# METHODS & TOOLS

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

Spring 2014 (Volume 22 - number 1) www.methodsandtools.com

## Does Experience Help in User Experience?

Some of the older readers might remember software development in the 20th century where end-user interaction with computer was performed using 80x24 characters terminal screens where the definition of user experience was making sure that your F(unction) keys were properly standardized across the screens. Does anyone still press F1 for help today? At least there are still these "Fx" keys on the keyboard. Those were days were one of the last thing you wanted was that somebody put his (obviously greasy) finger on the screen ;O).

Today's user interface and user experience designers are faced with different challenges, as applications might have to work on tiny mobile screens or 27' high resolution monitors. People speak about intuitive user interfaces, but I am a little bit dubious about the "intuitivity" as we see also some fashion influencing interface design. The first users of a mouse didn't know how to handle it. In those times there was still an actual ball in the mouse to track its moves and some people were using the mouse upside-down, making the ball move with their fingers. Today, the existence of hidden scroll bars in some applications like Facebook or Rdio seems also to have little to do with intuition as I my mouse pointer meanders on the screen to locate where the actual scroll bar could be... if there is one. It might be one of these "you know that you are old when..." things. The point is that there is often training and habits on how to use software than intuitivity. This being said, some screen designs work better than others for specific task or devices.

The multiplication of screen sizes forces the developer to make choices, sometimes displaying different applications subsets on different devices. Some applications might be driven with one thumb, but accountants could prefer keeping their finger on a numeric keypad. I don't think that Apples accountants use an iPhone for their activity, even if there are some rumors that they are very creative with it ;O) Two good consequences can derive from this situation: the necessity to give more importance to the use in user experience and the increased need to separate the user interface logic from business logic due to multiple interfaces for the same back-end services. These two global activities, user experience and software architecture, could be challenging to integrate in an agile software development approach that sometimes focuses mainly on the next iteration results. Their quality is however a major component of the long-term success and health of your application.

[illegible]

## Inside

## Distribution Sponsor

STAREast Conference - Click on ad to reach advertiser web site

# The Principles of Done

Phillip Cave, Enterprise Agile Consultant
SolutionsIQ, http://www.solutionsiq.com/, @SolutionsIQ

Simplicity. Why do we, in general, get so hung up on getting stuff done?

There are two things that I wish everyone could experience. One of them is driving on the Autobahn in a Porsche 911 at over 200km/hour. The other is for software engineers to build and deliver product from scratch in a technology startup.

If the first wish came true, then I would not be stuck behind slow drivers in the fast lane. *It is a PASSING lane people, and I mean pass within the next hundred feet, NOT mile. Pass and move over, don't police me by thinking I should obey the speed limit (speed limits are suggestions in my mind).* Thus, I love the Autobahn. Germans know how to drive.

Another wish is for all people in the software world to experience living and succeeding at a product company startup. Like driving on the Autobahn, startup companies must move. Startups must get stuff done - or they will not be around. When they finally ship, it is in hopes of being successful. My experience in a startup taught me the fine balance of what we now call a minimal viable product.

When I am coaching companies on implementing Agile, I often find myself saying, *"You forgot what it was to be a startup"*. All companies are startups at one point in their history. Some people who started the company may still be around, but often that is not the case. Something happens along the way. People within a maturing startup look to become efficient over effective. The leadership looks to optimize in small ways but forgets to look at the larger picture. The product grows and the value stream to deliver features grows. This growth creates a dynamic of increased coordination and transaction costs. These costs appear in the guise of a lot of documentation about how to manage the coordination and transaction costs. The thinking is that we may forget what we said we wanted, so we'd better write it down, create an approval process, and create a lot of artifacts to coordinate our activities and agreements.

I am always fascinated by this growth pattern. Having experienced startup thinking and large corporate thinking, I began exploring why some patterns emerged as companies grew. Part of this pattern is found in titles, trust, and an attempt to create security.

