METHODS & TOOLS

Practical knowledge for the software developer, tester and project manager **ISSN 1661-402X**
Winter 2013 (Volume 21 - number 4) www.methodsandtools.com

Is Your Project Team a Rock Band or Boys Band?

It all started with the idea that you could compare software development projects and part of the musical world. With maybe a simplistic view about the music industry, I started thinking how different project teams organizations might share the same differences that exist between rock bands and boy bands.

We could compare rock bands to self-organized teams. They are mostly formed by mutual interest, even if they can also "audition" to find missing components. They choose their goals and mostly compose their own music. If you have to judge them by their fame or the ability to make enough money from their music, there is a low success rate for rock bands. This isn't however a problem for the bands that play music mostly for pleasure. Some have one or more leaders that are more visible in the media. There are also people, producers or managers, which help to facilitate the creative process of the band without being formally members. Both producer George Martin and manager Brian Epstein have been nicknamed the "Fifth Beatle". This is not very different from a ScrumMaster role. This type of musical project might have a long life expectancy if they keep having fun, meet their audience and manage to survive the difficulty of long-term interpersonal relationships. Boy bands are more similar to traditional project teams. A music producer forms the band as a "music product". He will pick people for specific ~~looks~~ skills and have a command and control influence on the band. He will choose composers, songs and musical style to shape a band for a target audience. Money is the metric for this type of musical project and the final "product" has often (always?) a short life expectancy. This being said, you might find talented individuals in boy bands that will have then their own successful career like George Michael or Robbie Williams.

Considering these two models, you might think that start-up organizations are more akin to have "rock bands" software development teams and big companies will create only "boy bands" project teams. If this might be generally the case, I don't think this is inevitable. Large companies can also nurture project teams that enjoy some autonomy and freedom to explore. On the other side, some software start-up could also become mostly money-oriented and shift their management style toward a boy band model, which could be the case when venture capital backers are in control and are looking to recover quickly their investment.

[illegible]

Inside

STAR Canada Conference - Click on ad to reach advertiser web site

GREAT SOFTWARE REVOLVES AROUND GREAT TESTING
STAR
CANADA
APRIL 5–9, 2014 / TORONTO, ONTARIO / HILTON TORONTO
REGISTER BY FEBRUARY 7 AND SAVE UP TO $300
STARCANADA.TECHWELL.COM

Simple sketches for diagramming your software architecture

Simon Brown, @simonbrown
http://www.codingthearchitecture.com, http://www.simonbrown.je

If you're working in an agile software development team at the moment, take a look around at your environment. Whether it is physical or virtual, there's likely to be a story wall or Kanban board visualising the work yet to be started, in progress and done. Visualising your software development process is a fantastic way to introduce transparency because anybody can see, at a glance, a high-level snapshot of the current progress. As an industry, we've become pretty adept at visualising our software development process over the past few years although it seems we've forgotten how to visualise the actual software that we're building. I'm not just referring to post-project documentation, this also includes communication *during* the software development process. Agile approaches talk about moving fast, and this requires good communication, but it is surprising that many teams struggle to effectively communicate the design of their software.

Prescribed methods, process frameworks and formal notations

If you look back a few years, structured processes and formal notations provided a reference point for both the software design process and how to communicate the resulting designs. Examples include the Rational Unified Process (RUP), Structured Systems Analysis And Design Method (SSADM), the Unified Modelling Language (UML) and so on. Although the software development industry has progressed in many ways, we seem to have forgotten some of the good things that these older approaches gave us. In today's world of agile delivery and lean start-ups, some software teams have lost the ability to communicate what it is they are building and it is no surprise that these teams often seem to lack technical leadership, direction and consistency. If you want to ensure that everybody is contributing to the same end-goal, you need to be able to effectively communicate the vision of what it is you're building. And if you want agility and the ability to move fast, you need to be able to communicate that vision efficiently too.

Abandoning UML

As an industry, we do have the Unified Modelling Language (UML), which is a formal standardised notation for communicating the design of software systems. I do use UML myself, but I only tend to use it sparingly for sketching out any important low-level design aspects of a software system. I don't find that UML works well for describing the software architecture of a software system. While it is possible to debate this, it is often irrelevant because many teams have already thrown out UML or simply don't know it. Such teams typically favour informal “boxes and lines” style sketches instead but often these diagrams don't make much sense unless they are accompanied by a detailed narrative, which ultimately slows the team down. Next time somebody presents to you a software design focussed around one or more informal sketches, ask yourself whether they are presenting what's on the sketches or whether they are presenting what's still in their head.

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

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

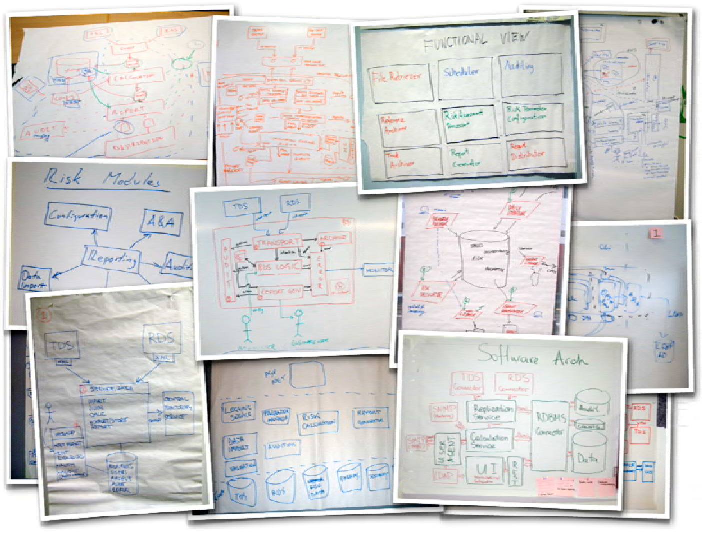
Training the NYC Marathon live website to be the fastest in the world for three years running.

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

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

Load Testing Software & Services

webperformance.COM



Abandoning UML is all very well but, in the race for agility, many software development teams have lost the ability to communicate visually too. The example software architecture sketches above illustrate a number of typical approaches to communicating software architecture and they suffer from the following types of problems:

- Colour-coding is usually not explained or is often inconsistent.
- The purpose of diagram elements (i.e. different styles of boxes and lines) is often not explained.
- Key relationships between diagram elements are sometimes missing or ambiguous.
- Generic terms such as "business logic" are often used.
- Technology choices (or options) are usually omitted.
- Levels of abstraction are often mixed.
- Diagrams often try to show too much detail.
- Diagrams often lack context or a logical starting point.

Some simple abstractions

Informal boxes and lines sketches can work very well, but there are many pitfalls associated with communicating software designs in this way. My approach is to use a small collection of simple diagrams that each shows a different part of the same overall story. In order to do this though, you need to agree on a simple way to think about the software system that you're building. Assuming an object oriented programming language, the way that I like to think about a software system is as follows: a software system is made up of a number of containers, which themselves are made up of a number of components, which in turn are implemented by one or more classes. It is a simple hierarchy of logical technical building blocks that can be used to illustrate the static structure of most of the software systems I have ever encountered. Some diagrams will help to explain this further.

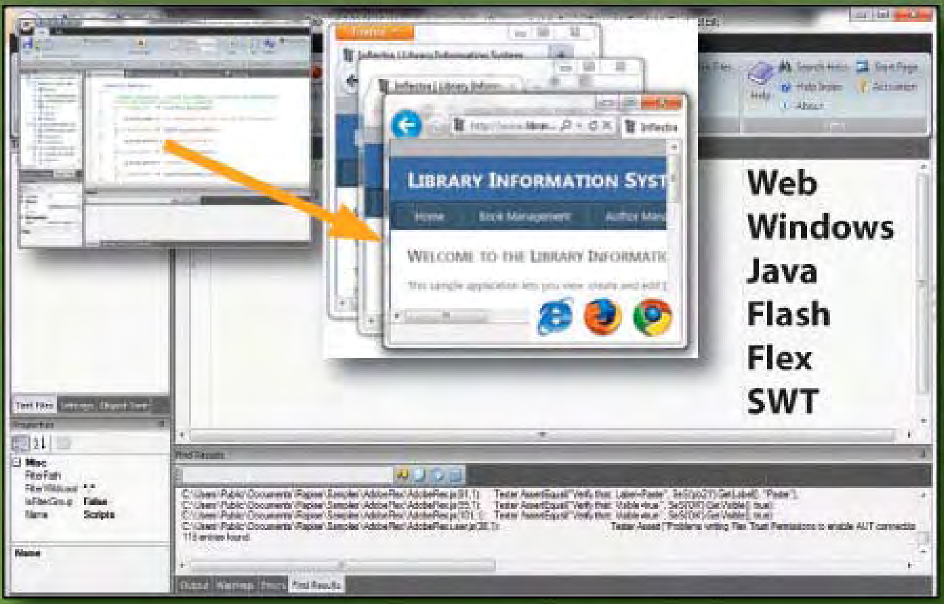
Context diagram

A context diagram can be a useful starting point for diagramming and documenting a software system, allowing you to step back and look at the big picture. Draw a simple block diagram showing your system as a box in the centre, surrounded by its users and the other systems that it interfaces with.

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

Need to Test Your Application on Multiple Environments?

Writing Test Scripts Too Slow?


Library Information System
Welcome to the Library Information
Web
Windows
Java
Flash
Flex
SWT

It's time to try a better way.

Rapise

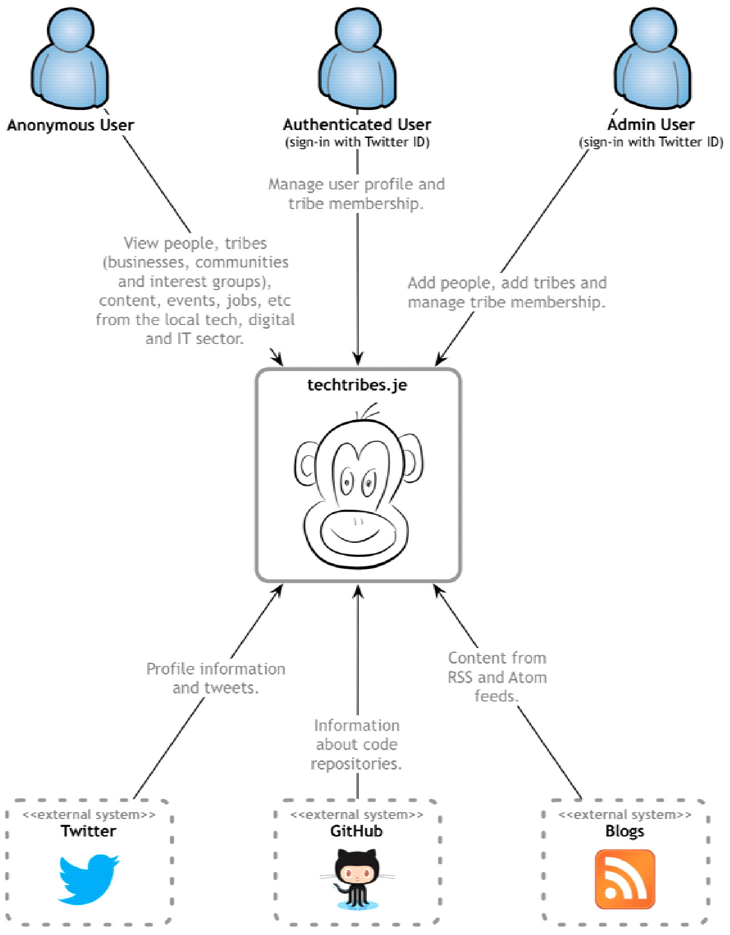
RAPID & FLEXIBLE TEST AUTOMATION

Learn more at: inflectra.com/rapise

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

inflectra

Let’s look at an example. The techtribes.je website provides a way to find people, tribes (businesses, communities, interest groups, etc) and content related to the tech, IT and digital sector in Jersey and Guernsey, the two largest of the Channel Islands. At the most basic level, it is a content aggregator for local tweets, news, blog posts, events, talks, jobs and more. Here's a context diagram that provides a visual summary of this.


Anonymous User
Authenticated User
(sign-in with Twitter ID)
Admin User
(sign-in with Twitter ID)
Manage user profile and tribe membership.
View people, tribes (businesses, communities and interest groups), content, events, jobs, etc from the local tech, digital and IT sector.
Add people, add tribes and manage tribe membership.
techtribes.je
Profile information and tweets.
Content from RSS and Atom feeds.
Information about code repositories.
<<external system>>
Twitter
<<external system>>
GitHub
<<external system>>
Blogs

techtribes.je - Context

Detail isn't important here as this is your zoomed out view showing a big picture of the system landscape. The focus should be on people (actors, roles, personas, etc) and software systems rather than technologies, protocols and other low-level details. It is the sort of diagram that you could show to non-technical people.

Containers diagram

Once you understand how your system fits in to the overall IT environment with a context diagram, a really useful next step can be to illustrate the high-level technology choices with a containers diagram. By "container" I mean something like a web server, application server, desktop application, mobile app, database, file system, etc. Essentially, what I call a container is anything that can host code or data. The following diagram shows the logical containers that make up the techtribes.je website.

Performance Testing with BlazeMeter - Click on ad to reach advertiser web site

BlazeMeter

THE LOAD TESTING CLOUD

FIND AND FIX
PERFORMANCE BOTTLENECKS
IN HOURS, NOT DAYS
WITH BLAZEMETER

Sign Up for a free account and load test your site right now!

Open Source compatible Apache JMeter

Self-service, start testing in 5 minutes

Agile performance testing for DevOps Teams

Mobile, API, Web Services and Web Apps

Scalable to 300,000+ concurrent users

Load test from the cloud and behind the firewall

BlazeMeter Integrates With

Apache JMeter™

New Relic.

