, from the negative space of other shapes.

One great example of 'The Law of Closure' is the Kanisza Triangle, seen below. This illusion was first explored in the 1950′s by an Italian psychologist who showed that two overlapping triangles are seen by the human eye, even though no complete triangles exist in the image.



The Kanizsa Triangle

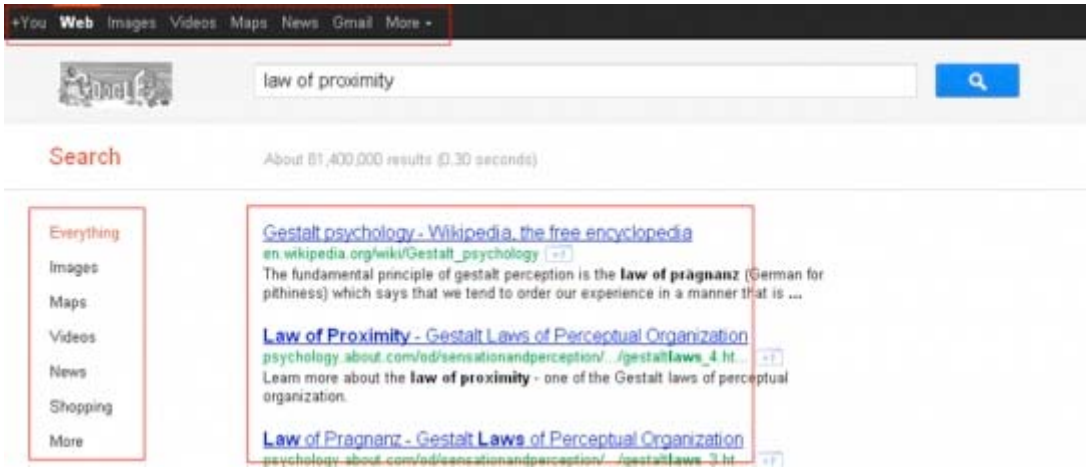
The IBM logo makes use of the ‘Law of Closure’ by suggesting letters with unconnected lines and gaps, but the human mind never for a second doesn’t understand that it reads ‘IBM’.
It's just a bunch of lines

The Law of Similarity

Our minds group similar objects together to create collective entities. The similarity of the objects is determined by their shape, size, and colour. In this example below, the human mind connects the lime green shapes together because they have a similar colour and form, even though they aren’t on the same plane or in the same area spatially.

The Law of Proximity

The spatial positioning of objects causes our minds to perceive of collective entities, or in layman's' terms: things that are close together are believed to go together. In the example below, the header navigational links are all close together (and in similar colours), the sidebar items are all close to each other, and the search results are all in alignment as well. These clear groupings and the distances between the groupings helps us establish similarity and relationships.



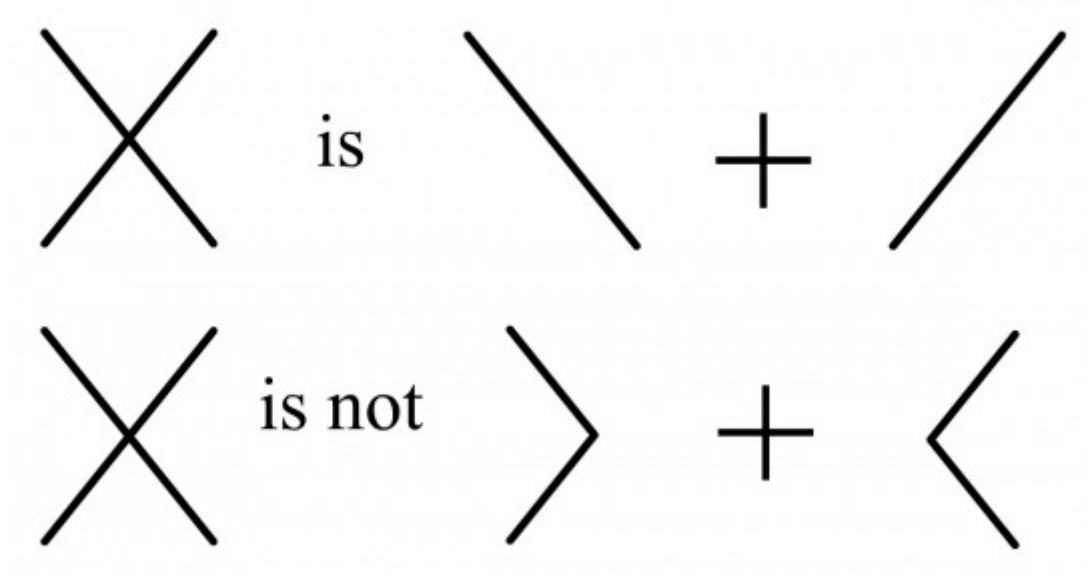
The Law of Symmetry

Symmetrical objects are perceived to be collective entities, regardless of their distance apart. Really I think we can translate this to mean that any objects which are identical, or symmetrical, are understood to be related to each other. In the example below, a common remote control design shows how symmetrical buttons relate to each other and humans innately perceive of these similarities.

Opposites but equals

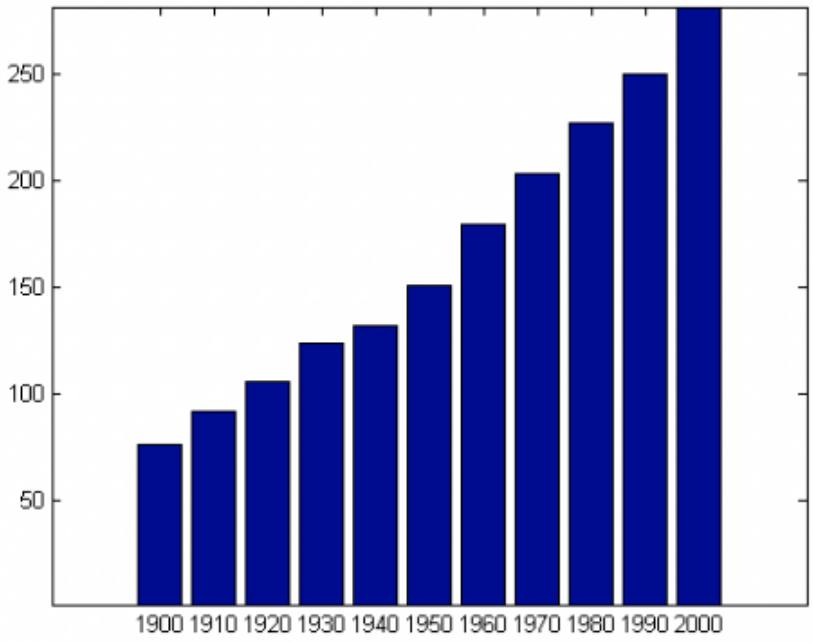
The Law of Continuity

The Law of Continuity focuses on how our minds see visual flow and patterns, rather than broken up shapes or angles. Our minds naturally continue these patterns, like in the example below where we see two lines crossing each other, not two angles meeting.



The Law of Common Fate

This states that objects with the same direction of movement are perceived to be a collective entity. Let's look at this example of a graph below. All the bar shapes are of different heights, but because they move in the same direction (upward), we relate them to each other and collectively they tell us something.

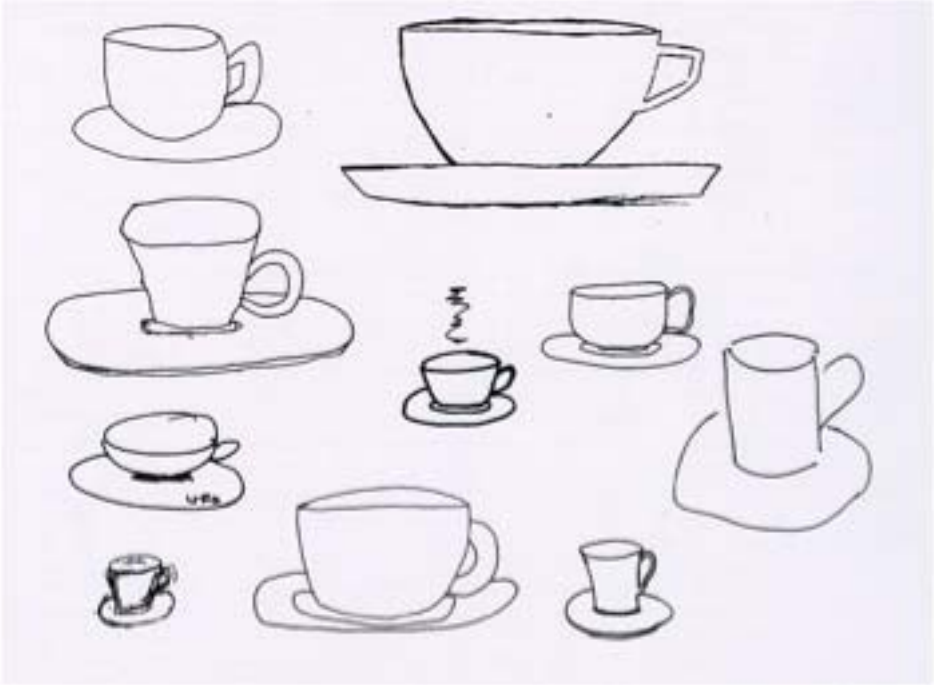


Two other important principles

The following two principles aren't from the Gestalt theory of visual perception, but they also apply to how the visual system in humans work.