There is work (the features we deliver to customers) and there are people to do the work. At my first startup, I managed the entire value stream as a software engineer. I did all the work. The discovery, analysis, exploring options, prototyping, UX, design, coding, testing, writing (did I leave anything out? I did that too.). So, too, did the other two engineers I worked with to architect the product from scratch. We each took responsibility for different areas of the business domain to create the product. We collaborated multiple times per day, every day, on the technical architecture and design. We made decisions in the moment, made adjustments, and got stuff done. We iterated on the product every single day to get more and more stuff done. We had no ginormous specification - we had a collection of user-based experiences. We had a business domain expert on staff with whom we collaborated daily so that we could get stuff done, every day.

Growth created specialization in a lot of companies. Instead of one person being able to do the work (e.g. implement a customer feature), we began segmenting that work. The customer still wanted the same features delivered, but how we delivered it changed.

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**

Specialization is not likely to go away, and I am not advocating that it should. I am reminding people that with change comes managing change. How we manage that change is integral to getting stuff done as effectively as possible. (Please note that I am NOT using the word "efficiently" - there is a difference). We tried to take a manufacturing model and apply it to software engineering and product development in hopes that it would help us manage that change, thinking that has prevailed for many years.

We talk a lot about getting to done, being "done done", exit criteria, acceptance criteria, and definition of done, definition of ready, and so on. These are all good implementations of the practice of getting done.

A couple of years ago, I came across a poster at Microsoft about the "cult of done". Someone put it up to remind the team that we were there to get stuff done regardless of our specialization. Microsoft was obviously a startup at one point, and then a lot of stuff happened and new people (a lot of new people) came on board. Those new people had their own comfort levels regarding how to get stuff done, which eventually led to creating a lot of specifications about making conversations more "efficient". A lot of command-and-control-style leadership was instituted, as at many companies in the industry, to help those people feel in control of this very complex environment. Like many organizations I consult with, Microsoft forgot how to be a startup. Not in all areas, mind you, but certainly in a lot of them.

All of my work (about 6 years) at Microsoft as both a consultant and FTE was about reminding teams and leadership of what a startup feels like, how startups (and larger companies that still operate like that way) succeed, and how they have the ability to run circles around larger companies with entrenched thinking. I had supporters and those who wondered what the heck I was doing there. I was called a Principal Program Manager to boot - just a title to me.

This article is meant to help you remember how to get stuff done just as if you were in a startup … remember those days? If you have never worked at a startup, much of what you read here will be new to you but perhaps it will resonate with the way you think of the world or speak to your inner systems-thinking views. My goal is to describe my experience in hopes that it spurs creating your own, right where you are.

As you read, please keep in mind that in all things we need a goal or target. Startups succeed, just as many companies do, because they understand reaching a goal and target. Where they differ is that the culture of a startup more easily adjusts to learning. Getting to done does not mean that we don't think about where we are going - we actually do. It means we create goals based on the information we have today, and then adjust those goals as necessary when we receive new information. Would a startup take four months to determine its goals? Probably not. So with this thought, please read on.

These principles of being done reflect my experiences during my 24-year career in the software industry, startups and otherwise. My inspiration comes from a very simple list in a blog post entitled The Cult of Done Manifesto by Bre Pettis and Kio Stark [1], to which I have added my thoughts.

## 1. There are three states of being: Not knowing, action, and completion

I will only take a moment to discuss the "not knowing" state. Your sole purpose when faced with "not knowing" is to start knowing, quickly. It fascinates me when people batch up work into ginormous requirements documents and treat all of those requirements as the same. Instead they should start working on what they absolutely know needs doing.