TeamCity

Atlassian Bamboo

Jenkins

Drupal

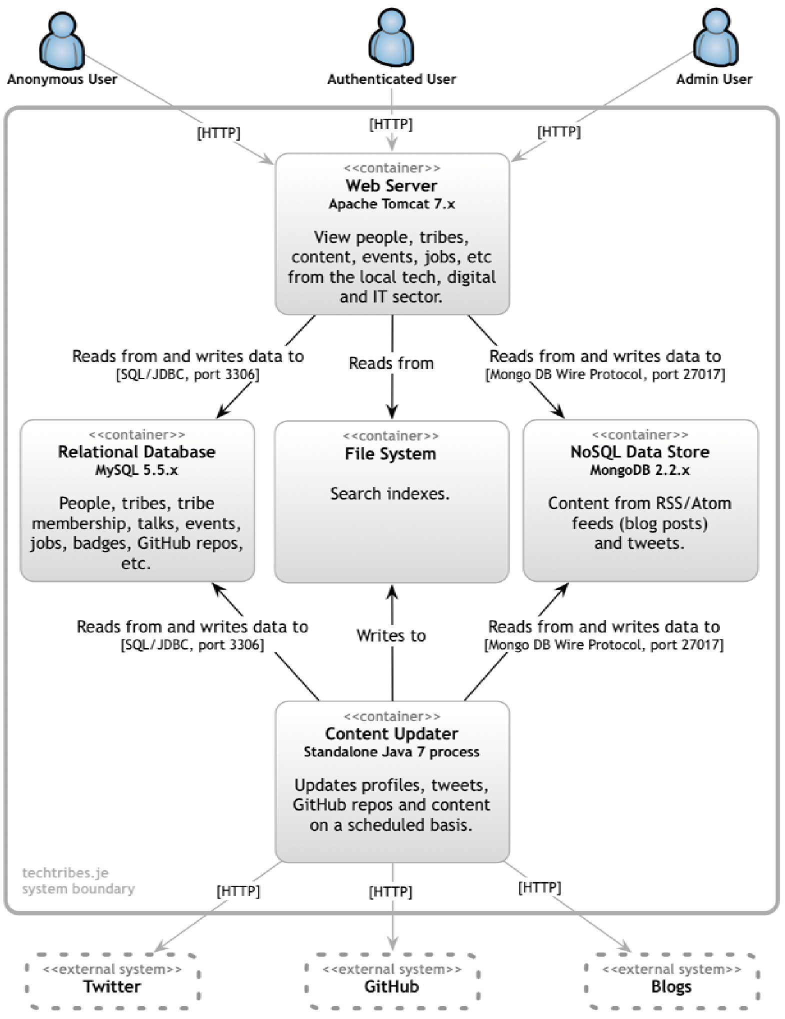
BlazeMeter Is Trusted By

BBC · heroku · MIT Massachusetts Institute of Technology · irdeto · ZAGG

NIKE · HubSpot · tableau · ticketfly · Adobe

Tel: 1.855.445.2285
info@blazemeter.com
@blazemeter

www.BlazeMeter.com


Anonymous User
Authenticated User
Admin User
[HTTP]
[HTTP]
[HTTP]
<<container>>
Web Server
Apache Tomcat 7.x
View people, tribes, content, events, jobs, etc from the local tech, digital and IT sector.
Reads from and writes data to [SQL/JDBC, port 3306]
Reads from
Reads from and writes data to [Mongo DB Wire Protocol, port 27017]
<<container>>
Relational Database
MySQL 5.5.x
People, tribes, tribe membership, talks, events, jobs, badges, GitHub repos, etc.
<<container>>
File System
Search indexes.
<<container>>
NoSQL Data Store
MongoDB 2.2.x
Content from RSS/Atom feeds (blog posts) and tweets.
Reads from and writes data to [SQL/JDBC, port 3306]
Writes to
Reads from and writes data to [Mongo DB Wire Protocol, port 27017]
<<container>>
Content Updater
Standalone Java 7 process
Updates profiles, tweets, GitHub repos and content on a scheduled basis.
techtribes.je system boundary
[HTTP]
[HTTP]
[HTTP]
<<external system>> Twitter
<<external system>> GitHub
<<external system>> Blogs

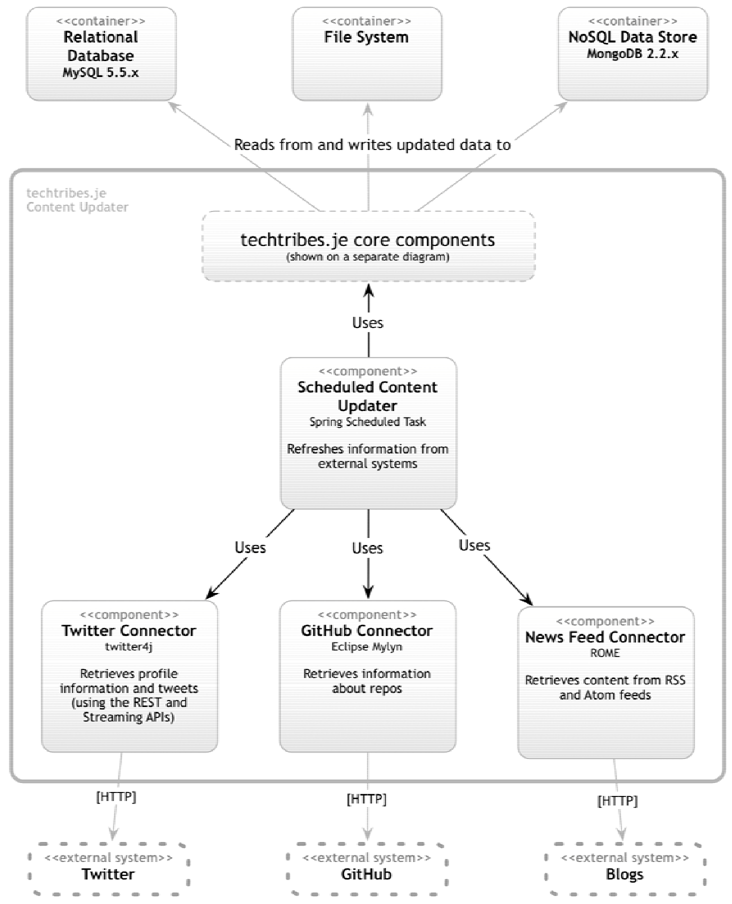
techtribes.je - Containers

Put simply, techtribes.je is made up of an Apache Tomcat web server that provides users with information, and that information is kept up to date by a standalone content updater process. All data is stored either in a MySQL database, a MongoDB database or the file system. It is worth pointing out that this diagram says nothing about the number of physical instances of each container. For example, there could be a farm of web servers running against a MongoDB cluster, but this diagram doesn't show that level of information. Instead, I show physical instances, failover, clustering, etc on a separate deployment diagram. The containers diagram shows the high-level shape of the software architecture and how responsibilities are distributed across it. It also shows the major technology choices and how the containers communicate with one another. It is a simple, high-level technology focussed diagram that is useful for software developers and support/operations staff alike.

Components diagram

Following on from a containers diagram showing the high-level technology decisions, I'll then start to zoom in and decompose each container further. However you decompose your system is up to you, but I tend to identify the major logical components and their interactions. This is about partitioning the functionality implemented by a software system into a number of distinct components, services, subsystems, layers, workflows, etc.

As illustrated by the containers diagram, techtribes.je includes a standalone process that pulls in content from Twitter, GitHub and blogs. The following diagram shows the high-level internal structure of the content updater in terms of components.



techtribes.je - Components - Content Updater
Standalone Java Process

In addition to a number of core components, the content updater is made up of four components: a Scheduled Content Updater, a Twitter Connector, a GitHub Connector and a News Feed Connector. This diagram shows how the content updater is divided into components, what each of those components are, their responsibilities and the technology/implementation details.

Class diagrams

This is an *optional* level of detail and I will typically draw a small number of high-level UML class diagrams if I want to explain how a particular pattern or component will be (or has been) implemented. The factors that prompt me to draw class diagrams for parts of the software system include the complexity of the software plus the size and experience of the team. Any UML diagrams that I do draw tend to be sketches rather than comprehensive models.

Think about the audience

There seems to be a common misconception that "architecture diagrams" must only present a high-level conceptual view of the world, so it is not surprising that software developers often regard them as pointless. In the same way that software architecture should be about coding, coaching and collaboration rather than ivory towers, software architecture diagrams should be grounded in reality too. Including technology choices (or options) is a usually a step in the right direction and will help prevent diagrams looking like an ivory tower architecture where a bunch of conceptual components magically collaborate to form an end-to-end software system.

A single diagram can quickly become cluttered and confused, but a collection of simple diagrams allows you to easily present the software from a number of different levels of abstraction. And this is an important point because it is not just software developers within the team that need information about the software. There are other stakeholders and consumers too; ranging from non-technical domain experts, testers and management through to technical staff in operations and support functions.

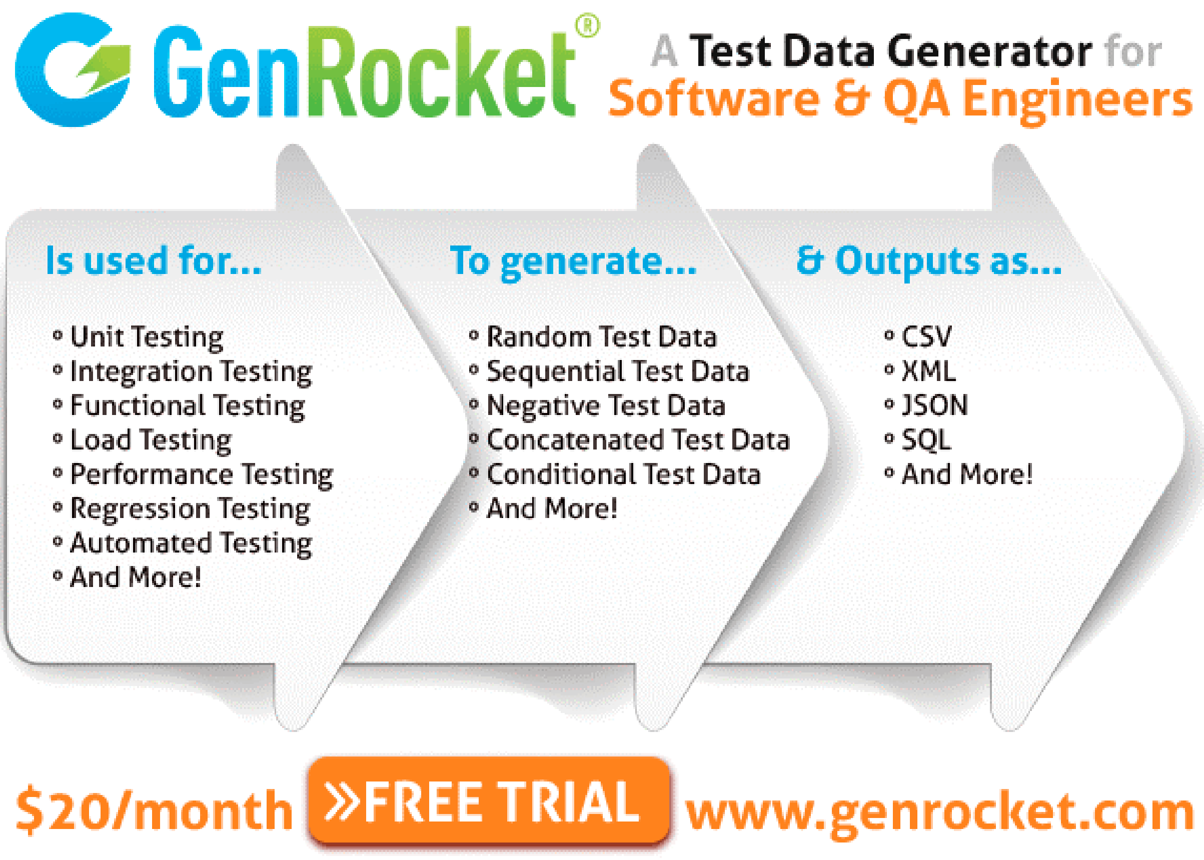
For example, a diagram showing the containers is particularly useful for people like operations and support staff that want *some* technical information about your software system, but don't necessarily need to know anything about the inner workings.

Common abstractions over a common notation

This simple sketching approach works for me and many of the software teams that I work with, but it is about providing some organisational ideas and guidelines rather than creating a prescriptive standard. The goal here is to help teams communicate their software designs in an effective and efficient way rather than creating another comprehensive modelling notation.

UML provides both a common set of abstractions *and* a common notation to describe them, but I rarely find teams that are using either effectively. I'd rather see teams able to discuss their software systems with a common set of abstractions in mind rather than struggling to understand what the various notational elements are trying to show. For me, a common set of abstractions is more important than a common notation.

Test Data Generation with GenRocket - Click on ad to reach advertiser web site



Most maps are a great example of this principle in action. They all tend to show roads, rivers, lakes, forests, towns, churches, etc but they often use different notation in terms of colour-coding, line styles, iconography, etc. The key to understanding them is exactly that - a key/legend tucked away in a corner somewhere. We can do the same with our software architecture diagrams.

It is worth reiterating that informal boxes and lines sketches provide flexibility at the expense of diagram consistency because you’re creating your own notation rather than using a standard like UML. My advice here is to be conscious of colour-coding, line style, shapes, etc and let a set of consistent notations evolve naturally within your team. Including a simple key/legend on each diagram to explain the notation will help. Oh, and if naming really *is* the hardest thing in software development, try to avoid a diagram that is simply a collection of labelled boxes. Annotating those boxes with responsibilities helps to avoid ambiguity while providing a nice “at a glance” view.

"Just enough" up front design

As a final point, Grady Booch has a great explanation of the difference between architecture and design. He says that architecture represents the "significant decisions", where significance is measured by cost of change. The context, containers and components diagrams show what I consider to be the significant structural elements of a software system. Therefore, in addition to helping teams with effective and efficient communication, adopting this approach to diagramming can also help software teams that struggle with either doing too much or too little up front design. Starting with a blank sheet of paper, many software systems can be designed and illustrated down to high-level components in a number of hours or days rather than weeks or months.

Illustrating the design of your software can be a quick and easy task that, when done well, can really help to introduce technical leadership and instil a sense of a shared technical vision that the whole team can buy into. Sketching should be a skill in every software developer's toolbox. It is a great way to visualise a solution and communicate it quickly plus it paves the way for collaborative design and collective code ownership.

The UX Runway - Integrating UX, Lean, and Scrum cohesively

Natalie Warnert, Thomson Reuters, info @ nataliewarnert.com
www.nataliewarnert.com, @nataliewarnert

Scrum has been the buzz for the past decade and User Experience (UX) wasn't far behind. There is much confusion about UX integration and my company is no exception. We no longer desire to have the entire system design completed before the coding starts as waterfall practices have prescribed; however, not all design can or should be thought about only during the development Sprint. The rapidly changing requirements and priorities these projects welcome can be detrimental if UX is not incorporated in a timely and correct way.