Canonical Perspective

The canonical perspective states that most people imagine objects from a similar perspective and that is: from slightly above as if they were looking down at the object, and to the right or left just a bit. Researchers who asked people to draw pictures of common objects and animals found that most people drew these objects from this subtle 3D perspective. Look at this example of coffee cup drawings:



Here you can see that most of the coffee cups were drawn nearly identical and all from the 'canonical perspective'. But why would that be? Surely we don't see every object from the same perspective. Take a house for instance- you normally see a house from below as it is taller than you, yet people would still draw a house from a canonical perspective or straight-on. The reason for this is that a) we imagine objects in the canonical perspective in our minds and b) this perspective gives maximum information about the visual depth of an object. Interesting. So this view helps us understand the size and depth of an object.

Affordance

James J. Gibson, a Gestalt psychologist, coined the term visual 'affordance' after he studied the affects of gradients and textures on the human retina and found out that they allow us to have better depth perception. When we can see the gradients, shadows and textures on an object, we can understand its size, weight, depth and overall form.

How does all of this affect web design?

Simply, humans first see the whole form and then notice the details and small changes. If your site has an "unpleasing" form, the design will never be taken to. This means that as designers we

shouldn't start designing the navigation, or the sidebar, or how our buttons will look– but first design the overall shape and then fill in the contents as we go. When creating webpages within a site, if you want to maintain a feeling of continuity and relationship, then the overall form of each page design should remain virtually the same, otherwise the website as a whole will feel disjointed.

The visual Gestalt principles we can take away for web design are:

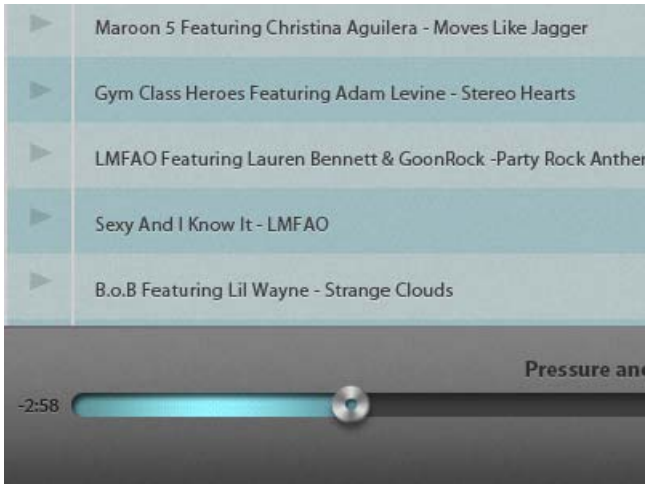
- People observe the entire design/form first, and then dive into the details. Think about the flow of the overall look of the site, rather than obsessing over minute details.
- Similar colours help suggest relationships. Maintain colour schemes throughout similar items and functions.
- Similar shapes help suggest relationships. Make items that do similar things look the same in form.
- Grouping of objects helps us identify relationships. Organising items with adequate proximity and negative space around them helps us clearly see the connection of those items.
- Patterns are easily noticed by and continued in our minds. Creating similar steps in a workflow or visual affordances with interaction will allow users to notice the pattern and remember it for future interaction.

I'm sure most of those principles seem pretty obvious to the average designer, or just an average person, and that's because these principles are so innate to our human perception of the world, that not doing these things would seem unnatural or somehow chaotic.

In regards to web design and the canonical perspective, create icons and shapes from the canonical view as people will be familiar with that representation of the thing, and they will be able to read as much information as possible about it from the angle. Do this especially if it is a new brand or product where people aren't familiar with your logo.

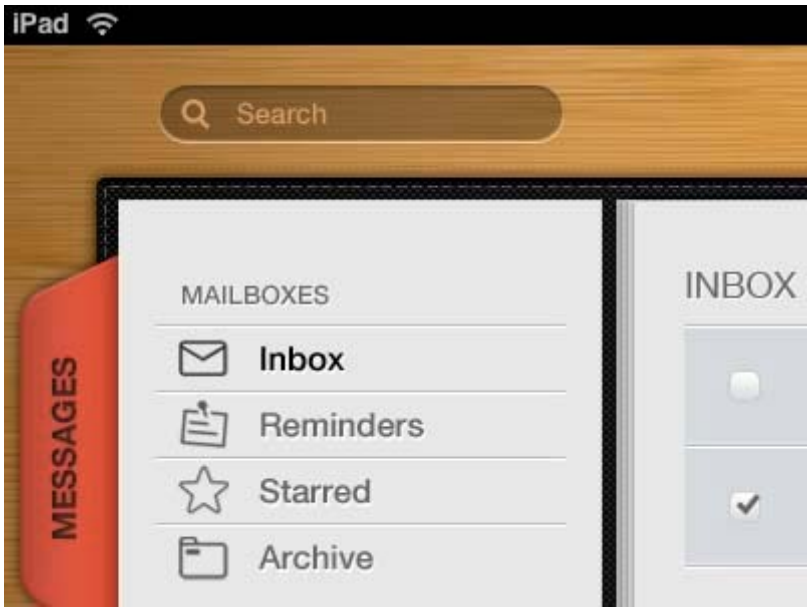
In this icon above, we can tell that it is a book because of the canonical perspective. We can also see the subtle A-Z engraving and the bookmark falling down thanks to the angle, which suggests that this is either an address- or phone-book application.

Visual affordance is crucial to "good" web design because it helps people relate to the web. By bringing the 3D aspect of the real world that humans are familiar with to the 2D realm of the web, it feels more familiar and we more intuitively know how to interact with it.



Sliders are a concept we are familiar with in the real world since the 20th century. We understand holding something and dragging it. The gradients on the progress section of the bar hint at real-world lighting and suggest position of lighting (from above casting downward shadows). The deep shadow at the top of the bar suggests this is cut into the grey area and tells us about the object's depth.

These buttons have dark shadows to suggest they are quite chunky and to really urge you to want to press them. Similarly the dial knob has a long shadow which hints at its height. The glossy metallic texture is familiar to the real world and how light hits metal. The engravings of the icons also hint at industrial processes for carving shapes in materials. Finally the subtle texture on the background mixed with the central gradient shows us where the light is focused.



And finally in this example it brings to mind textures we are very familiar with throughout human history: paper and wood. The subtle gradient on the paper suggests it is a non-shiny, flat object. The sharp lines separating the mailboxes suggest scoring. The shadow behind the paper shows the tab is lurking there under. The stitching in the journal makes this have a very real-life feel to it (as if you needed stitching to keep together a virtual notebook). Finally, the search box looks as if it's been shallowly etched into the wood. Overall it's something we can relate to.

Conclusion

I hope you enjoyed this second part, but more so I hope you enjoyed this article as a whole (see, I went all "gestalt" on you). I hope that throughout these ten points you've learned some fundamental truths about how the human mind works and picked up a few ideas on how we can pay heed to this when designing for the web. I've certainly enjoyed writing this article, and this won't be the last of me when it comes to writing about UX and the human mind, so check back soon my blog http://vcareyux.wordpress.com/ for more! Thanks for reading."