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®

Then they can figure out what they don’t know by creating short experiments to begin knowing. If a team is faced with considerable risk due to a lack of knowing, that team should start work on experimenting to gain knowledge. If you don’t know something that you really ought to know, then start to know it! That is the action part. Writing documentation is not action. Well, technically you are in motion as you type, but what are you getting done? A lot of words. How does that help the customer? Does the customer pay you to write a spec or deliver software? Yeah, yeah, you can argue that writing the spec helps me get to done and I will tell you that is a bunch of crap. Yes, you need to know what the feature is, what done means for the feature, what in initial look and feel may be, but that does not have to be in a 500-page spec - it just needs to be a notation somewhere. That somewhere could be a white board or a list of stories in a spreadsheet or jotted down in a collection of features and stories in the cloud. I wrote multi-million dollar software without a 500-page spec, so don’t tell me you need it. We had a white board and conducted A LOT of experimentation with design and writing code. Actually doing something is action. Discovering something cannot be accomplished by writing a specification. You can’t get to done until you start, so start building or start learning what to build by creating experiments that deliver information. In both of those cases, begin creating something so you can get to done.

**2. Accept that everything is a draft. It helps to get to done.**

Too often I have heard the words, “But we might have to throw it away if we don’t get it right,” to which I reply “…and … <pause for effect> how do you know it is wrong or right until you get feedback? You can’t get feedback on what the thing should do until you actually start to build it. So guess what? You will have ‘throw-away’ work because you are going to learn later what to keep and what not to keep. So quit your whining and get it done.”
A favorite movie of mine is Disney’s *The Kid*. Bruce Willis is an “image consultant”, and one of my favorite lines comes at the moment he tells a political figure to stop her whining. “Somebody call the waaaaaaambulance,” he says. I say that internally much of the time. Instead of focusing on what can’t be done, focus on what can be done. Some of my most fun I’ve had in architecting software products has been in discovering and saying out loud, “Well, that was complete crap! Next option?” A startup-thinking organization of people focuses on what can be done and then experiments to validate those assertions.

**3. There is no editing stage.**

Well, there actually is. It is when you are typing and you hit the backspace and retype. Other than that, build in small increments, validate in small increments, get feedback in small increments, deliver in small increments … get the idea? Small feedback cycles are much more conducive to achieving success. The longer you edit, the longer it takes to get to done.

Editing (other than the excellent people helping me edit this article, mind you ☺) is an excuse to postpone being done. I was once on a project at Microsoft that entailed building out a business scorecard. The new manager had just come from another area of the company and was what I call an entrenched thinker - stuck doing things the only way he knew how to, in big batches. Despite his many years in the industry, he had no other experience. When he observed the quick/iterative process I had implemented and the empowerment to start and do something that came along with it, he commented about how cool it was to see a working model within a few weeks instead of months. The interesting thing about this guy was that he could not let go of the old big-batch thinking model, so my functional specification was woefully behind what the team was actually accomplishing. I met with the team every day to discuss options and then build them. They did not wait for me to produce a “how to build” document when the thing was already in the process of being built.

This is how it should be. I left as soon as my stock vested and moved on to helping companies that actually want to learn how to build things instead of edit things. Sigh. I recognize it is a comfort-factor thing. There are people in the world who want to control everything and are afraid to try anything new, and then there are companies like Apple and certain divisions within Microsoft (like Xbox) that just do stuff and get it done. In those places, editing does not exist, only getting stuff done.

**4. Pretending you know what you're doing is almost the same as knowing what you are doing, so just accept that you know what you're doing even if you don't, and do it.**

Give yourself permission to start. Give yourself permission to experiment. Give your intuition some credit. Give yourself permission to explore what you don't know and play so that you discover what you don't know, and then get on to the other stuff you don't know by experimenting.

Hell, there was plenty I didn't know when I was starting out, but I was one of those people who loved to play around with stuff. I am a natural "What does that do?" and "Where does that road go?" kinda guy. I was never a "fake it 'til you make it" person - I'm an "I don't know, but let's find out!" person.