Additionally, UX work is hard to classify as "done" because there is always more that can be added, user tested, and improved. Lean UX helps to keep the focus on delivering the minimum viable product (MVP), the very base amount of detail needed, in a timely manner for development integration. This led me to create a UX Runway concept to help tackle some of these problems up front instead of waiting until the development Sprint. This concept originated from the Architectural Runway concept that the Scaled Agile Framework® (SAFe™) details.

This article looks to educate developers, project managers, ScrumMasters, Product Owners, product managers, UX team members, and the like about a way to integrate UX and Lean UX principles into Scrum projects.

It specifically focuses on the Scrum framework so familiarity with that method is encouraged when implementing the UX Runway practice detailed here and understanding this article. There are some concepts from SAFe but an in depth understanding is not critical. Though I have based the UX Runway around Scrum, it does have reusable concepts and could be readily adapted for other Agile methods.

What is User Experience (UX)?

It is important to understand what UX is, the different roles that can be on a UX team, and where these roles fit in relation to Scrum development. Simply put, UX is how a person interacts with a system - its information flow, intuitiveness, accessibility, ease of use, and their perceptions when using it. UX also often has responsibility for branding and consistency within an application, or suite of applications, and how it interacts and interfaces with other systems. Figure 1 shows a high level view of UX interaction.

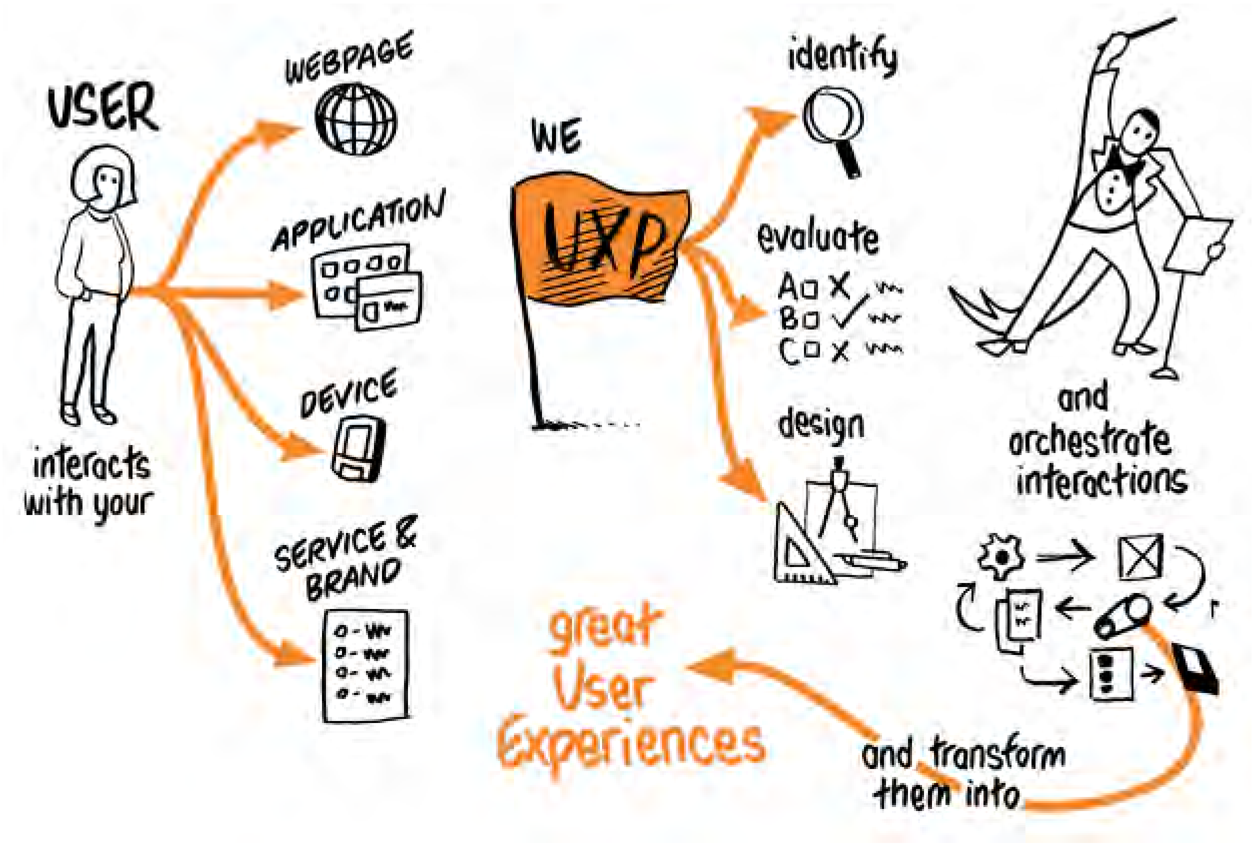


Figure 1 - source: www.uxpassion.com

The UX Centralized Scrum Team

The team I am the ScrumMaster for is very cross functional in their expertise and roles and is centralized within department. The team “consults” on many different projects throughout the organization. Each of the members is assigned to one or more projects and they work with many different development teams to direct the UX of the system. The team consists of user researchers, information architects (IAs), visual designers, and CSS developers.

STAREAST Software Testing Analysis & Review Conference - Click on ad to reach advertiser web site

This is different than what I call a decentralized UX team; one where UX experts are part of development teams as cross functional members and do not have a central department or a UX specific team with individual disciplines. Instead, the UX expert on the development team is meant to be an expert in all of the aforementioned roles. This is what Scrum would prescribe, but in large highly specialized organizations, this sometimes can't be the case. Here is a closer look at what the different roles within a UX team are responsible for.

User researchers conduct usability research in many forms before, during, and after applications are developed. Some of the more common studies are contextual inquiries, card sorts, user surveys, observatory research, interactive and paper prototyping and heuristic evaluations. These are all different ways of observing and interacting with users to understand how they use and react to the applications.

Many projects don't allocate much of their budget to research, but it has been proven to be very effective when it is correctly incorporated into the design of the application. By utilizing ongoing user research during the project, user behavior as a result of a misunderstood feature or interaction can be discovered earlier. This allows UX and development teams to respond rapidly to change and incorporate the feedback into a later Sprint to make a better end product.

Information architects take the user research, if applicable, and determine the logical user and data flow through the system. They detail how a user should interact with a system or follow a process to accomplish their end goal. They craft the wording contained in error or instructional messages and determine where additional information and/or directions are needed. They draft how problems should be solved, resulting in how features should be laid out on the page and interact with each other and the user and how the data should flow. This work is generally done in low fidelity wireframes and whiteboard sketching. These are the deliverables IAs produce to the development teams to aid in their coding of the application.

Visual designers work closely with the IAs and researchers to determine how the application should look in order to support the functionality. They determine the appropriate colors and styles to keep the user engaged and keep the system cohesive. They determine where buttons, widgets, and messages should be displayed to make the flow logical and make important features stand out. As my team members say, "make it pretty." Visual designers produce high fidelity mock-ups complete with the appropriate imagery, icons, and colors to be included in the application development.

CSS developers are a bit different than the other three roles above. CSS developers are often included as part of the cross functional development teams (though not in my organization). Contrary to the other roles, CSS developers do not need to work ahead of the development team for the most part; their work is parallel to the development work as they are creating actual code that can be deployed as part of a feature or story. They basically translate the visual designs into the code so widgets are aligned properly, correct images are used, and applications are coded to be accessible and compliant with government 508 guidelines (or the equivalent) surrounding screen readers and the like for users with disabilities. Some teams also have accessibility experts as a separate discipline, too.

Depending on how the development environments and associated systems are set up, there may be additional CSS work that can be done ahead of development. Certain HTML templates and packaged style sheets can be worked on in conjunction with the visual design direction. This allows the technology teams to be able to apply styles as they are developing and helps to prevent rework from the inline styles that may be incorrectly used. This is not the case in all projects or environments but there may be more planning that needs to be done up front in some

CSS cases than others. The specialization on this centralized UX team has some challenges arise but makes better products when the work is managed effectively. There can be confusion for the UX team when creating the Product and Sprint Backlogs because of the hand-offs and pre-work. It is also difficult for the development teams to write and break down stories and tasks because of the sheer number of dependencies and some of the uncertainty around a design or wireframe that is not yet complete.

This leads me to the UX Runway concept. After all, a runway is only a stretch of time or area when preparation for the next step is done (e.g. an airplane on a runway is preparing to take-off). As SAFe put it, “having the UX design track a bit ahead as part of the architectural runway for a system.” But what does this look like and how is it actually done?

UX Runway Project Kickoff and Sprint 0

When a business case is proposed, the Product Owner likely has a rough idea of some of the key functionality that will make the product desirable and aid its potential success. UX can provide some helpful context by creating business case designs. These designs can assist the Product Owner to think through their ideas, determine research opportunities, and put some visualization behind the words of the business case.

After business case approval and before development work can begin there is much demanded of UX, making a case for a period of time called the initial UX Runway. This initial UX Runway allows the UX team sufficient time to do their due diligence in mapping out the application and certain problems at a high level prior to development work starting. The Lean UX principle of problem focused designs makes more sense at this point instead of features because UX needs to be looking at the whole system and problems instead of working in a feature constrained box. This helps the system to remain cohesive and not constrain problem solving solutions at this early phase.

Another key UX deliverable during this time is the user personas. As a Product Owner is forced to think through who will actually use the product, what scenarios they will be using it for, and what their role in the system is it helps to extract more detail around the business case and high level view of system functionality. These revelations can lead to some of the epics that will initially drive release planning and the UX work.

User research can also use these user personas to identify potential roles and actual customers who can be used for user testing. Testing will vary depending on the phase of the project. At this point, contextual inquiries may be the best - simply watching how users do their job, either using an existing system, a competitors system, or even doing things manually. The most important part of this is the Lean UX principle of GOOB: Get out of the building. By observing users in their environment and learning how they do their job, much more valuable information can be gathered. Upon returning, this can be incorporated into solving the problems and the overall design of the whole system.

Development teams generally appreciate high level wireframes, designs, or prototyping work to put more context and transparency behind what they need to develop. I suggest and encourage UX team representatives to be involved in road mapping and epic and story creation sessions prior to the project kick off and Sprint 0. This allows UX to incorporate their opinions regarding where to include user research studies, critical linked system integration points, and identify other scenarios that will require extensive UX work.

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

Are You Tired of Having Separate Tools for Requirements, Testing, Bug-Tracking and Planning?

It's time to try a better way.

spiraTeam®

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

Learn more at: inflectra.com/spiraTeam

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

inflectra®

During road mapping sessions, it is helpful to identify each of the known epics in the first release. As the development teams and the Product Owner(s) are prioritizing, brainstorming, and writing stories to break down the epics, UX representatives should also be in the room. They can help to identify UX milestones and dependencies which should be documented as stories, too. These stories will help to create the initial UX Product Backlog. It is important to note, similar to traditional Scrum teams and planning, these are not the final or only stories for the release. As more details emerge in further planning and Sprint work, more stories will be added for UX and development teams.

When arranging the stories in a planning roadmap format, the development team stories should be planned prior to planning the UX stories. Then when the stories are in the agreed upon priority and development order, UX dependencies should be ordered in Sprints prior to the respective dependent development stories. It is done in this order because it gives both the UX team and development teams a better idea of how long features will take to develop. Also, it allows the UX team to see how the development teams are breaking down the epics, which guides research, wireframe, and design work delivery. Depending on feature size and problems to solve, the UX stories should be placed one to two Sprints prior to when the feature is set to be developed. This creates another facet of the UX Runway.

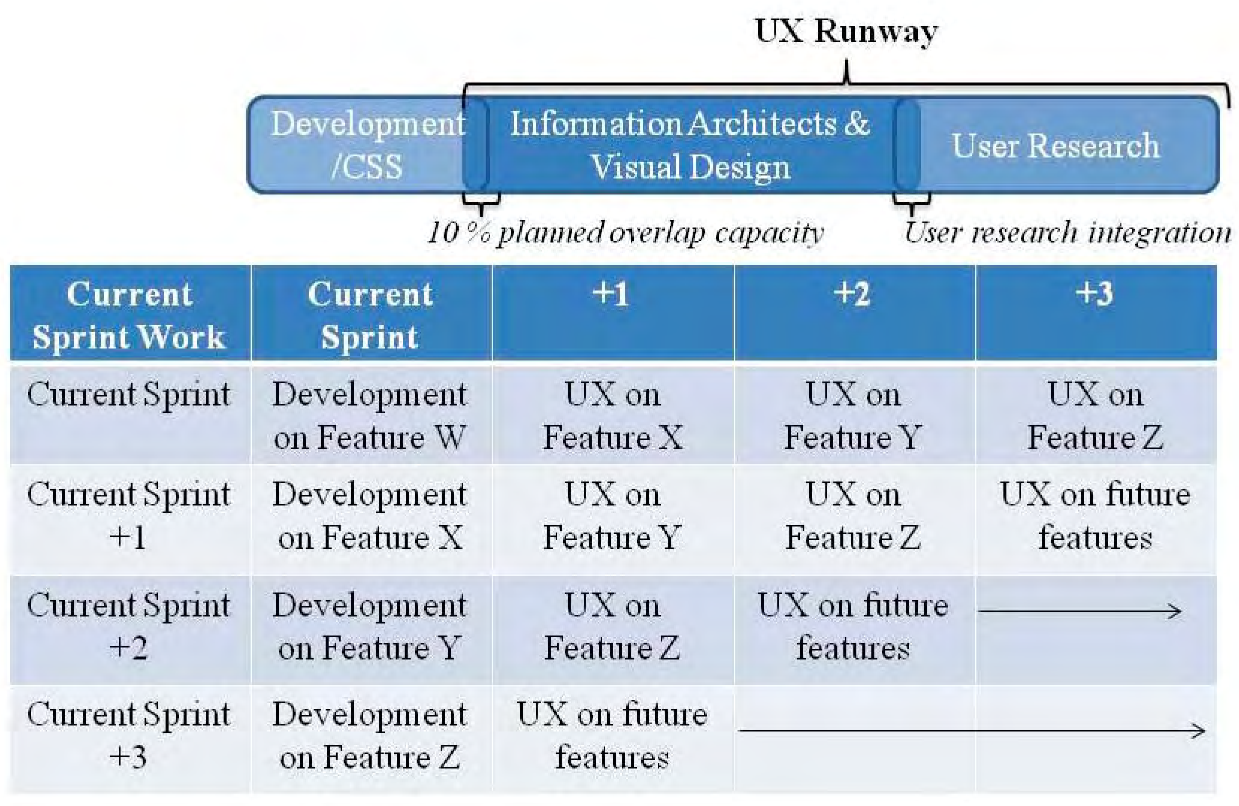
As mentioned earlier, after a rough roadmap is developed, the UX team should be allotted an initial UX Runway prior to and including Sprint 0, before development teams start any user interface (UI) work. The time length is not strictly prescriptive; it really depends on the project size (e.g. a rough estimate is about one week per large feature in a release per two UX team members). This allows the UX team adequate time to contribute to the project vision for the project sponsorship team while doing some additional user research and creating other desired deliverables. As the development teams start doing backend and other Sprint 0 work, they can see incremental progress on how the product will function and what it may look like. Architecture should also be involved with UX Runway initiative and the key members should work together to determine what usable and architecturally sound solutions the development teams should be setting up infrastructure for.