Sources

- Change Blindness http://bit.ly/pEuPhE
- You Look Where They Look http://bit.ly/qPufmb
- You Have Inattention Blindness http://bit.ly/nJySa9
- Your Attention is Riveted by Pictures of People http://bit.ly/obkuvL
- Food, Sex or Danger http://bit.ly/mRnTCN
- Emotions Affect Cognition http://bit.ly/oGy9pF
- Unconscious Mind Wiki http://bit.ly/SvIFO
- 'Blink: The Power Of Thinking' book http://amzn.to/u8Vyhh
- Unconscious Mind http://bit.ly/srgoEo
- Consciousness, The Brain's WiFi System http://bit.ly/18gHWu
- Reasoning is More Intuitive Than We Think http://bit.ly/qdx866
- People Create Mental Models http://bit.ly/elQdv6
- The Secret to Designing an Intuitive UX http://bit.ly/u4H5EL
- Wiki Mental Models http://bit.ly/16cfXw
- Mental Models http://bit.ly/b4joDt
- Mental Models and Usability http://bit.ly/jJNZg1
- Design Theory For Web Designers http://bit.ly/dVWo4F
- Gestalt Theory http://bit.ly/tR1VOD
- James J. Gibson http://bit.ly/tsuUEX
- Canonical Perspective http://bit.ly/1oO7ua

Cucumber

Richard Lawrence, Humanizing Work, http://www.humanizingwork.com/

Cucumber is a tool to support Behavior Driven Development with plain text specifications and unobtrusive automation in Ruby. Alternative implementations of Cucumber exist for Java, .NET, and several other platforms.

Web Site: http://cukes.info/
Version tested: 1.1.2
System requirements: Windows, OSX, Linux with Ruby 1.8.7 or later
License & Pricing: Free, MIT license
Support: Issue tracker and wiki at https://github.com/cucumber/cucumber, mailing list at http://groups.google.com/group/cukes

Bertrand Russell once wrote, "Everything is vague to a degree you do not realize till you have tried to make it precise." Herein lies one of the core problems of software development. Developing software is fundamentally an exercise in making the vague and unknown - the stuff of wishes, ideas, and conversations - sufficiently precise to make a machine behave properly.

Behavior Driven Development (or BDD) is an approach to software development designed to address just this problem. How do we discover what our software ought to do, specify it clearly, and validate that the software does and continues to do what we intend? With BDD, we begin development of each feature with the desired new behavior, specifying that behavior with concrete examples, and making those examples executable as automated tests. Only after we have a failing test - a desired behavior currently unsatisfied by the system - do we consider implementing the behavior.

When Dan North originally proposed BDD, it was an answer to issues he had doing and teaching Test-Driven Development. At this time, the focus was at the class and method level. Later, BDD grew to encompass requirements and analysis, and the emphasis moved to describing behavior at the system level. (See http://dannorth.net/introducing-bdd/ for more BDD background.)

Today, there are several tools that support BDD. I have used and taught a handful of them, including FitNesse, JBehave (the original BDD tool), Concordion, and Cucumber. Of the BDD tools I have worked with, Cucumber is by far my favorite.

Cucumber was originally a Ruby tool. It grew out of the Ruby unit-level BDD framework rspec. The Ruby version of Cucumber is still the reference implementation and the most widely used, but there are now versions for Java, .NET, JavaScript, as well as several other languages and platforms. I'll introduce the Ruby version and then briefly compare it with the Java and .NET versions.

Installation

Cucumber is installed with Ruby's package manager, RubyGems. Assuming you already have a current version of Ruby (1.8.7 or 1.9.3 as of this writing), to install Cucumber simply open a command window and run

```
gem install cucumber
```

This will install Cucumber along with its dependencies.

Note: If you intend to use Cucumber under Ruby on Rails 3.x, you'll want to install Cucumber using Bundler instead. See the Cucumber wiki for details.

Features and Scenarios

Specifications written for Cucumber have two parts: feature files containing scenarios expressed in the Gherkin language and Ruby files containing unobtrusive automation for the steps in the scenarios.

Feature files begin with a feature title and description:

```
  Feature: Medical Provider Search
    In order to avoid paying out-of-network fees for
    medical care, insurance policy holders (aka members)
    want to find medical providers covered by their
    insurance for particular specialties, locations, etc.
```

This would be followed by a number of scenarios to elaborate how the feature ought to behave. Scenarios are generated as product people, developers, and testers discuss the feature. They'll ask questions like, “What's an example of a simple search?” “What's the next most important example?” and “What kind of change to the context would produce different results from that same action?”

Teams typically discuss scenarios in natural language first. Then, they express the scenarios more precisely using Gherkin’s scenario structure:

```
  Given <some initial context>
  When <an event occurs>
  Then <ensure some outcomes>
```

Each line in a scenario is called a *step*. There may be more than one step of a particular type, in which case the keyword `And` or `But` is used instead of `Given`, `When`, or `Then`. If no initial context setup is required, there may be no `Given` step.

After we add a few scenarios, the feature for the Medical Provider Search might look like:

```
  Feature: Medical Provider Search
    In order to avoid paying out-of-network fees for
    medical care, insurance policy holders (aka members)
    want to find medical providers covered by their
    insurance for particular specialties, locations, etc.

    Background:
      Given I'm logged in as a member

    Scenario: Policy Holder Can Get to the Search Form
      When I click "Find a Provider"
      Then I should see the provider search form

    Scenario: Empty Search
      Given no available providers
      When I search without specifying any search criteria
      Then the results should indicate, "No matching providers
found"
```

```
  Scenario: Search by Specialty and Location
    Given the following providers:
      | Name  | Provider Type | Specialty        | ZIP   |
      | Jones | Doctor        | General          | 90010 |
      | Smith | Doctor        | Endocrinology    | 90010 |
      | Khan  | Therapist     | Physical Therapy | 90010 |
      | Cho   | Doctor        | Endocrinology    | 80113 |
    When I search for a provider with the criteria:
      | Provider Type     | Doctor        |
      | Specialty         | Endocrinology |
      | ZIP               | 90010         |
      | Search Radius     | 5 miles       |
    Then the results should include only provider Smith
```

The `Background` section describes any common context to be established before each scenario. The first scenario doesn't need any additional context, so it only includes `When` and `Then` steps, while the other scenarios set up context regarding the available providers. The third scenario contains more complex data; it uses tables to structure that data in a readable way.

Step Definitions

Cucumber scenarios become automated tests with the addition of what are called *step definitions*. A step definition is a block of code associated with one or more steps by a regular expression (or, in simple cases, a string). Two step definitions for the scenarios above might look like this:

```
Given "I'm logged in as a member" do
  visit home_page
  fill_in 'Username', :with => 'testmember'
  fill_in 'Password', :with => 'Passw0rd'
  click_button 'Sign in'
end

Given /^the following providers?:$/ do |providers_table|
  Provider.delete_all
  providers_table.hashes.each do |row|
    Provider.create :last_name => row['Name'],
      :provider_type =>
        ProviderType.find_by_name(row['Provider Type']),
      :specialty => Specialty.find_by_name(row['Specialty']),
      :zip => row['ZIP']
  end
end
```

The body of each step definition is just Ruby code. The first example uses the web application driver library Capybara to interact with a web page. The second example uses Rails' ActiveRecord to set up particular providers in the database. We could just as easily run a command line application, work with the file system, or call a REST service if we needed to.

Capture groups in the regular expression become arguments to the step definition. For example, if we wanted to extend the first step definition to support multiple roles, we might modify it to look like the following:

```
Given /^I'm logged in as an? (.+)$/ do |role|
  credentials = {
    'member' => {
      :username => 'testmember', :password => 'Passw0rd'
    },
    'admin' => {
      :username => 'testadmin', :password => 'secret123'
    }
  }
  unless credentials.keys.include?(role)
    throw "Unknown role: #{role}"
  end

  visit home_page
  fill_in :username, :with => credentials[role][:username]
  fill_in :password, :with => credentials[role][:password]
  click_button 'Sign in'
end
```

Running Scenarios and Integrating with Other Tools

Automated tests are only useful if you can run them. The main interface for running Cucumber scenarios is the `cucumber` command line executable that installs with the Cucumber gem.

Like many Ruby tools, Cucumber is opinionated: it assumes you organize your features, step definitions, and supporting code in a particular way. You can override the defaults, but life is easier if you don't. The standard directory structure is:

> **`features`** - Contains feature files, which all have a .feature extension. May contain subdirectories to organize feature files.
>
> **`features/step_definitions`** - Contains step definition files, which are Ruby code and have a .rb extension.
>
> **`features/support`** - Contains supporting Ruby code. Files in `support` load before those in `step_definitions`, which makes it useful for such things as environment configuration (commonly done in a file called `env.rb`).