Once I was on a short gig as a software engineer and the person who was responsible for indoctrinating me into the culture and technology stack was impatient. He just wanted me to follow his script step by step without explaining anything and without letting me experiment and learn by experimenting. I left there as quickly as possible. I call those kinda people lifers. They plant themselves at a conservative-thinking culture and retire. Not exactly disruptive, but not disruptive enough (or at all). Disruptions get us to new places - pretending that we know what we are doing and just doing it is what gets us there. They create innovation and transform old/crappy ways and products into new/better products and services and ways of doing stuff and getting it done.

You know how to start something, right? So pretend you know what you are doing and just start, which leads to…

**5. Banish procrastination. If you wait more than a week to get an idea done, abandon it.**

When I coach teams, I tell them that if they spend more than a week trying to figure something out they are wasting their time and the time of the business. Figure stuff out by doing stuff in small experiments: Build and experiment. If you allow other stuff to get in the way, then the thing you were working on must not have been that interesting or important. Move on to a variant of the thing so that you actually get some value created, which really is the purpose of done … value creation.

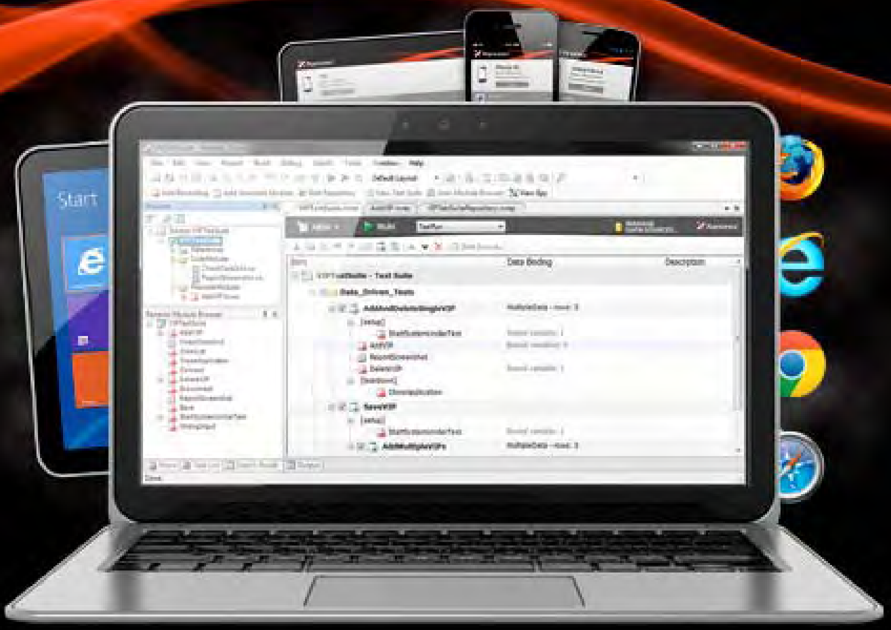
I "love coach" friends to look at what people value by observing where they spend their time and money. If it is not spent on you, then it is time to find someone else that does spend time and money on you. It is the same with software products: Spend the most time and money on the stuff that matters most and get it done. Be nice to the feature you don't like by just dumping it. Procrastination may occur anywhere in the value stream and is best described for management as the cost of delay. In lean organizations, this metric is measured throughout the production process to objectively track the cost associated with making or not making decisions - delayed decisions create overhead and missed-opportunity costs.

Ranorex Automated Testing Tool- Click on ad to reach advertiser web site

Ranorex®

Automated Testing of

Desktop. Web. Mobile.



- ✓ Use connectors for data-driven tests
- ✓ Build robust test automation frameworks
- ✓ Generate EXEs for pure flexibility
- ✓ Write custom code in C# or VB.NET

Why Use Ranorex

www.ranorex.com/why

Award-winning test automation tools provide seamless testing of a wide range of desktop, web and mobile applications.
HTML5 | IE | FF | Chrome | Safari | WPF | Flash/Flex | Silverlight | Qt | SAP | .NET | MFC | Delphi | 3rd Party Controls | Java

When management and executives ask me what they should measure as they adopt Agile ways of delivering software, I tell them to quantify cycle time and the associated costs of delay. The idea of banishing procrastination is to minimize the costs of delay - delay of requirements, delay of estimates, delay of decisions, delay of coding, delay of testing, delay of implementing, and so on.