As the project kicks off and development teams start Sprinting, the ongoing UX Runway continues a Sprint or two ahead to make new wireframes and designs, add more detail to existing ones, redesign when necessary, conduct user research, answer questions, and react to change.

UX Runway during an Agile Project

When looking at the construct through a Scrum perspective, it is nearly impossible to research, wireframe, design, code, style, and test, and re-research a feature within a Sprint. Even if it were possible, many advocates would likely advise against it because it is nearly impossible to create stories small enough to allow all that work to be completed in a two to three week Sprint.

The centralized UX team works like most other Scrum teams with a few modifications. They follow the same cadence as the development teams they’re working with but not the same delivery schedule. When the development teams are coding the stories for the current Sprint, the IAs, designers, user researchers, and Product Owner are working from one to three Sprints ahead of them on future Product Backlog items creating a runway of sorts. There is some overlap that happens, which is planned and will be discussed in the following paragraphs. See Figure 2 below for a visual representation of the UX Runway and work breakdown between teams and roles.


UX Runway
Development /CSS
Information Architects & Visual Design
User Research
10 % planned overlap capacity
User research integration

Current Sprint Work	Current Sprint	+1	+2	+3
Current Sprint	Development on Feature W	UX on Feature X	UX on Feature Y	UX on Feature Z
Current Sprint +1	Development on Feature X	UX on Feature Y	UX on Feature Z	UX on future features
Current Sprint +2	Development on Feature Y	UX on Feature Z	UX on future features	→
Current Sprint +3	Development on Feature Z	UX on future features	→	

Figure 2 - a visual representation of the UX Runway

Though the concept seems simple it takes a lot of look ahead planning and coordination on the upcoming work in the Product Backlog. This requires staying in constant communication with all Product Owners and other development team's ScrumMasters. Cooperative Product Backlog grooming and Scrum of Scrum meetings are vital to the success of the UX Runway.

At grooming meetings, the Product Owner discusses features for the upcoming Releases and Sprints. As more details emerge around features, it is determined if there are going to be UX dependencies. When UX dependencies arise, the UX Product Owner for the centralized UX team and gathers notes and details to use in constructing the UX team's Product Backlog. These details also emerge during UX team working sessions with the project Product Owner and questions from development teams during grooming.

These working sessions are key to the UX Runway and Lean UX. This is where the Product Owners start to define the problem and the requirements. The UX team helps by acting as business analysts and problem solvers to better define what the Product Owner is looking for. Again, they are looking for the outcome, not the output. By working this closely together it creates a shared understanding between all parties. No one is the expert; instead the team is cohesively working together to avoid repetition and to be innovative. Additionally, these working sessions value action over analysis. This means they value generating ideas versus talking them to death. Using Lean principles, we know the first things will probably be wrong, but it is better to fail fast and continuously deliver until it is the right solution.

Ideally this leads to having a UX Product Backlog that is defined two to three Sprints ahead and a development team Product Backlog that clearly has dependencies linked to associated UX stories. The UX Product Backlog is groomed regularly to ensure continuity remains with possible requirement or priority changes.

UX Runway - Sprints

When Sprint planning, the stories being committed to by the UX team link to stories and features being developed by the development teams one to two Sprints ahead of the current Sprint being planned as Figure 2 shows. This UX Runway gives appropriate time for UX spikes, research, wire framing and designs to be developed and delivered to the development teams before the development Sprint for those features. With some of the larger features this runway may be adjusted to allow more time for any of the above activities, however just in time (JIT) Lean design and continuous delivery is the main goal.

JIT and Lean design lessens the confusion that can be related to working on Product Backlog items that simply aren't ready to be worked on, either because the Product Owner is not ready to start discussing them or the feature is not yet funded. This generally emerges in backlog grooming but it is wise to make sure stories are ready to be started by making a Definition of Ready for both the centralized UX team and the development teams. In fact, the Definition of Done for the UX team and the Definition of Ready for the development team should have overlapping pieces as should all dependent and dependee team's definitions.

Utilizing a Definition of Ready can help immensely with the UX Runway. If all cross functional team roles within the UX team are utilized in solving a problem, it creates a series of potential bottlenecks. A UX Theory of Constraints results: the UX Runway is only as short as its longest dependency: research, wireframes, design, or CSS. In order to incorporate the research into the wireframes and design it needs to be ready to do so, meaning it needs to be done. In looking back at Figure 2, there is not much of a runway between the cycles. User research often has a lot of front end planning, too. Staying ahead of these things and defining when research is ready to be continuously incorporated just in time is vital to keeping the project moving. Even when research is not needed, the principle of small batch sizes is once again used to continuously deliver the least amount of work for the development teams to start.

UX Sprint planning is done in a similar fashion to traditional Scrum teams. As mentioned before, the centralized UX team is on the same cadence as the development teams. Generally, it is wise to do UX planning after development team's planning is complete but within the same day. This ensures that if a feature gets pulled forward and it has not yet been on the UX Runway and is not yet "ready" from a UX perspective, the UX team can plan accordingly and reprioritize their Sprint work or advise against development in the Sprint.

In Sprint planning, UX team members volunteer for stories and write tasks. The team then commits to the work for the Sprint. One distinction is the stories (wireframes, designs etc.) being delivered from the IAs and designers will only be about 90 percent complete upon delivery to the development teams at the end of the Sprint. There are about 10 percent of team capacity built into every Sprint for a bit of re-work and question time from the development teams relating to those deliverables. This can happen for many reasons including:

- Technical issues or limitations that cause wireframes and user flow to change
- Identification of an issue or user scenario that was overlooked
- Low hanging fruit that hasn't been on the UX Runway yet and is pulled forward

A final thought to Sprint planning is the reality of the dependent team in a Scrum environment, UX aside. When a team is explicitly dependent on another to complete their work and is working in a fast changing environment it is hard to stick to a strict Sprint Backlog as Scrum prescribes. This is again why 10 percent capacity is left in the Sprint after UX feature delivery. Sometimes when development priorities are shifted late in the previous Sprint and the Product

Owners are in support of it, the Definition of Ready can get ignored and the development team will need to start working on something that is not UX “ready” and has missed the UX Runway.

Scrum tells us to say “no” to new work being added during the Sprint, but being a dependent team it is not that simple. To combat this additional work I have found it to be best to monitor how much unplanned work is submitted after the Sprint starts by development teams and make a velocity of it. When that unplanned work velocity is normalized as much as it can be, the additional capacity needed can be built into the UX Sprint Backlog by subtracting it from the current UX team’s velocity. This can mimic an expedite lane as seen on some Kanban boards. This velocity should be communicated to Product Owners and if unplanned work is added to the Sprint, they need to choose what is the most important.

Is this the ideal solution? No, but as changing requirements and priorities are embraced and teams are dependent on UX, we need to concede that not everything will make it on the UX Runway. The ultimate goal is that development teams are driven by UX work through the Product Owner explicitly, but is not the norm. Sprint planning needs to take this reality into account to ensure things are still getting delivered and the team collaboration is not upset by these very real changes.

After planning is done a Sprint Backlog of what the centralized UX team committed to is emailed to all development teams and parties involved. The Sprint Backlog is also available to view in the shared work management tool by stakeholders, development teams or anyone else with a stake in the work.

UX Runway - Scrum Meetings

The UX team does have Scrum meetings, though only about twice a week. This allows for members of the UX team to attend the daily Scrum of the teams they are working with a few times a week. Through this, all team members of both the UX team and the various development teams can stay up to date on the work and get valuable face time to answer questions and have some discussion if necessary. As with most Scrum teams, this also cuts down on some of the emails that can flood members’ inboxes with questions and requests more efficiently solved in person.

These Scrum meetings can be done as a whole UX team or as the smaller cross functional UX teams that are working on each project. Each has value: the centralized team can keep up to date on other projects and obstacles that may become issues to their project while keeping it to the small cross functional team can allow for more project specific discussion after the meeting. The teams should determine what they find to be useful and be willing to inspect and adapt.

We also use a Scrum board during our Scrum meetings. It helps to limit the work in progress and also visualize where the different work items are. There are many steps stories go through, as could be imagined with the many different team roles. To ensure the products stay consistent, wireframes and designs are reviewed while they are in progress by different members of the team at weekly internal reviews. This ensures quality and continuity between wireframes, designs, coding, and final CSS, all of which can take a few Sprints to be fully completed. They are also reviewed with the development teams and Product Owners in the end of Sprint demos.

UX Runway - Sprint Demos

The demos are a great time to showcase the wireframes and designs to the Product Owner and the development teams. Demos are a time to ask questions and analyze the proposed flows and user interaction recommendations the UX team is making. This helps the development teams to identify potential technical limitations with the recommended designs. This step is vital and gives the UX team time to rethink designs and offer other solutions when limitations arise.

The demos also allow the development teams and Product Owners to take a second look at the stories on the development team's Product Backlog and their acceptance criteria. Often wireframes and designs can help to uncover details that may have been missed in the earlier phases of discussion or where additional user research results have been added post story creation. The wireframes and designs can also help the development teams to flesh out story tasks during their Sprint planning. As previously mentioned all of these additional details and questions that result spur more conversations with the UX team.

These questions can lead to revisions to the wireframes and design, which is why the aim is to deliver 90 percent completion during the UX Runway Sprints and to providing just enough detail for the development teams to get started. This allows the UX team to remain lean by keeping the batch sizes small each Sprint. By not doing too much extra work, they eliminate the waste and re-work that can come from providing too much detail at the beginning. This lets the design iterate through UX continuous delivery and work out the smaller detail as they go. The UX team finishes the remaining 10 percent in the development Sprint based on what arises. This leads to a more well thought out and user-friendly final product.

Working Space

Another thing to consider is the concept of co-location when integrating a UX runway with a centralized team and multiple project teams. When each member of the UX team is working with multiple projects, teams, features, and Product Owners it is not prudent to have team members split their time between multiple team rooms. Though it may not be feasible for all teams, a project room for the UX team is great if there is availability.

It is a place for the UX team to collaborate on designs and wireframes to keep look and feel consistent between applications and problems that may be being solved by different team members. It also is a common place where Product Owners can stop in and clarify requirements and development team members can ask questions and have impromptu meetings. As in most team rooms there should be plenty of whiteboard space, places to meet, and quieter spaces for team members to have creative time.

A few great ways I have seen this utilized is by hanging up a product vision or storyboarding on the white boards, or in Lean terms, externalizing your work. Team members look at the vision individually and put sticky notes on things that are not clear. As for storyboarding, quite often the windows will be covered with sticky notes and the white boards with drawings and pictures. This helps to be leaner by getting out of the deliverables business. The artifacts we come up with don't solve the problems; I have seen wireframes done on a paper plate. It is the collective understanding and frequent collaboration to get everyone on the same page. The UX Runway simply allows for adequate time to have those conversations and think through the problems in whatever way is best at that time.

ScrumMaster and Product Owners for UX team

The UX ScrumMaster, in addition to the planning, grooming, and daily Scrum facilitation, attends Scrum of Scrum meetings for each project. This allows yet another layer of transparency with what the UX team and the development teams are working on. It is a place where issues are discussed and demos and working session meetings are coordinated. The role of a ScrumMaster is very important. They facilitate the Scrum meetings such as retrospectives, planning, daily scrums etc. and work to remove impediments and make things easier for the team to do their work. A senior UX expert, or less ideally the ScrumMaster, can act as the main Product Owner for the team, too.

Within the UX team the main Product Owner provides priority for the different project feature problem solving work each member is doing (remember each member is likely working on multiple projects and priority setting is key among projects), help them to set up sessions, ask the right questions to the right stakeholders, and have a forum to share their deliverables with the development teams and business Product Owners.

This should provide some context on how UX, Scrum and Lean can successfully work together in a large organization with a centralized and specialized UX team that is in very high demand. When UX teams are given appropriate UX Runway time both before the project starts and during the Sprinting phases, they can think through problems and make several solutions. This UX Runway allows for more iterations of problem solving, wireframing, design, and research to deliver high quality, user based solutions.

By being involved at all points of the development process and sitting at the table with development teams throughout, dependencies can be identified earlier and reduce the “fire drills” that so often are part of large projects, even in a Scrum framework. It is also important for the UX team to remain flexible and lean. When integrated correctly, all features should go through with the UX team. This increases consistency and provides a great tool for development teams to use in story writing and coding. If the process is managed and adapted to fit the project and the teams, the overall product will more usable and intuitive and added overhead will be minimal. Though it is not following all Scrum guidelines, the UX Runway has been proven to work in allowing inspection and adaptation for efficiency and quality among multiple applications.

References

Lean UX Applying Lean Principles to User Experience, Jeff Gothelf with Josh Sieden, O'Reilly Media, ISBN 978-1449311650

508 Compliance - http://www.hhs.gov/web/508/accessiblefiles/index.html

UX Runway - How to Incorporate UX with Agile/Scrum teams - http://nataliewarnert.com/ux-runway/

Definition of Ready (DoR) in Dependent Relationships - http://nataliewarnert.com/definition-of-ready-in-dependent-relationships/

Scaled Agile Framework (UX) - http://scaledagileframework.com/ux/
Scaled Agile Framework® and SAFe™ are trademarks of Leffingwell LLC

UX image - www.uxpassion.com

Toyota Kata - Habits for Continuous Improvements

Håkan Forss, Lean/Agile Coach at Avega Group, hakan.forss @ avegagroup.se
http://hakanforss.wordpress.com, @hakanforss

Do you feel your team has plateaued? Is the team not improving as much? Is the team lacking the motivation to challenge themselves to improve? Are the improvements you implement moving you in many different directions with no focus? Are you collecting a long list of problems but you never get around resolving them?

Time to stop collecting problems and start improving! Toyota Kata is a structured and focused approach to create a continuous learning and improvement culture. A kaizen culture.