In a command window, simply navigate to the directory above your `features` directory (your project directory in a Rails application) and run `cucumber` to use the default options. Cucumber will attempt to run every scenario in every feature file in the `features` directory, looking for matching step definitions in `step_definitions`. When it finds a match, it will execute that step definition. When it can't find a match, it will suggest code you could use to create a matching step definition. Passing steps are colored green, failing steps red, and undefined and pending steps yellow. When a step fails or is undefined, Cucumber skips to the next scenario and colors the skipped steps cyan.

You can specify many options on the command line beyond the defaults. For example, suppose you're working on a story that includes scenarios in two feature files. You might tag those scenarios with `@current` to say, "These are the scenarios I'm currently working on." On the command line, you could run `cucumber --tags @current` to run just those scenarios. Another example: You might want to run Cucumber with different output. You could specify `cucumber --format html --out output/report.html` to get a nice looking HTML report. Or you might use `--format junit` to get a JUnit-style report to integrate

with a continuous integration server. There are too many options to list here; run `cucumber --help` to see them.

While most users run Cucumber from the command line, Cucumber does integrate with editors such as TextMate and JetBrains RubyMine. Both the TextMate bundle and RubyMine plugin provide syntax highlighting, formatting, code completion, and the ability to run features and scenarios inside the editor.

Why I Like Cucumber

Cucumber is optimized for BDD. It supports a particular set of interactions between team members and stakeholders. It's not optimized for testing (though I think it does that part of its job sufficiently well). When I see explanations of new tools aiming to “fix Cucumber's problems” (e.g. Spinach and Turnip in recent weeks), I note that what the authors consider Cucumber's problems are often things I consider Cucumber's strengths. I infer that they want Cucumber to be a different kind of tool for a different kind of purpose.

So, what do I like about Cucumber?

Separation of examples and automation - Cucumber's unobtrusive automation means product people are more likely to engage with features and scenarios than if code were mixed in. Whether or not product people actually write scenarios, they should be able to read, verify, and adjust them.

Some structure but not too much - The Given-When-Then syntax of Cucumber scenarios imposes some structure on scenarios while leaving plenty of room for teams to grow their own language to describe their system. Similarly, the use of regular expressions for mapping between steps and step definitions leaves you room for flexibility in language but naturally limits how much flexibility you employ, lest the regular expressions become unreadable.

Pressure to grow a ubiquitous language - One of the apparent weaknesses of Cucumber as a test automation tool is that step definitions are all global. This quickly reveals, via ambiguous step matches, whether you use the same words to mean more than one thing in your domain. This pressure, plus the precision of specification by concrete examples, helps a team grow a precise language to express their domain.

Just enough syntax - The Gherkin language walks a fine line between natural language and a programming language. The Background and Scenario Outline features support simple refactoring to remove excess duplication. But more complex structures such as procedure calls and includes are left out because they would hurt readability for non-programmers.

Potential Drawbacks

Extra overhead - Compared to writing tests in a general-purpose language like Ruby, Cucumber introduces extra overhead. If you're not going to have the BDD-style interactions, if you're not going to have non-programmers read scenarios, and if you're disciplined enough to use domain language in your tests, you may prefer to write functional tests in test/unit or rspec.

Regular expressions - I consider regular expressions to be a strong point of Cucumber, but many people, even developers, are uncomfortable with them. If you use Cucumber, you'll find it hard to stay away from regular expressions, so this may be a reason to choose a different tool. To overcome this, I wrote an article and cheat sheet highlighting the most useful subset of

regular expressions for Cucumber users. You can read this article on my blog at http://www.richardlawrence.info/2011/08/23/cucumber-regular-expressions-cheat-sheet/.

Cucumber for Other Platforms

Cucumber projects are available for other platforms beyond Ruby. Some use Ruby Cucumber with a bridge into the target language (e.g. cuke4php and cuke4lua). Others use the Gherkin parser but implement everything else in the target language.

For the Java platform, cucumber-jvm is a pure Java implementation of Cucumber. As of this writing, it's still under development. While cucumber-jvm fully supports Gherkin, it's missing many of the runtime features in Ruby Cucumber. See https://github.com/cucumber/cucumber-jvm.

Features and scenarios in cucumber-jvm look exactly the same as shown above - they still use Gherkin. A step definition in Java looks like this:

```
@Given("^the following providers?:$")
public void createProviders(cucumber.table.Table providersTable)
{
  // create the providers in Java
}
```

Various other JVM languages are supported. The same step definition in Groovy, for example:

```
Given(~"^the following providers?:$") {
  cucumber.table.Table providersTable ->
    // create the providers in Groovy
}
```

Cucumber-jvm scenarios can be run from the command line or with JUnit.

SpecFlow is a pure .NET implementation of Cucumber, again based on the Gherkin parser, with integration into Visual Studio 2008 and 2010. See http://www.specflow.org/. Step definitions in C# look much like those in Java:

```
[Given(@"^the following providers?:$")]
public void CreateProviders(SpecFlow.Table providersTable)
{
  // create the providers in C#
}
```

Scenarios in SpecFlow are generated into NUnit or MSTest unit tests in the background and run with whatever unit test runner you prefer. Many SpecFlow users use the Visual Studio unit test runner provided by JetBrains ReSharper.

Because Cucumber tests typically interact with the target application out-of-process and because the Ruby language and libraries are nice to work with, many teams use the Ruby version of Cucumber to test Java or .NET applications.

The Bottom Line

Teams using or considering the approach variously known as Behavior Driven Development, Acceptance Test Driven Development, or Specification by Example should look at Cucumber. It supports this approach better than any tool I have used. Teams simply looking for a functional test automation tool may find that other, more technical test frameworks fit their needs better.

Sureassert Exemplars: Unit Testing Without Unit Tests

Nathan Dolan, nathandolan @ sureassert.com, http://www.sureassert.com/uc/about-us/

Sureassert UC is a new multi-faceted annotation-driven tool that seeks to cut the expense, increase the effectiveness and simplify the process of unit testing for Java projects. It introduces specification-driven, declarative tests called Exemplars along with the capability to perform continuous test execution and coverage reporting within the Eclipse IDE.

Web Site: www.sureassert.com
Version tested: 1.3.0
System requirements: Any Eclipse 3.4+ distro
License & Pricing: Open source (Annotations Library); free (Engine for Eclipse)
Support: via forums, guide and specs at www.sureassert.com

Unit Testing and the Importance of Specification

Unit testing applied correctly reduces delivery risk and increases maintainability by providing a regression safety net. Having been standard practice for some considerable time however, writing comprehensive unit test suites (typically in JUnit) remains an expensive exercise and one that is often poorly applied in the real world. For the purposes of the article, it's important to differentiate the term "unit tests" from longer-running tests that depend on the presence of external resources (such as integration tests and functional tests). The term refers to the smallest testable part of an application, i.e. in Java, isolating and testing methods.

Writing poorly named classes full of undocumented, poorly named methods introduces maintainability issues. Establishing an effective project vocabulary and semantic model is important. Building on this, methods should include a basic specification that describes what the method does (but not how), what it returns, what parameters it expects and the exceptions it may throw. Additionally it may describe pre and post conditions of the method, invariants, and it may summarise example usage scenarios. This is commonly achieved using javadoc.

If you doubt the importance of specifying classes and methods, step back and consider how pleased you would be if all the APIs on which your code was dependent (like the JDK, Spring, Commons Utils, etc) had their class and method specifications (javadocs) removed. This is not just about readability, it is about reducing complexity by allowing the reader to abstract and conceptualize, which makes code more extensible and maintainable thereby reducing cost and risk of change.

How does this relate to unit testing? Generally, unit tests test methods. If a method doesn't have a specification, it is perhaps valid to ask what the test is actually testing. Unit tests should assert that the *contract* of each method holds under varying conditions. Ideally, the *contract* between class/method and consumer is that the consumer adheres to the *specification*. If no contract is defined for a method, consumers may use it as they please and subsequently the results are undefined. Asserting a method works without defining the boundaries of what "works" means has little value other than in trivial software that has few consumers or inter-dependencies.

As a brief aside: It has been claimed that unit tests themselves can form the specification. Confusing what can sometimes work for user acceptance testing with unit testing is a mistake. Consuming an API with reference to its suite of unit tests does not help abstract or conceptualize: it just gives the consumer unnecessary puzzles to solve. However, unit tests can form *part* of the specification. At this point let's introduce Sureassert *Exemplars*.