Delay means we are not done. Not being done costs money. So get done.

**6. The point of being done is not to finish but to get other things done.**

The point is to create value. The faster we can get one thing done, the faster we can get "done done" with other stuff. I coach teams on a pet phrase: "Stop Starting and Start Finishing". If you have too many things in process, then nothing is getting done. And if nothing is getting done, it means you are just procrastinating and we now know that procrastinating has a cost associated with it.

During my 24 years in the software industry, I have never had stuff I could not deliver. Well, except for that time at what is now Verizon Wireless and after our second merger during the days of cellular company buyouts and consolidations. I was essentially day trading while looking for my next job - just sitting on my ass as my position was being outsourced to a company in India. And that is when I went to my first startup, fed up with large corporations. It was there that I learned how to truly deliver software products. I am super-grateful for my time at my first startup. It taught me one thing well: You might look busy, but guess what? You are not providing value until you get stuff done, and you are not effective until you can move on to get more stuff done … so get it done. Once one thing is finished, it means we have the capacity and availability to get other things finished. And getting finished with the first release meant that we could get more customers to buy our product and get the next million-dollar contract. The more we finished, the more we had to do and the more customers we had.

**7. Once you're done, you can throw it away.**

Sure, you take pride in creating stuff.  I certainly did when I was creating solutions, especially at that first startup. But once created, then what? You move on to the next. At some point, the stuff you created will need to be changed. Or it will get thrown away because requirements have changed. Products evolve. Evolution means we will improve it, at least we should. It means that the old stuff we are holding onto is stinky and moldy. Let it go. The world changes. So should our software. We can get so emotional about it. Let economics like cost of delay guide your decisions.

Consulting in many industries and companies over the years, I have often seen this inability to move on or move quickly enough through delivery cycles. Once a feature is done, it is on the fast track to obsolescence. You are not going to get it perfect, so stop pretending you should. You are going to throw it away and build the next version at some point, which leads to …

**8. Laugh at perfection. It's boring and keeps you from being done.**

It means let the thing go, validate it, move on. Revisit it later when you have more information or have to adjust to something learned. This does not mean you abdicate quality!!! A saying I like is "The perfect is the enemy of the good." Perfection slows us down and causes us to second-guess and create blame. Perfection holds us back from making progress, from getting it in the hands of customers in order to validate what we created.

Perfection causes startups to go out of business. Adapting to learning causes startups to succeed. We learn by doing and exploring quickly. Perfection keeps us from learning more and learning faster. Perfection keeps us from learning what we should not do as well as what we should do. We are so busy trying to get something perfect that we delay learning, we delay releasing, and we delay succeeding.

Don't "perfect" yourself right out of business. Get something done, learn from it, and build the "good enough" version and release it. Again, this does not mean that it is buggy software. It means that I don't have to dwell on it or satisfy every customer, which is an interesting dysfunction I have seen in some of the companies I have consulted with over the years.

We can be so perfect at satisfying every customer that it creates dysfunctional decision making not based on what is best for the company. If I have a whiny customer that demands more than is economically feasible or if people in my own company subvert a sustainable sane pace to product delivery and do an end run around standard practices, this creates delay.

Just because a customer wants something does not mean we should build it to the detriment of the product or the teams building it. If it costs me more to have a customer then the value I am receiving, then it is time to think of replacing that customer with one that is more reasonable to work with. Instead of trying to create the perfect product for every client, create the best-quality product that meets most of the needs of most customers. Perfection costs more than the value you are getting out of it … in economics, they call that diminishing returns.