Toyota Kata in a nutshell

Toyota Kata is a structured way to create a culture of continuous learning and improvement at all levels. It is an organizations daily habits or routines forming its “muscle memory” for continuous learning and improvements. The daily habits/routines help us to strive towards our vision, or our state of awesomeness in small focused experiments.

- First, we need to Understand our desired Direction
- Secondly, we need to Grasp our Current Condition
- Thirdly, Set the next Challenge on the desired path
- Fourthly, we Run small Experiments through the unknown towards our next challenge
- Then we repeat!

This is Toyota Kata in a nutshell.

In the Toyota Kata book, Mike Rother describes the continuous improvement habits, routines, behavior patterns or Katas if you will, observed at Toyota during his research. The two main Kata described in the Toyota Kata book are Mike Rother´s codification of these habits, routines and behavior patterns.

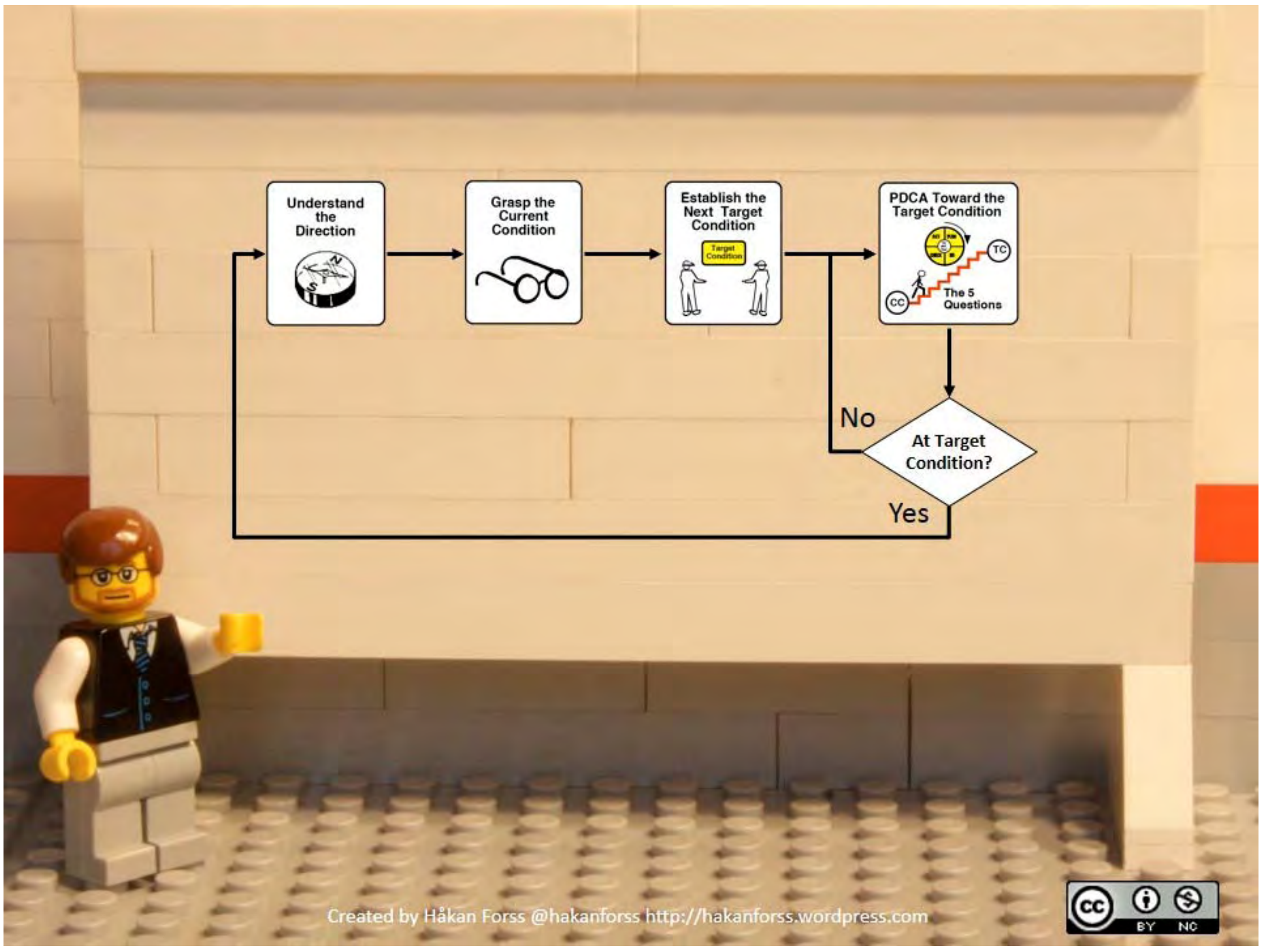
Let us look at the two main Katas described in Toyota Kata, the Improvement Kata and the Coaching Kata.

Improvement Kata

At the heart of Toyota Kata is the Improvement Kata. The Improvement Kata forms the continuous improvement habits of the method. The Improvement Kata guides the learners, the teams, through a four-step process focused on learning and improving your way of working.

The purpose of the Improvement Kata is to learn more about the organization processes. Building the organizations understanding of how work works. With this understanding and ability to learn the organization can improve their way of working striving towards a state of awesomeness in small focused experiments.

The Improvement Kata has four stages: *Understand the Direction, Grasp the Current Condition, Establish the Next Target Condition* and *PDCA toward the Target Condition*.


Understand the Direction
Grasp the Current Condition
Establish the Next Target Condition
Target Condition
PDCA Toward the Target Condition
TC
CC
The 5 Questions
No
At Target Condition?
Yes
Created by Håkan Forss @hakanforss http://hakanforss.wordpress.com
BY NC

Understand the Direction

“A goal is not always meant to be reached, it often serves simply as something to aim at” ~ Bruce Lee

Having a shared understanding of the direction of how you want to improve is important. Without a theory of what you think is better, it is very easy to get lost and just randomly make changes that will not move you in a clear direction. Having a clear and shared understanding of the direction also creates a sense of purpose for the people doing the improvement work. Without a clear purpose, people have a much harder time to get motivated.

To create a shared understanding of the direction you should set up a Vision of how work is done in an ideal state. This vision should be process focused, not outcome focused. The vision is a vision of how work is done, not a business vision. The vision should describe how your processes work when at the ideal state.

This is Toyota’s long-term vision for its Production Operations

- Zero defects
- 100 percent value added
- One-piece flow, in sequence, on demand
- Security for people (No injuries)

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

tinypm

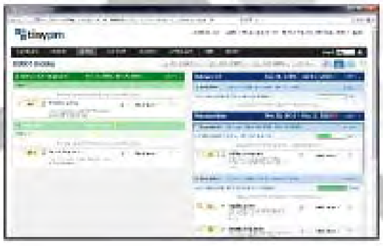
NEW HOSTED VERSION

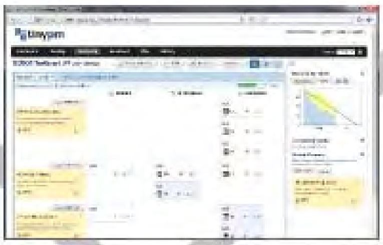
FREE 5-USER Community Edition

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

Tiny Effort, Perfect Management

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





Team collaboration

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

Customer engagement

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

Agile management

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

Integrations

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

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

Now with **Customizable Taskboard!**
www.tinypm.com

Features

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

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

Release Planning
Grouping iterations into releases
Release delivery forecasts

Iterations
Iteration planning and tracking
Iteration goals

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

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

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

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

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

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

Contact us
support@tinypm.com

Follow us
twitter.com/tinypm

What could a potential vision look like for a software development process? There is of cause no right and wrong and you really need to develop your own for your context. Here is one potential vision:

- Zero defects
- Every check-in to production
- Highest value first, on demand
- Motivated people

Every organization has to define its vision. You have to look at your context. What is your ideal state? What is your state of awesomeness?

Grasp the Current Condition

When you have a shared understanding of your direction, it is time to understand where you are now. You need to create a true understanding of the current condition of how you are operating. How do you really do work? Create a description of your actual processes. You should describe how you really work, not how you are supposed to work. A simple block diagram of the different steps is usually good. You should also collect process metrics that describe how your process is operating. You should also start to collect two types of metrics: process metrics and outcome metrics. Process metrics describe how your process is operating. Process metrics are typically leading indicators that indicates the outcome of the process before the fact. Output metrics on the other hand describes the outcome of the process, the result. These metrics are after the fact and is the result of how we operated the process.

Here are some examples:

Process metric	Cycle time	The period required to complete an item, or task from start to finish in one part of the process
	Number of people	Number people currently working in the process
	Work-In-Process	The amount of items, or tasks currently in one process step
	Queue size	The amount of items, or tasks currently in a queue in the process
	Iteration length, Takt time	The duration of a process cycle, at what pace/cadence the process should run
	Defects	The number of defects
Outcome metric	Lead-time	The time to complete an item, or task from start of the process all the way to the end of the process
	Throughput	The number of items, or tasks completed in a period of time
	Quality	The quality of the product you produce

Establish the Next Target Condition

Now that you have a clear Understanding of your Direction and of your Current Condition, it is time to describe your desired state in the near future. It is time to Establish the Next Target Condition. A Target Condition should describe how your process should operate when you are at the desired state. The focus should be on the process, not the outcome. The outcome should be the result of how you run the process. The Target Condition should be a hypothesis that takes you one-step closer to your Vision. The hypothesis should align with your theory of what is an improvement, and with your operations strategy.

You set a Target Condition by copying the Current Condition. This includes your process description, the process metrics and the output metrics. Then, based on your hypothesis you make a change that will move you a step closer to your Vision. The Target Condition should be in absolute numbers, not relative. It should be absolute clear if you have reached the Target Condition or not. When you set a Target Condition, you define an expiration date. The expiration date is typically set one to three months out. The date should be set to create a sense of urgency that will motivate you to get going right away. Target Conditions expires. Either the expiration date is passed, or you reach the Target Condition.

The Improvement Kata and the Coaching Kata has a very high focus on learning. As we learn more about how our processes work, we understand more what would be an improvement. Try to set the Target Condition just beyond your current knowledge threshold. It should force you to think outside the box. It should push you to try what you have not tried before. It should feel like putting a square peg in a round whole. Be aware of not setting a Target Condition that is too challenging, as this tends to demotivate people. Try to follow the Goldilocks rule: Not too hard, Not too easy, but Just Right.

Having a clear Target Condition is very important for effective process improvements. Toyota will usually not start their improvement work until a Target Condition is clearly defined.

PDCA toward the Target Condition

Time to start improving! We now have a shared understanding of our Current Condition, we have a defined Target Condition and when to archive it. We run small experiments, or Plan-Do-Check-Act cycles, to try to remove one obstacle at a time. We only address the obstacles that stand in the way to get to the Target Condition.

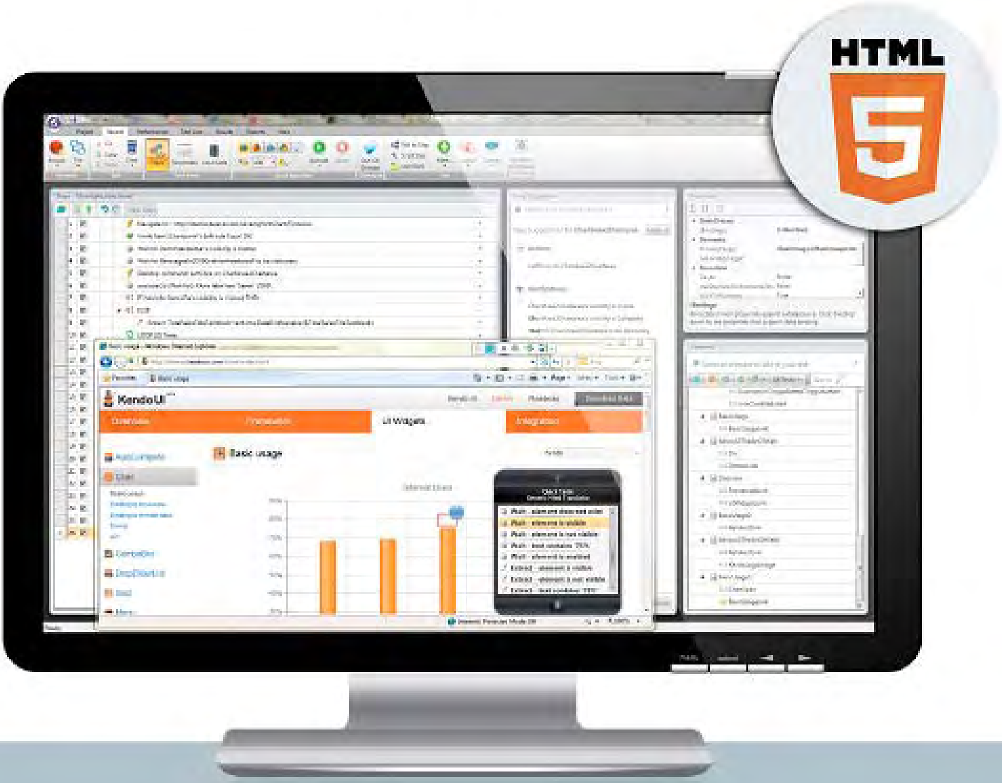
This process of running experiments is really about learning, learning how our processes work in our context. As we learn, we can adjust our theoretical model of how our processes work. As we adjust this model it will feed into our next experiment as we address the next obstacle. Our experiments, or PDCA cycles, follows the scientific model.

First, we formulate a hypothesis based on our theoretical model. Secondly, we make an explicit and detailed predication of what we think will be the result of running the experiment. We also define the exact date and time the experiment will end. The third step is to run the experiment. As we run the experiment, we should closely observe and collect appropriate metrics to tell us how the experiment went. Forth, now it is time to compare the difference between the predication and the actual outcome of the experiment. This is the time to reflect and learn. The delta between the predication and the outcome is really our opportunity to learn. If we always get the expected result, we have not really learned anything. Based on what we have learned it is now time to adjust our theoretical model, take on the next obstacle with a new hypothesis.

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

Test Studio

Easily record automated tests for your modern HTML5 apps



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

Download Trial

www.telerik.com/test-studio

We repeat this cycle until we have reached the Target Condition or the time has expired for the current Target Condition. When the Target Condition is reached or it has expired we loop back to step one of the Improvement Kata, Understand the Direction.

Doesn’t sound like rocket sciences? It is not hard too intellectually understand how to do it. It may even sound as if it is too easy too really work. Intellectually understand and actually do it is two totally different ball games. You need to practice and practice too actually get it right. Toyota Kata is by design set up to give you support as you practice. The Coaching Kata provides this support.