An Exemplar encapsulates a single test thread targeted at a specific method and enables *Declarative Unit Testing.* Traditionally, Java unit tests are written as classes containing methods that invoke a "method under test". The test code fundamentally does three things:

1. **Create or otherwise retrieve an object** (the "object under test"),
2. **Execute a method** on the object under test with typical inputs, and
3. **Assert one or more expected results** on the method's returned value, thrown exception, the object under test's state, and/or the state of some other affected object.

It typically might also:

1. **Perform some initialization and/or clean-up**; although this is more typical and necessary within *integration* and *functional* test classes rather than *unit* tests as these more often deal with resources external to the object under test;
2. **Setup stubs or mocks** to force external dependencies to behave in a manner determined by the unit test, and
3. **Assert the behaviour** of the method under test, e.g. what methods have been invoked by the method under test and in what way.

This being the case, why code it? Why not declare the object under test, test inputs and expected results as part of the method specification itself? After-all, providing example execution scenarios often forms part of the worded specification. For example, take the `String.substring` specification:

```
public String substring(int beginIndex, int endIndex);
```

Returns a new string that is a substring of this string. The substring begins at the specified beginIndex and extends to the character at index endIndex - 1. Thus the length of the substring is endIndex-beginIndex.

Examples: "hamburger".substring(4, 8) returns "urge"

"smiles".substring(1, 5) returns "mile"

Parameters:

beginIndex the beginning index, inclusive.

endIndex the ending index, exclusive.

Returns: the specified substring.

Throws: IndexOutOfBoundsException - if the beginIndex is negative, or endIndex is larger than the length of this String object, or beginIndex is larger than endIndex.

Introducing Exemplars

Let's use Sureassert Exemplars to show the same thing. To define them, annotate the method-under-test:

```
@Exemplars(set={
 @Exemplar(instance="'hamburger')", args={"4", "8"}, expect="'urge'"),
 @Exemplar(instance="'smiles'", args={"1", "5"}, expect="'mile'") })
public String substring(int beginIndex, int endIndex);
```

Each Exemplar attribute takes a string, but it is parsed by Sureassert as a *SIN expression*. These match Java very closely, the main notable differences are the use of ' instead of " (as escaping /" wouldn't be pretty), and the ability to use *SIN types*. These are prefixed special types designed to simplify object creation.

For instance to create a new map of strings to integers for use as an argument, you could specify: args="m:'k1'=2,'k2'=4,'k3'=8". Expressions can get field values (including private fields), refer to the object-under-test or parameters of the method-under-test, and make any constructor or method call (including private methods) on any object or class. This means that if you need to, you can do things like build test data and assert expected results in test utility code called from an Exemplar.

Returning to the example, we could also assert the exception scenarios:

```
 @Exemplar(instance="'negtest'", args={"-1", "5"},
           expectexception="StringIndexOutOfBoundsException"),
 @Exemplar(instance="'negtest'", args={"1", "8"},
           expectexception="StringIndexOutOfBoundsException"),
 @Exemplar(instance="'negtest'", args={"3", "2"},
           expectexception="StringIndexOutOfBoundsException")
```

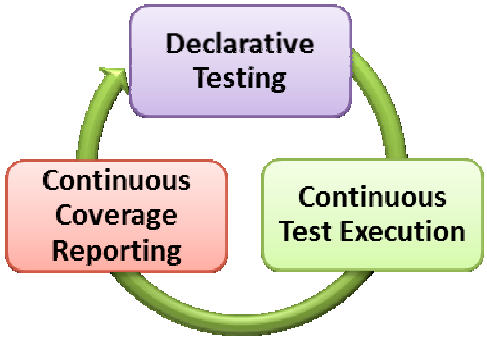
Most Exemplar properties have shorthand alternatives for brevity:

```
 @Exemplar(i="'negtest'", a={"-1", "5"},
ee="StringIndexOutOfBoundsException")
```

This is a very simple example. There are 16 exemplar properties as of version 1.3.0 providing a wealth of testing options including stubbing, behaviour verification, naming, setting up, tearing down, default class generation, templating and more. There are also several other annotations for doing things like integrating JUnits and defining "test double" classes. Crucially, you can define Exemplars on interfaces which are then inherited.

How are Exemplars run? Simple: just install Sureassert in Eclipse, enable it on your project and save your code (Ctrl+S). The Sureassert engine integrates with the Eclipse build process to execute all those Exemplars and integrated JUnit tests that are affected by the code changes you've made. Errors are reported just like compilation errors, except using a purple rather than a red cross. This includes nested dependencies on methods and field values.

Effectively, you never need to run tests: those that need to be, are run whenever you change anything. On top of this, Sureassert also includes integrated test coverage reporting which is also attached to the Eclipse incremental build process.



Let's move on to a discussion on test isolation, and how using Sureassert can help prevent "design for testability" from being a negative influence on your software architecture.

Test Isolation and Designing for Testability

There are two broad camps with regard to unit testing style and how best to isolate the code-under-test from its dependencies. A behaviour-driven approach dictates that unit tests should assert how the method interacts with other code, and that state "becomes a side-issue". A state-assertion approach dictates that tests should assert the state of affected objects and that the mechanism by which this state was reached is unimportant.

It's unlikely either approach is better than the other across all types of software. Regardless of approach, effective unit tests assert that the behaviour of the method-under-test matches its contract, under varying conditions. For example, let's say a method's contract states it changes a given external object's state (say it adds a Message to a notional Queue).

The test can assert that the object's state is changed (the Queue now contains the Message), or it can assert that a method is invoked on the Queue in a way that is known by virtue of its own contract to have this effect (e.g. addMessage was called on the Queue with the Message). The latter has indirection through a dependency on another method's contract, but enables mocking to simplify isolation. The former is direct and can simplify the test but requires dependencies are replaced with stubs which complicates establishing the testing context.

Advocates of a behaviour-driven style would prefer contracts that define interactions, not state changes, in order to reduce the indirection. Those in the other camp might say the opposite. Either of these decisions might be seen as dictating software architecture just to fit a unit testing approach. Regardless of the approach taken, for non-trivial software what is most important is that there are specifications acting as code contracts, they are exercised by the tests, and no "assumptions" are made about what a method does.

The problem with considering "design for testability" in isolation is that non-functional aspects play-off against each-other. It would not be prudent to design overly for testability at the greater expense of maintainability or delivery cost.

Let's introduce how using Sureassert can help prevent testability being a negative influence on your software architecture by simplifying test isolation. The screen-grab below shows some code in Eclipse with Sureassert enabled:

```
/**
 * A game that is played and scored several times. At the end of the game the
 * highest score is registered with the game leaderboard service.
 */
public class Game {

  /** The userID of the user playing this Game */
  private String userID;

  /** The remote leaderboard service for registering scores */
  private GameLeaderboardService leaderboardService;

  /**
   * Creates a new Game for use by the given user.
   *
   * @param userID The userID of the user playing this Game
   */
  @Exemplar(name="game1", args={"'user1'"},
      stubs="ServiceBroker.getServiceImpl=GameLeaderboardServiceImpl/leaderboard")
  public Game(String userID) {

    this.userID = userID;
    this.leaderboardService =
      ServiceBroker.getServiceImpl(GameLeaderboardService.class);
  }

  /**
   * Registers the highest of the given scores with the leaderboard service.
   *
   * @param scores The non-null non-empty list of scores achieved by the user
   * @return The new leaderboard position of the user, or -1 if unknown
   */
  @Exemplars(set={
  @Exemplar(instance="game1", args={"null"},
      expectexception="IllegalArgumentException"),
  @Exemplar(instance="game1", args={"1:13,5,17"},
      verify="GameLeaderboardServiceImpl.registerScore('user1',17)") })
  public int registerHighScore(List<Integer> scores) {

    if (scores == null || scores.isEmpty())
      throw new IllegalArgumentException("scores must contain at least 1");

    int highScore = 0;
    for (int score : scores) {
      if (score > highScore)
        highScore = score;
    }

    try {
      return leaderboardService.registerScore(userID, highScore);

    } catch (CommsException e) {
      showMessage("Sorry, your high score of " + highScore + " could not be registered");
      logError(e);
      return -1;
    }
  }
```

Exemplars in the Real World

In the example above we have a Game class that registers the highest of a given list of scores with an external GameLeaderboardService that is retrieved from a ServiceBroker class defined by the project (not shown). It hasn't really been designed for testability – the ServiceBroker has a static method and isn't injected. While it may be better to inject a singleton ServiceBroker

instance, this may not be desirable or even possible. We shouldn't have to make a decision like this simply to achieve testability. With Sureassert we can easily stub or mock the call; in fact there are several different techniques available for doing so.