When you stop focusing on the perfect, you end up managing the work-in-process to stop starting things and focus on finishing things. Getting to done means focusing on the right activity at the right time to deliver and be done.

**9. People without dirty hands are wrong. Doing something makes you right.**

In lean thinking we are urged to "go to the gemba", which means go to the people doing the work because **they know best** how to change HOW we work. Design is not development. Get your hands dirty by doing something.

I do this every day in my career as an Agile management consultant. I am asked to go to the gemba when I "assess" a team. My version of an assessment is a little study and a lot of answering the "how do we?" questions - "How do we… re-estimate work, use story points vs. hours, create a portfolio budget with Agile, estimate large projects, change our organization design…?" I learn a lot about an organization, its teams, and its leadership by answering these questions.

Do something and get your hands dirty, whether you are a C-level exec, a manager, a developer, a tester, a program manager, or any other team member responsible for delivering products to market or supporting the business in your IT shop.

As an executive, your number one responsibility is to own the system in which your teams deliver products and create a culture of improving that system. As a manager or supervisor, your number one responsibility is to incrementally improve that system. Get your hands dirty.

When managers ask me what they're supposed to do in an Agile world, I tell them that middle management has the super-important role of creating the best possible system for development, a duty that should not be taken lightly. Apply lean principles to help the team succeed by getting your hands dirty. Improve the HOW of how teams deliver.

If you are a developer of products and you are not testing, you are not getting your hands dirty. Your number one job is to deliver quality product, not pass it off to someone called a tester. You should not even need a tester. You build it, you test it. Get your hands dirty and get it done. Working with large systems, I may have testers in my environment that focus on testing the system interfaces and larger integrations, so their hands are dirty all the time.

If you are a tester - either a manual tester or a software development engineer in test (SDET) - your number one job is consulting with teams about how to build quality into the system. Your hands are dirty all the time with some of the crap you are handed from developers who should be doing some of their own validation. You may be busy, but are you effective? Stop the line. Create a quality mindset by showing teams how to build quality as they go instead of inspecting it at the end.

The point is: take responsibility and get your hands dirty by getting quality stuff done. Quality environment, quality delivery system, quality code, and quality product.

**10. Failure counts as done. So do mistakes.**

Getting done means we learned something. Something either worked, or did not work, or worked in a way we didn’t expect. Failure means we can now move on to the next thing, which will either work or not work. [2] It is okay to fail as long as we learn from it - what to do as well as what not to do.

A saying I like from my time in lean operations at a health insurance company is “fail forward fast”. It means that sometimes failure happens and it is just fine, and it is better to find out sooner rather than later. This is because it helps us get to the done we want to arrive at. Riddle me this … if you are going to fail (and you sometimes will), would you rather fail quickly or slowly? Me, I want to do so quickly and learn what works. We fear failure in general. I believe this is because we may have been victim to management that punished failure instead of fostering an environment of learning. This leads to dysfunctional political posturing, CYA actions, and a legalistic “document everything up front” behavior in the belief this will mitigate failure. Here is a thought … what if we embraced learning and mitigating failure by finding mistakes quickly? This is just one of management’s responsibilities in a Lean culture: Create a learning environment and help teams find and correct mistakes quickly.

This speaks to an attitude of starting. Instead of being afraid to fail, which leads to not starting, just start already. Not starting is unacceptable failure. Trying something - learning from it and failing - helps me get other stuff that does not fail done.

I once dated someone who told me she made a mistake when it came to lying to me. That was not a mistake, it was a failure. A mistake is spilling your coffee. Mistakes are made when we don’t know any better or we are clumsy (or caffeine deprived). Failure takes purpose. So be purposeful in your failures. Meaning, run experiments. You may not be sure if the experiment will work, but it definitely won’t work if you don’t try it.

Try it, you’ll like it.