Coaching Kata

The second and equally important part of Toyota Kata is the Coaching Kata. The Coaching Kata is supporting the Improvement Kata by helping the learners to focus on learning, improving, and pointing in the right direction. The Coaching Kata itself is primarily supporting the fourth step of the Improvement Kata, PDCA toward the Target Condition.

Purpose

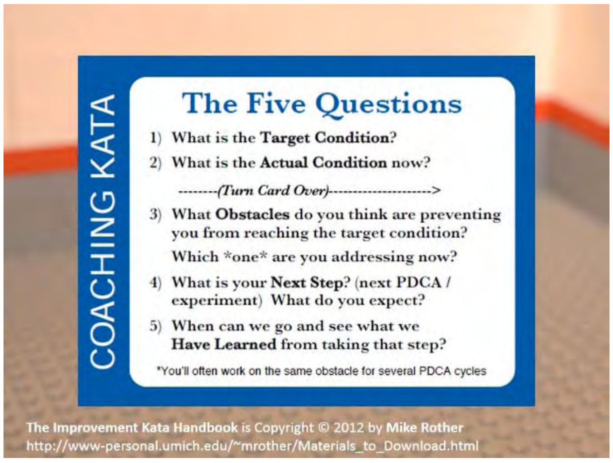
With the Coaching Kata the leaders of the organization should take a coaching and supporting role with the learners, the teams. The coach would be the one that is challenging the learners to take a small step beyond their current knowledge threshold, to challenge themselves.

When practicing the Improvement Kata the focus should be on one obstacle at a time. Only focus on obstacles standing in the way towards the next Target Condition. It is the role of the coach to help the learners to keep this focus.

In a full implantation, this means that there is a coaching-learner relationship throughout the whole organization on all levels. Every leader would be acting as coach for their team.

The 5 questions

The Coaching Kata 5 questions are used during the fourth step of the Improvement Kata, PDCA toward the Target Condition. For every PDCA cycle, experiment, we run to remove the obstacles that are stopping us from operating as described in the Target Condition we go through the following main questions as describes on Coaching Kata card.



COACHING KATA

The Five Questions

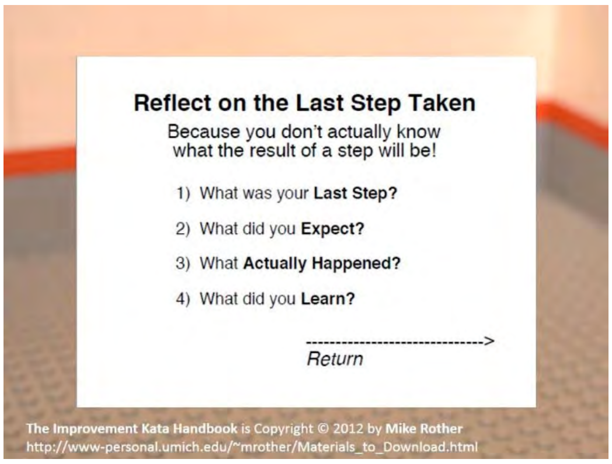
1) What is the **Target Condition**?
2) What is the **Actual Condition** now?

--------*(Turn Card Over)*-------------------->

3) What **Obstacles** do you think are preventing you from reaching the target condition?
Which *one* are you addressing now?
4) What is your **Next Step**? (next PDCA / experiment) What do you expect?
5) When can we go and see what we **Have Learned** from taking that step?

*You'll often work on the same obstacle for several PDCA cycles

The Improvement Kata Handbook is Copyright © 2012 by Mike Rother
http://www-personal.umich.edu/~mrother/Materials_to_Download.html



Reflect on the Last Step Taken

Because you don't actually know what the result of a step will be!

1) What was your **Last Step?**
2) What did you **Expect?**
3) What **Actually Happened?**
4) What did you **Learn?**

------------------------------>
Return

The Improvement Kata Handbook is Copyright © 2012 by Mike Rother
http://www-personal.umich.edu/~mrother/Materials_to_Download.html

Going through the question card

1. What is the Target Condition?

The purpose of the question is to reinforce the focus of where we want to move. It is a way for the coach to understand if there is an alignment of definition of the Target Condition. It is also an opportunity to understand if there is an alignment of the purpose of the Target Condition.

2. What is the Actual Condition now?

When asking this question the learners, the team, should be able to describe the how the process is currently operating. This is the same as the second step in the Improvement Kata, Grasp the Current Condition with an added focus on the current PDCA cycle. The learned should be able to show the current description of the process, the process metrics and the output metrics.

Now it is time to turn the card over. You can skip this if this is the first time you ran the Coaching Kata after you set a new Target Condition. Then skip down to question 3 below

----------------- *Turn Card Over* ------------------>

2.1 What was your Last Step?

With this question, we repeat what the last PDCA cycle, experiment, was. We have the opportunity to compare what we set out to do, and what we really did. If there is a difference, the coach has the opportunity to explore why we could not do what we set out to do.

2.2 What did you Expect?

We look back at what we expected would be the outcome of the last PDCA cycle, experiment. It is very easy to define what we expected after the fact. This step help us remember what we actually defined as the expectations so we can create a greater understand of what really happened.

2.3 What Actually Happened?

Now we take a real look at what really happened when we ran the last PDCA cycle, experiment. We look at the current condition. We look at what changed when we ran the last PDCA cycle, experiment. Doing this explicitly helps us learn more about how our process works.

2.4 What did you Learn?

This is probably the most significant question in the Coaching Kata, and Toyota Kata all together. Toyota Kata is about growing people, teaching people to learn how to learn. As we learn, we can use that learning to improve our processes. Do not haphazardly go through this step!

Reflecting on what we have learned and how that will feed into the next PDCA cycle is the focus of this question. We should compare what we expected would happen and what really happened. The delta between the two would be a big part of our potential learning. The bigger the delta, the bigger potential for learning. With no, or a very small delta, we are probably just late implementing the change.

With a big delta, we need take a good look at our understanding and assumptions of how things work. We need to update our theory and metal model of how things work. This can have a profound impact on our next experiments and Target Condition as we move towards our vision.

Return --------->

Time to return to the front side of the 5 questions card

3. What Obstacles do you think are preventing you from reaching the target condition? Which One are you addressing now?

We know the Actual Condition. We know our Target Condition. What obstacles do you think is standing in the way to get to the Target Condition? We are only interested in the current obstacles and only the ones preventing us to get to the Target Condition. There are many problems and obstacles that we want to remove, but we should only focus on the ones preventing us to get to the Target Condition. Why? As we are removing obstacles, some new will appear and some old will disappear. If try to fix everything we can spend a lot of time fixing things that are not really needed. We can very easily start to lose focus and direction.

After defining the current obstacles, we should select one obstacle that will get our focused attention. We focus on one obstacle at a time to make the learning of cause and effect easier.

Start addressing the obstacles that you have control over first. It is recommended to take on the easy and small obstacles first. By removing the easy and small ones that you have control over, you get some quick wins. Quick wins increase motivation and creates room to spend more time on the bigger obstacles. As we move forward, some of the big obstacles are no longer preventing you from reaching the Target Condition.

4. What is your Next Step (next PDCA/experiment)? What do you Expect?

You have identified the next obstacle to work on. Now it is time to formulate a hypothesis of that is causing the obstacle. When you have a hypothesis, you should design an experiment to try to remove the obstacle.

When you have an experiment defined, you should also as explicitly as possible describe what you expect will happen when the experiment is run. Spending some time at this step will really help you when you will evaluate what you learned from the experiment. It is very easy to skip over this part. If you do you will most often lose some valuable learning.

5. When can we go and see what we have learned from taking that step?

This question has two main parts:

When. By defining a specific time when it is time to go through the next cycle we create a clear understanding when we are supposed to be done with the next cycle. To get the most out of the Improvement Kata you should favor smaller and shorter PDCA cycles, experiments. If you can ran at least one cycle a day or even more than one cycle per day you will get the most out of Improvement Kata.

Go and see what we have learned. Go and see for yourself where the work is really done is an important part of Lean Thinking. When we run the Improvement Kata we should do it where the work is done. Not in a conference room separated from the actual work. We should also focus on the learning, not the actual results. If you focus on the learning, the results will follow. If you focus on the results, it is easy to start cut corners to get the short-term results. This in not what the Toyota Kata and Lean Thinking is about. It is about developing people and looking at the long-term sustainable results.

Summary

Toyota Kata is a structured and focused approach to create a continuous learning and improvement culture. The improvement Kata and Coaching creates organizational "muscle memory" for continuous improvements. The two Kata will give your organization familiar routines, as you probe through the unknown, striving for your state of awesomeness.

What to learn more

Here are some useful links to more information about Toyota Kata.

http://www-personal.umich.edu/~mrother/Homepage.html

http://www.lean.org/kata/

http://www.slideshare.net/mike734

http://www.slideshare.net/BillCW3/

http://hakanforss.wordpress.com/tag/toyota-kata/

What is a Kata?

Kata means pattern, routine, habits or way of doing things. Kata is about creating a fast "muscle memory" of how to take action instantaneously in a situation without having to go through a slower logical procedure. A Kata is something that you practice over and over striving for perfection. In the book "Managing Flow", Ikujiro Nonaka describes Kata as a traditional Japanese code of knowledge that describes a process of synthesizing thought and behavior in skillful action; the metacognition of reflection in action. If the Kata itself is relative static, the content of the Kata, as we execute it is modified based on the situation and context in real-time as it happens. Nonaka also describes Kata as different from a routine in that it contains a continuous self-renewal process.

Kata is not to blindly copy some else method, but to improve on it in an evolutionary way. You learn and evolve a Kata through the three stages of the learning cycle Shu (learn), Ha (break) and Ri (create). In the first stage Shu, you learn by following the teacher. You imitate the teacher's practices, values and thinking. You will only move on to the next stage when you have made the teacher's Kata your own. In the Ha stage, you break from the teacher's practices and make modifications based on your own creativity. In the Ri stage, you leave the teacher and you start creating your own unique Kata. As you expand your knowledge into new areas, you will loop back to the Shu stage for those areas in an ever-growing spiral of knowledge.

GVM, the Groovy enVironment Manager

Marco Vermeulen, vermeulen.mp @ gmail.com
http://wiredforcode.com, @gvmtool, @marcoVermeulen

Groovy enVironment Manager (GVM) is an open source tool for managing parallel Versions of multiple Software Development Kits on most Unix based systems. It provides a convenient command line interface for installing, switching, removing and listing Candidates. GVM is currently focused on the Groovy ecosystem, but will soon support other communities too.

Web Site: http://gvmtool.net/
Version: 1.3.12
System Requirements: Mac OSX, Linux, Cygwin, Solaris, FreeBSD
License and Pricing: Apache 2.0, Free OSS
Support: https://github.com/gvmtool/gvm/issues

Developers spend most of their time writing code. In order to do this they make extensive use of tools. One of the most basic tools used by developers is a Software Development Kit, or SDK.

Wikipedia defines an SDK as follows: "A software development kit (SDK or "devkit") is typically a set of software development tools that allows for the creation of applications for a certain software package, software framework, hardware platform, computer system, video game console, operating system, or similar development platform."

When dealing with Java technology, developers constantly need to search for, download and install these SDKs in order to perform their development. This in itself is not a very complex task, but is certainly mundane and time consuming. Wouldn't it be nice if we could delegate this laborious task elsewhere? Such a solution exists, and is called GVM, the Groovy enVironment Manager.

As a Groovy and Java developer, I had experienced this pain once too often. Not only did I need to install SDKs, but also needed to manage multiple versions side-by-side. I realized that I was not the only person having to do this often, so I embarked in writing a SDK manager to solve this problem once for all.

GVM went live in December 2012, and is still under active development. With over 23,500 installations worldwide at the time of writing, it is currently the preferred way of installing Groovy related technologies. However, it will soon become available for other Java related stacks such as Scala, Clojure and even Java too!

Key Concepts

GVM has a concept of Candidates and Versions. Candidates are SDK types, and each type can have multiple Versions. Examples of Candidates are Grails, Gradle, or even Groovy itself. Every Candidate has a default Version that is always set to the latest stable release. GVM will notify you when a new Version of a Candidate becomes available, and will allow for easy installation by issuing a single command. Even if you're not a programmer, the use of GVM is so simple that you can follow along easily. So please open a terminal and try this out!

Installation

Installing GVM is a very simple process. First, some prerequisites:

- a machine running some variant of Unix (Mac and Linux or even Windows with Cygwin will do!).
- curl (for downloading)
- zip (to handle the downloaded archives)
- sed (for string processing)
- a pre-installed Java SDK

When running the GVM installer, it will check for the presence of these on your system and ask you to install them if they are not present.

To run the installer, enter the following in a terminal:

```
$ curl -s get.gvm.io | bash
```

If all is well, you should see a message confirming that GVM is installed. Now close the current terminal and open a new one to start using GVM!

Usage

GVM has a bash client, allowing you to enter commands from the command line. This allows management of Candidate Versions with ease. For instance, to install the default version of Groovy, enter the following:

```
$ gvm install groovy
```

The tool will now download and install the latest version of Groovy and place it on your path ready for use. After watching the download complete and answering `Y` to the question of making it your default version, enter the following to verify that you have installed groovy successfully:

```
$ groovy -version
```

If all went well, you should see something like:

```
Groovy Version: 2.2.0 JVM: 1.7.0_45 Vendor: Oracle Corporation
OS: Linux
```

To get a specific Version in parallel to the one already downloaded:

```
$ gvm install groovy 2.1.9
```

To switch to another version:

```
$ gvm use groovy 2.2.0
```

Removing a version is as simple as:

```
$ gvm uninstall groovy 2.1.9
```

To see a list of all available Versions for a Candidate, enter:

```
$ gvm list groovy
```

Which renders something like:

```
=========================================
Available Groovy Versions
=========================================
 > * 2.2.0                2.0.4
   * 2.1.9                2.0.3
     2.1.8                2.0.2
   * 2.1.7                2.0.1
   * 2.1.6              + 2.0.0-beta-1
   * 2.1.5                2.0.0
=========================================
+ - local version
* - installed
> - currently in use
=========================================
```

Documentation

Of course, this is only the start of what GVM can do. For basic documentation, please visit our web site. For a comprehensive list of usage scenarios, please have a look at our body of Cucumber specifications that reside on GitHub:
https://github.com/gvmtool/gvm/tree/master/src/test/cucumber/gvm

Summary

GVM is the Groovy enVironment Manager. It is a tool used by developers to manage Groovy Software Development Kits (SDKs) on their workstations. It automates and eases the mundane task of downloading and installing SDKs, freeing up developers to focus on more important tasks at hand. In addition it allows for easy switching between different versions of an SDK during development. GVM is currently available for Groovy related technologies, but will soon be available for other Java stacks too.