We place an Exemplar on the constructor and define a *method stub* that replaces the ServiceBroker.getServiceImpl call with a *named instance* of GameLeaderboardImpl called "leaderboard". The *named instance* will have been created by another Exemplar defined in GameLeaderboardImpl (not shown). We give our Exemplar the name "game1", which assigns the object created by the constructor this name (i.e. we create another *named instance* within our testing context). By testing the constructor, we create an instance that can then be used by other Exemplars. Sureassert manages dependencies between these named instances automatically and understands when it needs to re-run dependant Exemplars. You can control whether to re-use or create new test instances (just postfix the name with a bang "!").

It's worth pointing out that rather than declaring a method stub, we could have declared a *test double* class of ServiceBrokerFacade which would have automatically replaced the *doubled* class whenever running Exemplars or integrated JUnits. Or we could have used an in-line *source stub* to replace the getServiceImpl code with something else in the testing context. This gives you the flexibility to employ the best test isolation techniques for your project, all integrated into Sureassert and all without writing any test code.

Moving on to the registerHighScore method, we've defined two Exemplars. The first tests the precondition that the list must be non-null by passing null as an argument and asserting that an IllegalArgumentException is thrown via the property *expectexception*. The second uses the l: prefix (*List SIN Type)* to quickly create an ArrayList populated with three Integers for the method argument. It then performs behaviour verification using the *verify* property to ensure that the GameLeaderboardServiceImpl.registerScore method is called with 'user1' and 17 (the highest of scores passed).

Continuous Test Execution and Feedback

You may have noticed the green ticks in the left border – this is how you know the Exemplars have been executed successfully. They appear as soon as the code is saved. Mouse-over them in Eclipse and we'd see details of the tests run. If the tests had failed, we'd have seen purple crosses instead, the mouse-over would show what went wrong, and the file and project would be marked as having errors in the *Package Explorer* and *Problems* views (just like compiler errors). You might have also noticed the three lines of code shaded red in the catch block at the end of the example. This is Sureassert's continuous in-line test coverage reporting at work – they're red because this code isn't covered by any Exemplars or JUnits.

Hopefully this article has provided a useful introduction to Sureassert UC and the concepts that inspired its creation. If you're interested in learning more, there's a full guide and downloads available at the website: www.sureassert.com.

ChiliProject

Felix Schäfer, thegcat.net,
finnlabs, http://www.finn.de/

ChiliProject is a modular web-based project management tool. It offers flexible issue tracking and project planning, time tracking, wiki-based knowledge management, integration with various version control systems, as well as team collaboration and client communication through news and forum systems including support for sending and even receiving emails.

Web site: http://www.chiliproject.org
Current version: 2.5.0
System requirements:

- Ruby (1.8.7 or 1.9.2)
- A database (MySQL, PostgreSQL or SQLite supported)
- An application server capable of running Rails projects (for example Phusion Passenger or Thin)
- Version control system support requires the binaries for the version control systems that should be supported by your installation

License: GNU General Public License v2
Pricing: Free
Support:

- The issue tracker and support forums on chiliproject.org,
- The IRC channel #chiliproject on the freenode network,
- Commercial custom development, hosting and support also available.

A young project with a long history

ChiliProject is a young project introduced in February 2011 as a fork of Redmine, building on the nearly five years of Open Source software development that went into it. Consisting of several former Redmine developers, the ChiliProject team forked Redmine because it felt that Redmine development was too sporadic and uncoordinated. The developer and user communities were not involved much in planning and decisions, which made contributions rather frustrating. Previous efforts to help with community involvement and development coordination before forking remained fruitless.

The ChiliProject team has committed to a fully transparent and open governance and development process aligned with the ideals of Free and Open Source Software, striving to include the ChiliProject community at each step of the development process. The goals of the ChiliProject team are to deliver a modern and extensible project management system built using present tried and tested software components and techniques while providing a more stable software and support lifecycle to individuals and organizations.

You can find more information about the fork on the Why Fork? page on chiliproject.org.

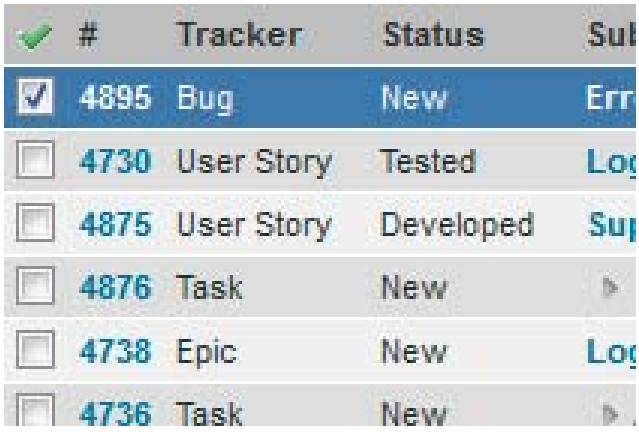
What can ChiliProject do for you?

Issue tracking

ChiliProject offers a powerful and flexible issue tracking system. Issues have a number of default attributes to help you categorize, find and assign them; more custom attributes can be added to individual issue types to store information specific to your needs. The timeline of an issue, similar to a facebook timeline, tracks any change to those attributes (status, assignee, custom attributes, …) as well as comments on the issue itself. The issue comments also are a great place to discuss the issue and keep everything pertaining to it in one place. Users can *watch* an issue to be notified when changes are made to the issue or the issue is commented upon. Finally, a set of issue relations allows you to keep track of related or even blocking issues.

The statuses that an issue can be transitioned to are controlled by the issue workflows, which are dependent on the tracker of the issue (the type of an issue) and the role the user has in the project the issue belongs to. Workflows are defined globally and can be adapted to your needs through total control of the issue statuses associated to a tracker and what status transitions users with a certain role can make.

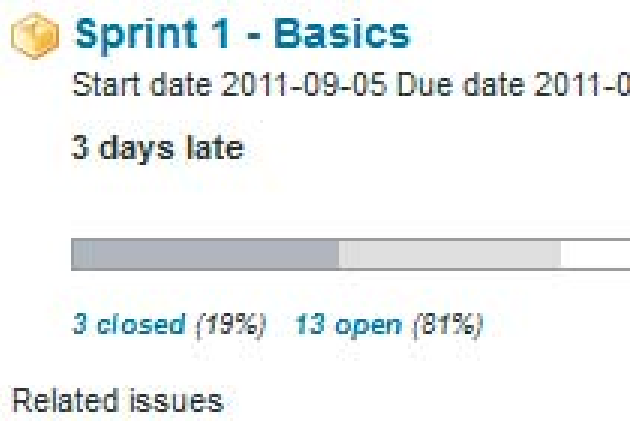
Consider for example a project for a newspaper with the roles *Journalist*, *Assistant*, and *Editor* and a tracker called *Article*. Any user might have an article idea and is able to create a new *Article* with the status *Idea*, which then has to be approved and commissioned by an *Editor* who sets the status of the *Article* to *Approved* and assigns it to a *Journalist*. The *Journalist* might need some information to complete the article and can re-assign the *Article* to an *Assistant*, or she can create a new task in the tracker *Information Request* and set the new task as a blocker of her *Article*. When the *Journalist* has completed the *Article*, she can transition the issue to the status *Written* and assign it back to the *Editor*, who can schedule it for printing once he's finished editing it.



The issue tracking system is completed by a powerful query system giving you an overview of selected issues, which can be filtered by type, assignee, status, any other attribute or custom attribute and any combination of those. The results can then be grouped for a better overview. Queries can be saved for later re-use and can even be shared with your other team members. Many teams use the same saved queries over and over throughout a project to get a standardized point of view and stay in the loop.

Project planning

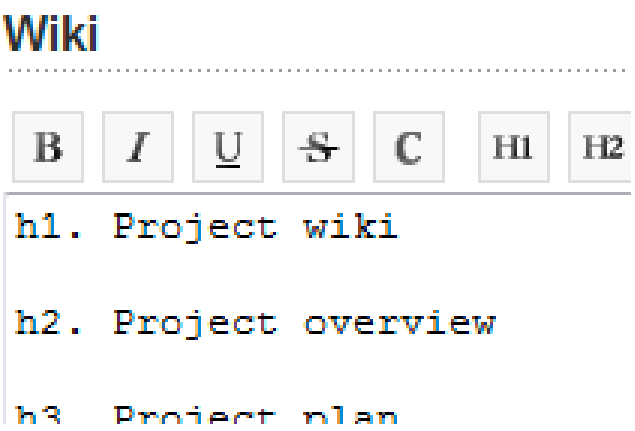
One ChiliProject instance can have many projects, allowing all the teams in your organization to work with the same tool and to connect different projects and issues within them to each other. Projects can be arranged hierarchically and each project has its own set of members and activated modules (issue tracking, wiki, forums, …) which can be managed individually for each project by a user with the role *Manager*.