Selenide: Concise UI tests in Java

Andrei Solntsev, @asolntsev,
Codeborne, http://codeborne.com/

Selenide is a tool for writing concise, expressive and stable UI tests in Java. Selenide resolves all the typical problems with testing of modern web applications like Ajax and timeouts in a simple and elegant way. Selenide is extremely simple to start with: you don't need to read hundreds-pages-long tutorials. Just open the IDE and start writing. Learning curve is close to zero. With Selenide you don't waste time on googling “how to make Selenium do that” - you can concentrate on business logic.

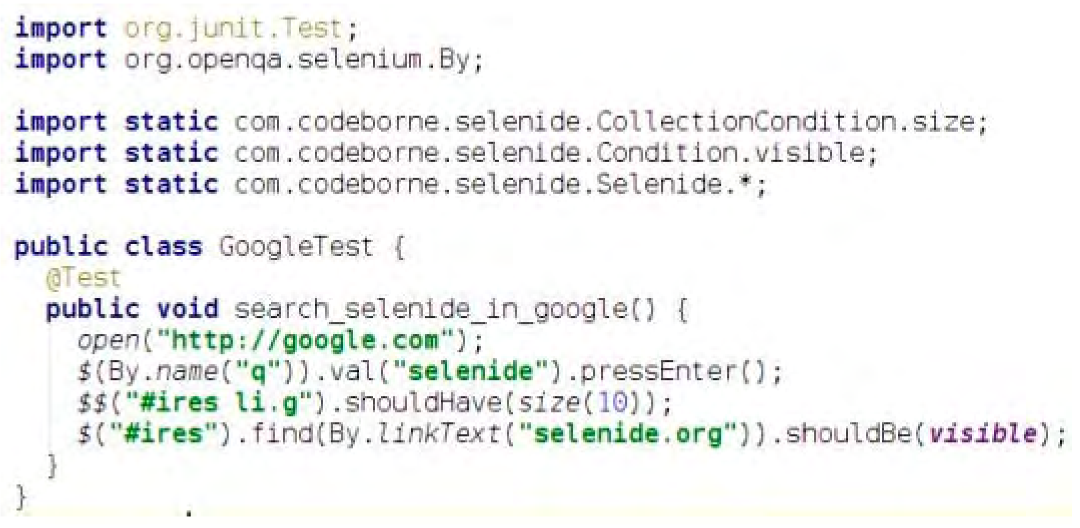
Selenide = a better Selenium

Web Site: http://selenide.org/
Version tested: Selenide 2.6.1
Language: Java and other JVM languages: Scala, Groovy etc.
System requirements: Windows, Linux, OS X, Android etc. - wherever Selenium works
License & Pricing: Open-source, free, LGPL 3.0
Support: Selenide is created and supported by Codeborne company. There are two communities in Google group:
* English: https://groups.google.com/forum/?fromgroups#!forum/selenide
* Russian: https://groups.google.com/forum/?fromgroups#!forum/selenide-ru

Introduction to Selenide

Ideally, we should write several types of tests for our applications: unit tests, functional tests, integration tests etc. For web applications we typically write scripts that open the application in web browser (either real or headless) and start clicking buttons. We call them user interface (UI) tests. The most common tool for UI tests in Java today is Selenium WebDriver. Though it is a great tool, it still can be improved. We were missing expressibility and stability in our tests, as well as we were tired of timeouts and lack of Ajax support. That's why we created Selenide - a dedicated library for UI Tests on top of Selenium.

To feel the taste of Selenide, take a look at the Google Test:



```
import org.junit.Test;
import org.openqa.selenium.By;

import static com.codeborne.selenide.CollectionCondition.size;
import static com.codeborne.selenide.Condition.visible;
import static com.codeborne.selenide.Selenide.*;

public class GoogleTest {
  @Test
  public void search_selenide_in_google() {
    open("http://google.com");
    $(By.name("q")).val("selenide").pressEnter();
    $$("#ires li.g").shouldHave(size(10));
    $("#ires").find(By.linkText("selenide.org")).shouldBe(visible);
  }
}
```

Isn't it concise?

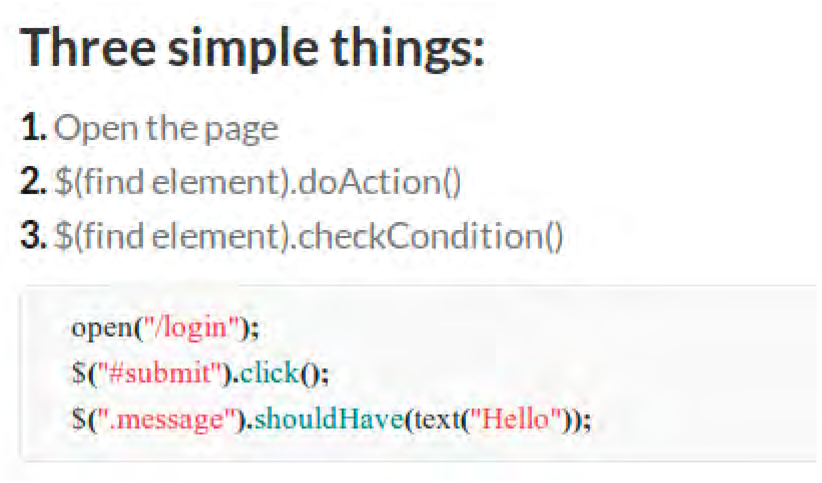
Selenide API

Poor software *doesn't have* documentation.
Brilliant software *doesn't need* documentation.

We are proud to claim that Selenide is so simple that you don't need to read tons of documentation.

The whole work with Selenide consists of three simple things!

Using Selenide is as simple as: $(selector).do()



Three simple things:

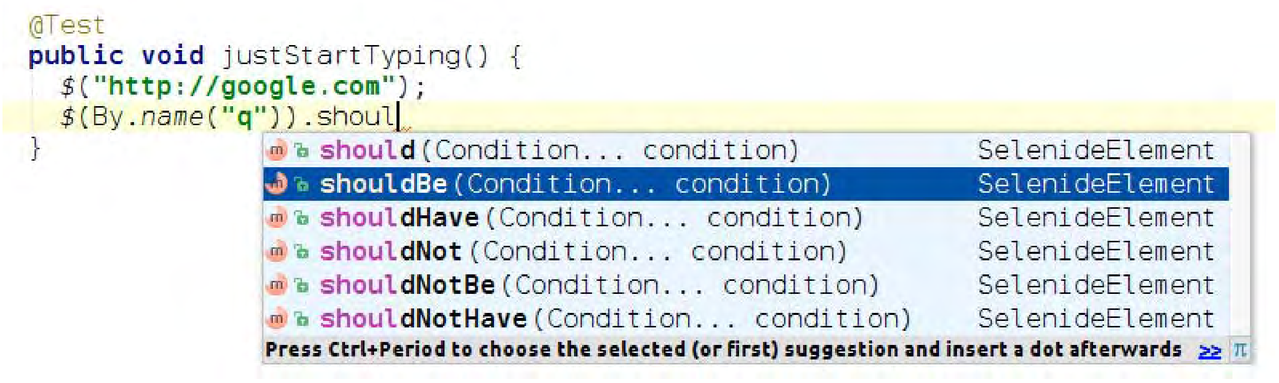
1. Open the page
2. $(find element).doAction()
3. $(find element).checkCondition()

```
open("/login");
$("#submit").click();
$(".message").shouldHave(text("Hello"));
```

Use the power of the IDE

Selenide has been designed to be very easy usable from any IDE without the need for reading documentation. You just write $(“selector”). - and the IDE suggests all possible variants once you pressed the dot.

You can even start typing some word - say, “ente” - and the IDE suggests all possible methods containing “ente” - “pressEnter” in this case.



```
@Test
public void justStartTyping() {
  $("http://google.com");
  $(By.name("q")).shoul
}
```

```
shouldd(Condition... condition)          SelenideElement
shouldBe(Condition... condition)         SelenideElement
shouldHave(Condition... condition)       SelenideElement
shouldNot(Condition... condition)        SelenideElement
shouldNotBe(Condition... condition)      SelenideElement
shouldNotHave(Condition... condition)    SelenideElement
Press Ctrl+Period to choose the selected (or first) suggestion and insert a dot afterwards >> π
```

Ajax and timeouts

The most annoying problem of UI Tests is timeouts. Testers live in a world without Ajax in their dreams. The test that you wrote today can work tomorrow, and next week, and even next month, but sometimes it will fail, just because the CI server is in a bad mood. Some Ajax request took a little bit longer time than before; some JavaScript happened to be a little bit slower; another process was run at the same time with your tests and occupied server CPU - and boom, your test is red! We spent a lot of hours if not months digging in these problems.

The typical solution is to use “sleep” or “wait_until” methods in your test. Though it usually helps, we do not live in “sleep world” in our dreams! Ideally, we should concentrate on business logic when writing tests, without the need to bother about WebDriver lifecycle, timeouts, sleeping, waiting etc.

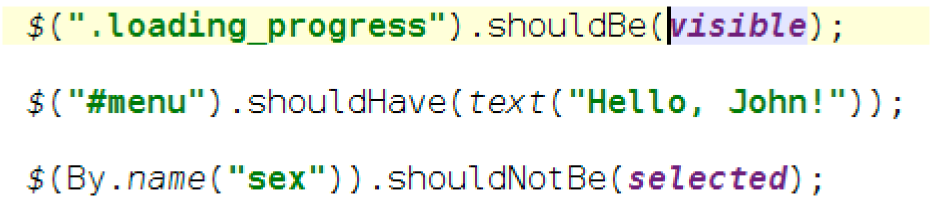
With Selenide you can concentrate on business logic. Forget Ajax!

How it works?

You are probably surprised about the simplicity of the solution provided by Selenide against the timing problems. No sleeps and waits anymore! Every *should* method just waits for a few seconds if needed.

When you write
```
$(“#menu”).shouldHave(text(“Hello”));
```

Selenide checks if the element contains “Hello”. If not yet, Selenide assumes that probably the element will be updated dynamically soon, and waits a little bit until it happens. The default timeout is 4 seconds, which is typically enough for most web applications. And of course, it is configurable.

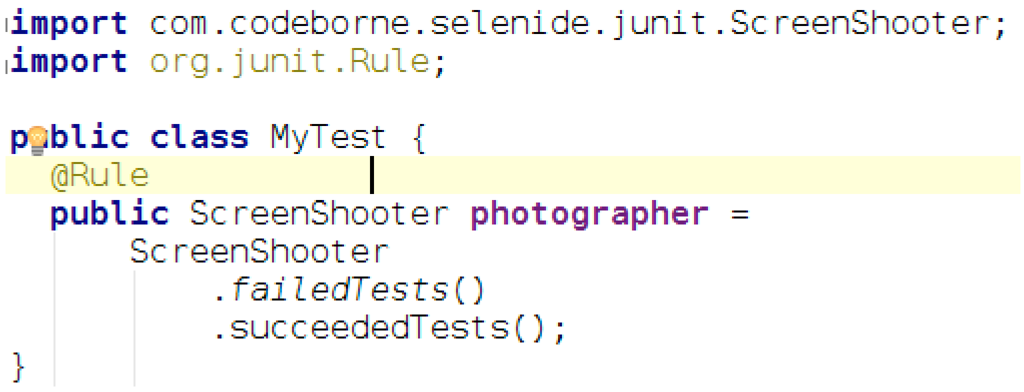

```
$(".loading_progress").shouldBe(visible);

$("#menu").shouldHave(text("Hello, John!"));

$(By.name("sex")).shouldNotBe(selected);
```

In the example above, you can see that with Selenide you can check that an element does NOT match the criteria: element is not checked, element does not contain “error” word, element has disappeared. This would take multiple lines of code including try/catch with pure Selenium. With Selenide you can do it with just one line of code.

Screenshots

Screenshots can help in case of any problems with UI Tests. They can also be used in reports, documentation, demonstrations etc.

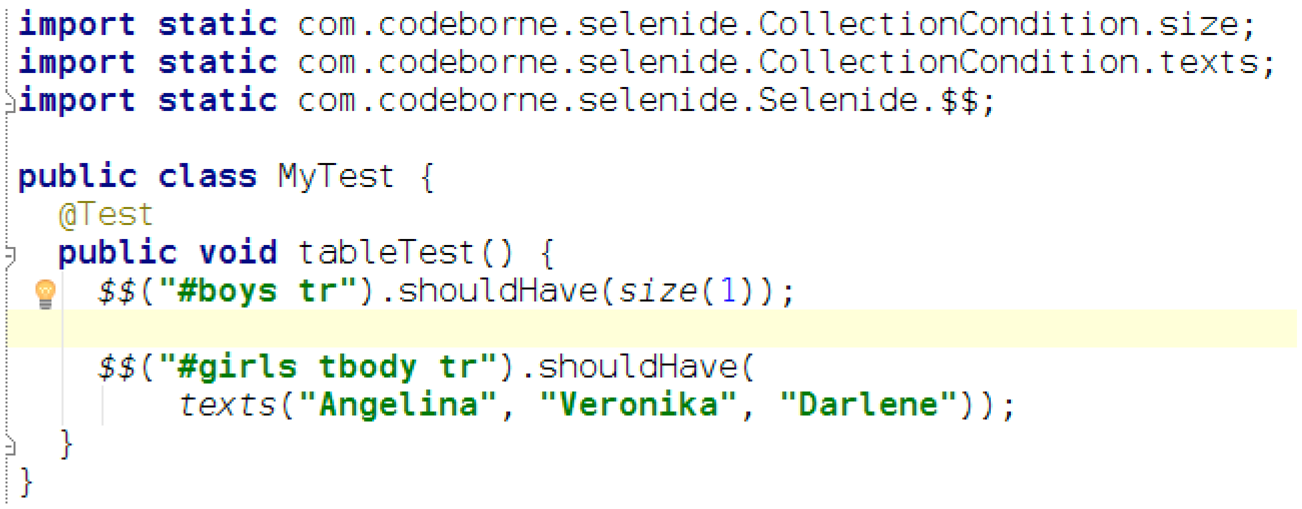
Selenide allows you to take screenshots after every failed test and after successful tests also if you want. You can do it with just one line of code:



```
import com.codeborne.selenide.junit.ScreenShooter;
import org.junit.Rule;

public class MyTest {
  @Rule
  public ScreenShooter photographer =
      ScreenShooter
          .failedTests()
          .succeededTests();
}
```

Working with collections

Sometimes it is useful to check the whole list of elements - say, rows of a table. Selenide allows doing it with one line of code using $$ method:



```
import static com.codeborne.selenide.CollectionCondition.size;
import static com.codeborne.selenide.CollectionCondition.texts;
import static com.codeborne.selenide.Selenide.$$;

public class MyTest {
  @Test
  public void tableTest() {
    $$("#boys tr").shouldHave(size(1));

    $$("#girls tbody tr").shouldHave(
        texts("Angelina", "Veronika", "Darlene"));
  }
}
```

The $$ method returns collection of elements that can be checked against some criteria.

Page Objects

QA engineers love Page Objects. Every time I speak about Selenide the first question is: “Does Selenide support Page Objects?”

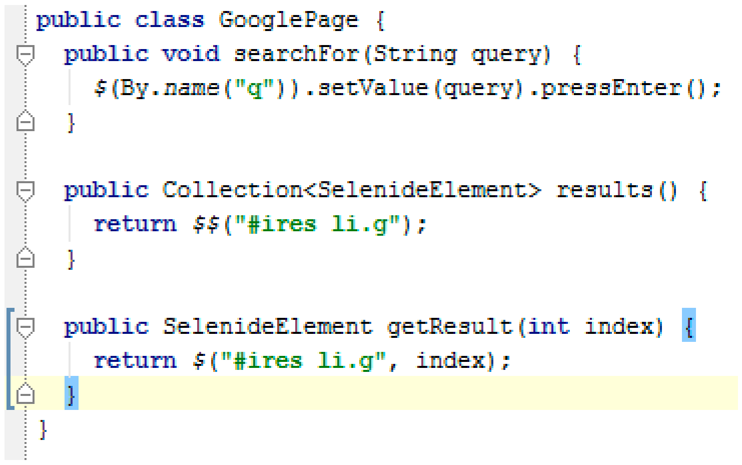
The short answer is: **yes, Selenide supports page objects**.

The long answer is: page object is not a thing that needs some special support. Test engineers are accustomed that their page objects need to extend some superclass, and use @FindBy annotations, and pass WebDriver instance to the constructor, and use PageFactory to initialize elements, etc.

It is a crap!

All this stuff is needed only because of lack of concise API. With Selenide you have got a concise API, therefore you do not need to do anything special to use page objects. No annotations, no page factories, no super-classes. Just encapsulate the logic of elements and pages to separate methods and classes, as in any other kind of programming.

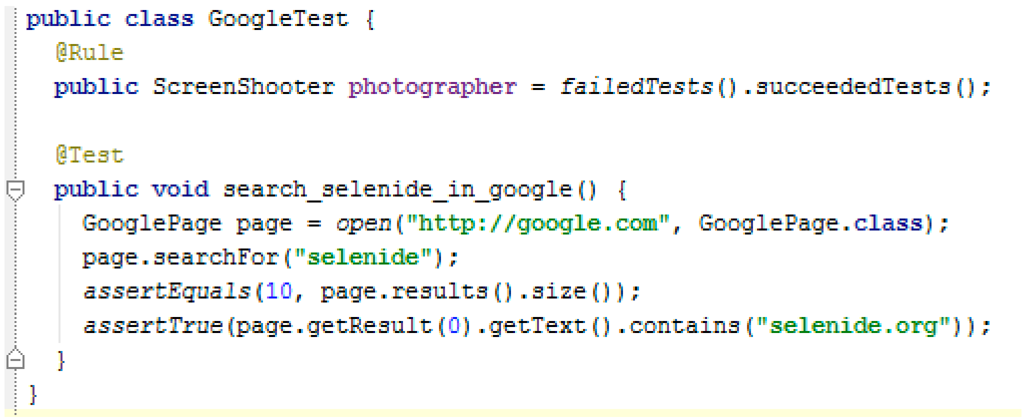
Watch this:



```
public class GooglePage {
  public void searchFor(String query) {
    $(By.name("q")).setValue(query).pressEnter();
  }

  public Collection<SelenideElement> results() {
    return $$("#ires li.g");
  }

  public SelenideElement getResult(int index) {
    return $("#ires li.g", index);
  }
}
```

It is a page object for GoogleTest that was provided at the beginning of this article. As you see, it doesn’t need neither annotations, constructor, PageFactory nor WebDriver instance. It is just a class that encapsulates logic of working with elements of a page. That’s how developers should create domain objects, and that’s how testers should create page objects.

By the way, this is how the test using GooglePage object could look like:



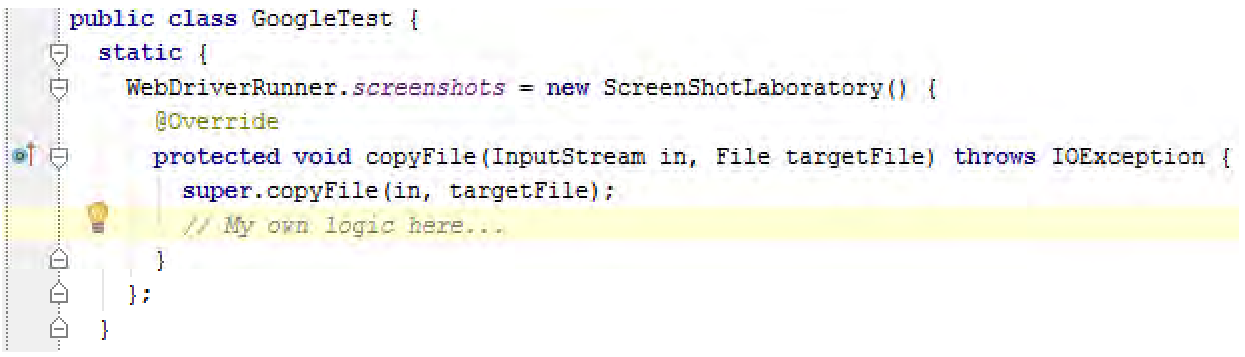
```
public class GoogleTest {
  @Rule
  public ScreenShooter photographer = failedTests().succeededTests();

  @Test
  public void search_selenide_in_google() {
    GooglePage page = open("http://google.com", GooglePage.class);
    page.searchFor("selenide");
    assertEquals(10, page.results().size());
    assertTrue(page.getResult(0).getText().contains("selenide.org"));
  }
}
```

Customization

There is an opinion that the primary failure of many tools (like Maven) is the impossibility to customize behavior. You can write your own plugins for Maven, but you cannot change the order of lifecycle phases in your build.

Inspired by Martin Fowler's “Internal Reprogrammability” article, we designed Selenide so that every little piece of its logic can be easily customized. There is no private methods in Selenide, you can supplement or override any method.

For instance, if you don't like how Selenide takes screenshots, you can override this logic by providing your own version of ScreenShotLaboratory:

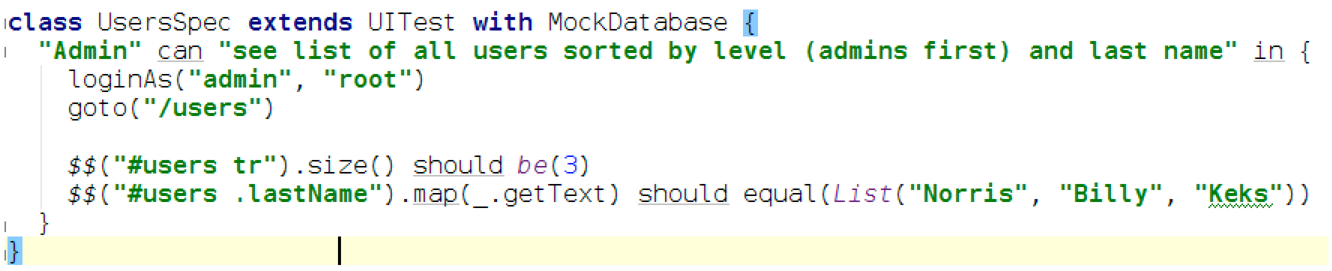


```
public class GoogleTest {
  static {
    WebDriverRunner.screenshots = new ScreenShotLaboratory() {
      @Override
      protected void copyFile(InputStream in, File targetFile) throws IOException {
        super.copyFile(in, targetFile);
        // My own logic here...
      }
    };
  }
```

Selenide in Scala

As a Java library, Selenide can be used in any JVM language.

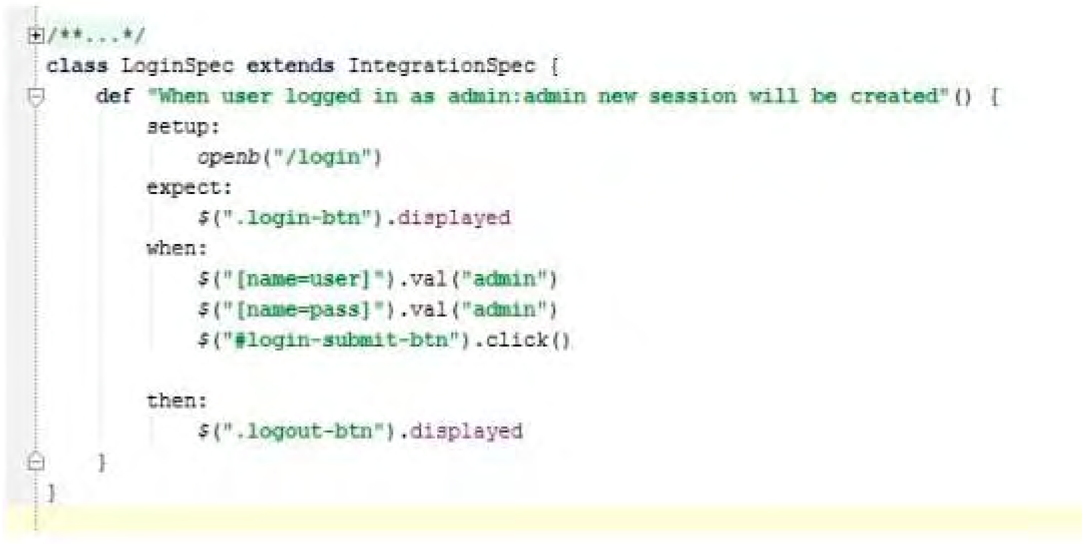
This is how ScalaTest+Selenide looks like:



```
class UsersSpec extends UITest with MockDatabase {
  "Admin" can "see list of all users sorted by level (admins first) and last name" in {
    loginAs("admin", "root")
    goto("/users")

    $$("#users tr").size() should be(3)
    $$("#users .lastName").map(_.getText) should equal(List("Norris", "Billy", "Keks"))
  }
}
```

Selenide in Groovy

And this is how Spock+Selenide looks like:



```
/**...*/
class LoginSpec extends IntegrationSpec {
    def "When user logged in as admin:admin new session will be created"() {
        setup:
            openb("/login")
        expect:
            $(".login-btn").displayed
        when:
            $("[name=user]").val("admin")
            $("[name=pass]").val("admin")
            $("#login-submit-btn").click()

        then:
            $(".logout-btn").displayed
    }
}
```

(thanks to Alexey Kutuzov for this example).

I also think that Spock and Selenide are really awesome together!

Tools similar to Selenide

Selenium WebDriver is a browser driving tool, not a testing tool. That's the reason why several tools have been created on top of Selenium like Thucydides, Yandex HtmlElements, fluent-selenium, FluentLenium, Watir-webdriver to name some of them. Selenide is one of them.

I am often asked about Selenide competitors. We are not competitors. We all do carry the same mission. If you write effective automated tests for your application, I am already happy, whatever tool you use for this. We created Selenide for those who appreciate simplicity, conciseness, expressivity and clean code.

You code it - you test it

At the end I want to deliver a really, really important message to you. No tool in the world can make your development effective - methods can. The name of this journal is “Methods & Tools”, but the methods are the most underestimated.

Why developers should write automated tests.

We really believe that software developers should write automated tests by themselves. It makes them feel the responsibility for the code. And it allows them to safely change the codebase that immediately leads to higher quality of code.

And the main goodness for tests is that developers create a simple architecture and clear design when they are forced to write tests first.

Compare:

- Poor testers need to open login page, write username and password and click login button 1000 times before every test. It makes tests incredibly slow.
- Developers create a “/fakeLogin/as/john.smith” URL that works only in test environment, and allows to login as any user faster than your eyes blink.

And even more,

- Poor tester writes 20 UI tests that try to login with 20 different invalid emails in order to assure that email validation works. These 20 tests take the whole minute.
- Developer writes 2000 unit tests that tests email validation in 0.1 second. And one UI tests that verifies showing of error message on the login page.

Testers just cannot do it as they treat the system as a black box. That's the reason why developers should help them by writing (at least some) automated tests by themselves. In this case developers create a testable architecture.

Happy coding, with right tools and right methods!

References

Selenide homepage: http://selenide.org

Selenide Harlem Shake: http://selenide.org/2013/08/29/selenide-harlem-shake/

Article about Selenide in RebelLabs (16.10.2013):
http://zeroturnaround.com/rebellabs/if-you-use-selenium-for-browser-based-ui-acceptance-testing-you-might-like-selenide/

Selenide presentation at Nordic Testing days (6.06.2013):
http://selenide.org/Valdo Purde Workshop Automated testing with Selenide.pdf

Martin Fowler, "Internal reprogrammability" (10.01.2013):
http://martinfowler.com/bliki/InternalReprogrammability.html

Discover STARCANADA, the premier software testing conference, April 5-9, 2014 at the Hilton Toronto. Experience the most up-to-date information, tools, and technologies available in the industry today. Reserve your seat today and receive a $50 Amazon.com gift card, plus save up to $300 with Super Early Bird pricing. Register non on http://vlt.me/matsc14

STAREAST, May 4–9 2014, Orlando, FL. Discover STAREAST — the premier software testing conference that brings you the most up-to-date information, tools, and technologies. Join industry experts and peers in the test and QA community for a week jam-packed with learning sessions. Mention promo code SW13MTE for an additional $200 off. Register on http://vlt.me/se14mtt

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

METHODS & TOOLS is published by **Martinig & Associates,** Rue des Marronniers 25,
CH-1800 Vevey, Switzerland Tel. +41 21 922 13 00 Fax +41 21 921 23 53 www.martinig.ch
Editor: Franco Martinig ISSN 1661-402X
Free subscription on : http://www.methodsandtools.com/forms/submt.php