Project planning is supported by the ability to define versions (milestones) for your project and to see those versions and their completion rate on a Roadmap view. Versions can even be shared between projects to allow for cross-project planning. In addition to the Roadmap view, a Gantt view provides an easy way to get an overview of the advancement of a project, its subprojects, as well as the versions and issues therein.

Knowledge management

Each project comes with its own wiki which is a great place to store and share knowledge. Wiki pages have a history of when and what changes were made and who made those changes in order to make sure you don't loose any information. They can also be linked to each other for knowledge sharing.



In addition to that, the rich wiki syntax is available not only in the wiki pages themselves but pretty much everywhere like issue descriptions and comments, news, forum posts, and so on. The wiki syntax supports shortcuts for often-used links such as links to issues (e.g. *#123* automatically links to issue 123), versions, forum posts, revisions or commits of the project repository and more, as well as wiki macros which can automatically integrate other pages or generate content in your wiki page.

Team collaboration and client communication

For those times the discussion section of an issue doesn't cut it, the discussion isn't specific to any particular issue, or you just want to give your clients access to a discussion space without allowing them access to the issue tracker of the project, ChiliProject has a forum system with support for multiple boards. The news section is a place where you can make announcements, like new software versions or planned maintenance for example.

In complement of the in-site communication features, the email notifications provide users the ability to *watch* (be notified of changes in) single issues, forum threads or wiki pages, a whole forum board or even whole projects. The email system doesn't only send emails though, it can also receive emails and append the email content as a response to the correct issue or forum thread, change the status of an issue or even create new issues.

Source code management and document management

ChiliProject supports software-related projects through the integration of various version control systems, including but not limited to subversion, git and mercurial. The repository browser has support for commit, diff, annotate and history views, and offers the possibility to download individual files from the repository. The rich formatting syntax available in many text inputs throughout ChiliProject offers support for links to files in version control, including linking to a specific line and/or commit. Commit messages can be formatted using this rich syntax too, mentioning an issue in a commit message will even link the commit to the mentioned issue.

Additionally, the version control system integration can be used for less software-related or technical projects. Have you ever received a word document with the name Mission_Plan_v1.3_johns_comments_final.doc? Do you still have an overview of who got the document, who edited it and what version the newest and really final one is? The version control system integration provides you with a clear trail of the different versions your document went through, who made which changes, and the possibility for each member of your team to always have the latest version of the document without having to rely on distributing it via email or tracking the version of the file in the filename.

Expandability

If all those great features aren't enough for you, ChiliProject can be extended by way of plugins. Plugins can add functionality from the most trivial additions, for example changing the label of a core ChiliProject part (you'd rather your "Versions" were called "Milestones"? No problem!), to the most complete modules and the deepest changes (e.g. ChiliProject can be turned from a rather waterfall-centric project management system into one supporting Agile/Scrum

methodologies using the chiliproject_backlogs plugin). While for plugins the sky is the limit, if plugins aren't enough, ChiliProject is Open Source and you can extend and modify it any way you deem necessary (warning: that makes upgrades harder though!).

Summary

ChiliProject is your one-stop solution for all your project management needs. It comes with a rich set of features to support you and your team to achieve even the most ambitious projects and can be adapted to your needs thanks to its modular nature and the possibility to extend the core functionality with plugins. ChiliProject is distributed under the GPLv2, a Free and Open Source License, which grants full access to the source code and thus allows for even the deepest customization. Support is available either from the growing and dynamic community of contributors and users on chiliproject.org or from any of the companies providing commercial support and services on and around ChiliProject.

The ChiliProject developer community is currently hard at work to prepare the release of the next major version of ChiliProject. Version 3 is planned for early next year and will feature a new design with improved usability and preparations for UI additions/refinements in subsequent minor releases, the integration of the liquid templating language in the rich formatting text fields and for wiki macros, as well as many smaller bug fixes and improvements.

Acknowledgements and License

I'd like to thank my colleagues, ChiliProject contributors and friends for their constructive contributions and assistance in writing and proofreading this article: Holger Just (finnlabs), Gregor Schmidt (finnlabs) and Jan Schulz-Hofen (Planio).

Scrum-it

Fabian Uehlin, BSgroup Technology Innovation AG, http://www.bsgroupti.ch

Scrum-it is a virtual Scrum board application that supports the entire Scrum project lifecycle. It reduces project overhead and allows a team to focus on the development of high-quality software products. The software is open source, encouraging users to study, change and improve it.

Web Site: http://www.scrum-it.ch or https://github.com/ti-dev/Scrum-it
Version Tested: Scrum-it 1.0 on Mac OS X 10.7.1 with Apache Tomcat 7.0.21 and MySQL 5.5.15
License & Pricing: CC BY 3.0 - http://creativecommons.org/licenses/by/3.0/
Support: Forums and web site

Introduction

In the past few years, software engineering and the corresponding software development process have undergone significant changes. There are several models to streamline this process. Traditional models, such as the *waterfall model* are still in use, but they can be inflexible and do not necessarily meet today's requirements for developing software applications. *Scrum* is an agile procedural model for software engineering that provides an alternative to traditional approaches. It is characterized by having few rules and reduced bureaucracy. Therefore, the development process can be significantly reduced and the real development goals tend to become more apparent. An essential aspect is the iterative cycle, so-called *sprint.* A sprint usually lasts from two to four weeks. Within each cycle, a part of the software is completed and the end product is built incrementally.

A sprint development team is self-organized. A board is used to list all tasks clearly and make them visible to all team members: *Tasks* are written on post-it notes and attached to the board. Every note represents a defined unit of work, and all tasks must be finished in order to accomplish successfully the goal of a sprint. However, several problems can arise with this type of board and become apparent during the daily project work. Since the data is not available in digital form, archiving or historically preserving information is a challenge. Furthermore, the current project status is limited to the boards' geographical location. Distributed teams cannot participate to the overall process.

Although software-based solutions that fix most common problems are available on the market, those typically desktop-based applications lack the board as the central communicative element. The new Scrum-it application can eliminate the limitations of current software. Its concept combines the advantages of software and modern touch-screen technology, as both conventional PCs and tablet devices (like the iPad) are fully supported.

Installation

The Scrum-it distribution can be downloaded at http://sourceforge.net/projects/scrum-it/files/ and the source code from https://github.com/ti-dev/Scrum-it. The war file (web application archive) needs to be installed into a servlet container like Apache Tomcat. A MySQL database stores the application data. The full installation guide can be downloaded at http://sourceforge.net/projects/scrum-it/files/Scrum-it-InstallationGuide.pdf/download

Once installed, Scrum-it is accessible via modern web browsers, such as Firefox, Safari or Internet Explorer via http://127.0.0.1:8080/scrumit/. Scrum-it is platform-independent and can be used on every operating system with a web browser, without the need for additional software.

Using Scrum-it

Scrum-it has an intuitive user interface and is easy to use, requiring little training. A tutorial is available on YouTube http://www.youtube.com/watch?v=zfcZF1EeY7Q. This article presents how to create a project, sprints and user stories with Scrum-it. It shows also how to use the Scrum board.

Projects & Persons

Scrum-it can manage multiple projects. The left-hand side of the screen shows a "List of Projects". Clicking on a project name inside the list view displays its details in the bottom right content area. To create a new project, click on the "Add" button, complete the form and press "Create Project". If you select a project, a "List of Persons" is displayed on the top right of the screen. Members can be added, edited or removed. Figure 1 shows a list of projects with a pre-selected project named "Scrum-it" and on the right-hand side two members belonging to the project.

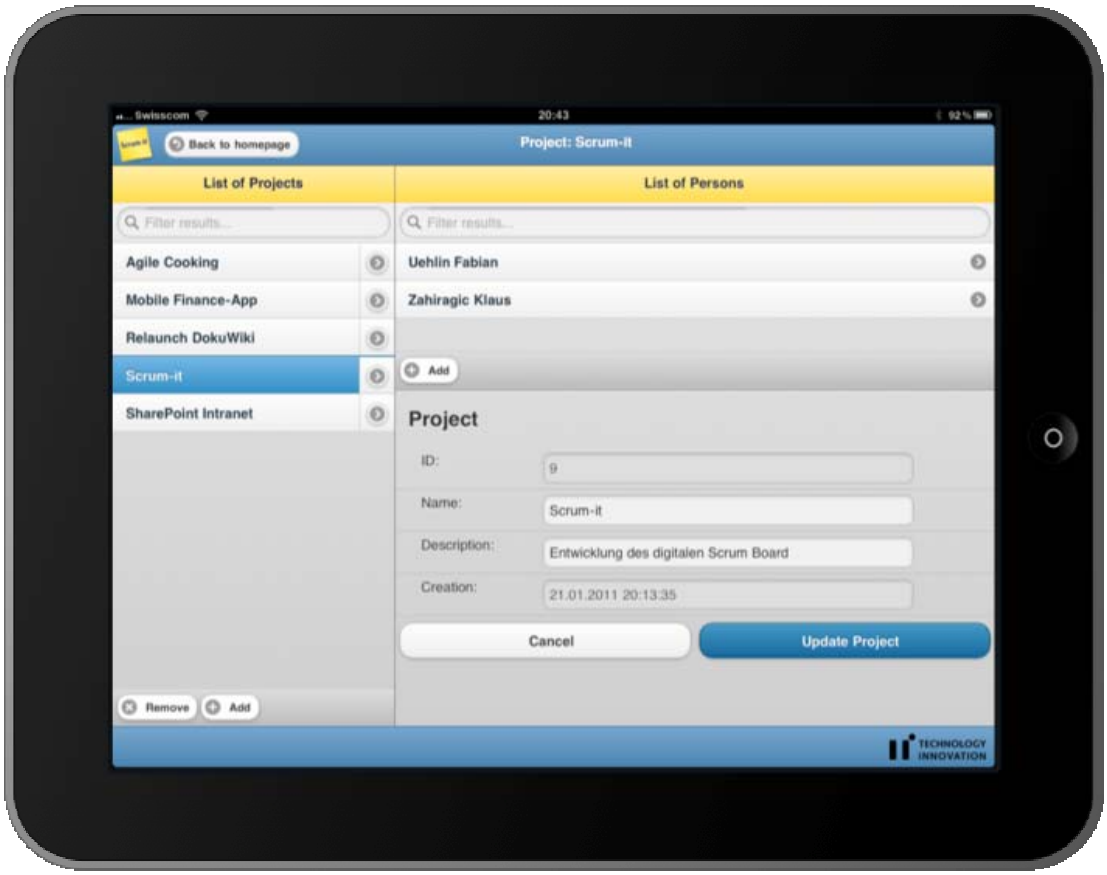


Figure 1 - Project / Person screen on an iPad

Sprints & User Stories

After successfully creating a project, the next step is to set up sprints. A sprint is the main component of a development cycle. The purpose of each sprint is to deliver increments of potentially shippable functionality that adheres to a working definition of "done".

Each project can have multiple sprints. To switch the screen from projects to sprints, click the arrow on the right of the project name. The resulting screen, including the views for sprint and user story, is shown in Figure 2.

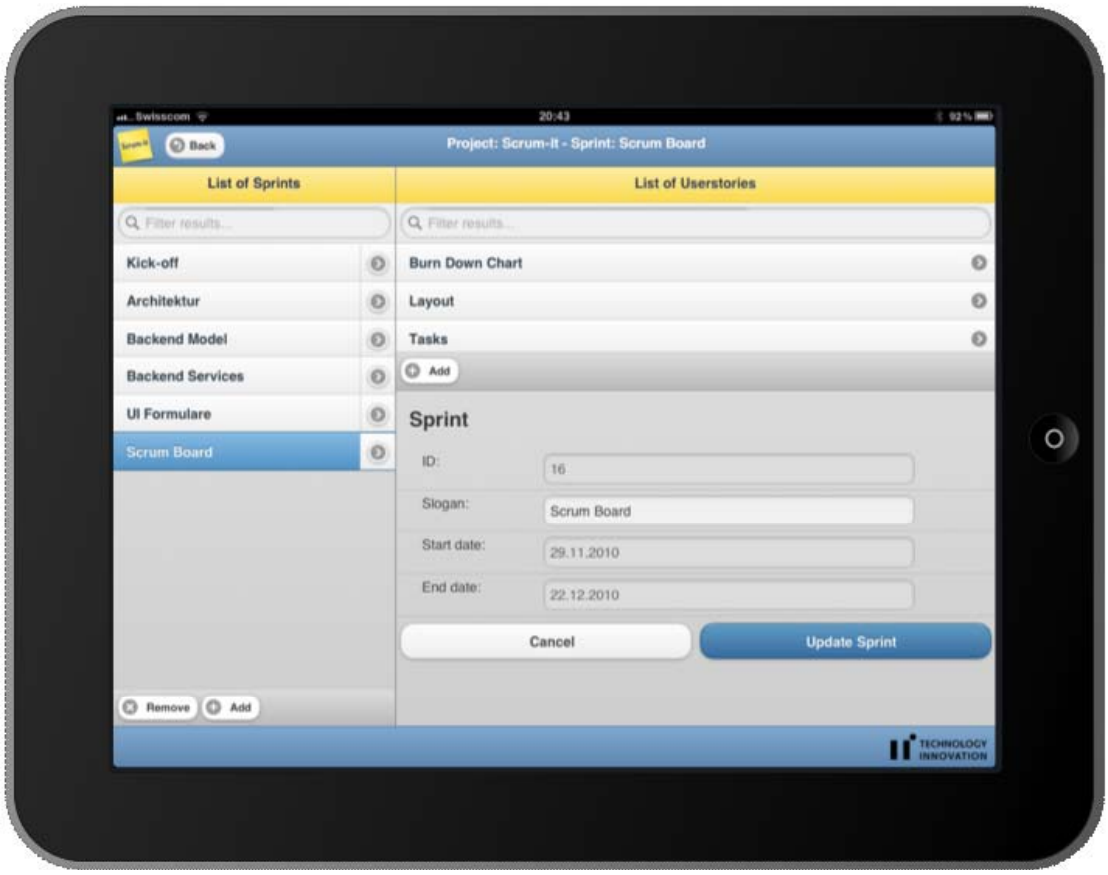


Figure 2: Sprint / User Story screen on an iPad

This layout is the same than for the project screen and it provides identical operations/functions. The left-hand side of the screen shows a “List of Sprints”. A new sprint can be added to a project by clicking “Add” and filling the form with the information for “slogan”, “start” and “end date”.

By selecting a sprint, a “List of User Stories” is displayed. A user story is one or more sentences in everyday language that captures what the end-user wants to achieve. A single sprint can have many user stories that can be added, updated or removed.

Scrum Board, Tasks and Burndown Chart

Each sprint is represented with a Scrum board view. To go to this screen, press the arrow on the right of the sprint name .A new Scrum board screen appears with a Burndown Chart, as shown in Figure 3.

Figure 3 - Scrum Board with Tasks and Burndown Chart on an iPad

On the Scrum board, you can see and change the state of tasks of the current sprint within defined categories. It presents user stories on the left side of the screen, prioritized from top to bottom. Tasks are assigned to the categories “open”, “in progress” or “done” representing the current status. A task is also part of a user story. User story notes have the same color as their associated task notes. A new task can be added to a user story by clicking on the plus icon next to it. Text can be added to describe the new task. You can drag and drop a task to change its status.

The *Burndown Chart* is a graph that displays the remaining amount of work. It will be updated every time the status of a task changes and gives an overview of the sprint progress, with the remaining effort to finish it. In the burndown chart of Figure 3, the orange colored line represents the ideal burndown, the turquoise colored line the actual burndown.

Conclusion and Outlook

Scrum-it was developed by BSgroup Technology Innovation AG at its head office in Zürich, Switzerland. It is based on a Java EE backend system combined with a front-end implemented with HTML5, CSS3 and jQuery Mobile.

The list of features planned for the next release includes the product and sprint backlog and usability improvements. The source code is available at https://github.com/ti-dev/Scrum-it and is published with a permissive license (Creative Commons Attribution 3.0) that can be freely used. The Scrum-it project is currently looking for contributors. Feel free to submit patches to support the project team.

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