## 11. Destruction is a variant of done.

Some things should be destroyed, like old dilapidated buildings and zombies, and old crappy code … and zombies (cuz they just keep coming at you). Think of the mythical phoenix. Its destruction makes room for the next iteration or version. If you don't need it, destroying it creates less complexity. If you need to improve it, start killing it off and replacing it with the next version. It's okay to let go of bad investments and code that just is not allowing us to get stuff done as effectively as we want to.

My first startup did this. We destroyed the old platform and started a new one. (Thinking about it now, we should have called that project Phoenix! Sigh, missed opportunity.) The project was birthed because the old crappy architecture was not going to work at the enterprise level.

I am working with a customer that has what we call a legacy code base. I'm talking legacy. Old COBOL mixed with new Java (which isn't so new no more) and CICS and … and … and. Yeah, you get the picture. I can navigate these waters because I started off in the tech industry as a COBOL programmer and migrated to object languages to end the coding phase of my career. My advice to this customer is to start destroying the legacy stuff and replacing it with not-so-legacy stuff (let's face it - in one to three years, the new stuff will be legacy as well).

Destruction means we are done. In this case we want to be done really badly.

## 12. If you have an idea and publish it on the Internet, it counts as a ghost of done.

But ghosts have no real substance. This is exactly like a ginormous requirements document - it has no substance and it is a ghost of done. Done is taking what you wrote about in the requirements document and actually giving it life. Done = Life, and ain't life grand?!

Alternatively, if someone picks up your idea that you published on the Web and then creates something (like the thought leader behind Tesla, Elon Musk, publishing his short and succinct idea for a high-speed train called the Hyperloop), then you contributed to done but you didn't get it done. Since an idea is a ghost of done, why spend a lot of time trying to communicate it in a ginormous document when you can write it down as a placeholder, in a short sentence or two, which elicits real conversation that can lead to done?

I once wrote multi-million dollar selling software with two other people using a whiteboard for architecting and having A LOT of conversation. I snicker at documentation jockeys (heh heh). What a waste of time to think that putting a conversation into a document is a good idea. Publish a little, just enough to have a conversation. Thus I love one-pagers or story cards or small feature-driven documents. (I am speaking to documentation to deliver products, not documentation for the end user.) We do not need heavy documentation to tell us how to write code. We need a little documentation to communicate intent, what done means, and our decisions. But we need a lot of talking.

Write small and talk a lot, but mostly just get it done.

## 13. Done is the engine of more.

Finally being done with turning this blog series into an article means I can get to the other blog posts I want to write and customer work and improving our engagement model and and and …. Yeah you get the idea. Done = get more stuff done.

A story I like to tell is about working with a group at Microsoft. One of the PMs I collaborated with had a team that was faced with an ever-changing environment and specialty-based team members. During a standup, it was brought to the team's attention that a story was blocked because the specialty team member was busy setting up his test environment. The team asked if he had to be the one to set up the environment. The answer was no, he did not have to be the one to set up his test environment and that, in fact, another team member could accomplish that task while he unblocked the team by working on the story.

Was it efficient for him to set up his own environment? Probably. Was it effective? Absolutely not. That story was about focusing on the work and not the team member, and reaching the most effective way in which to complete the work.

Getting done in one area helped the team get done, which created a team focus on done. Sounds like a well-run engine to me.

<this space left intentionally blank> … *well, with the exception of actually containing the words "this space left intentionally blank" because then it is not really blank, is it?!* You are now done with your choice to read this article, now go get other stuff done. And may the force of done be with you … gesundheit!

**References**

1. The Cult of Done Manifesto by Bre Pettis and Kio Stark,
http://www.brepettis.com/blog/2009/3/3/the-cult-of-done-manifesto.html

2. Scott Adams' Secret of Success: Failure,
http://online.wsj.com/news/articles/SB10001424052702304626104579121813075903866

# Doing Valuable Agile Retrospectives

Luis Gonçalves, Ben Linders
www.benlinders.com, @BenLinders; lmsgoncalves.com, @lgoncalves1979

At the end of an iteration, typically two meetings are held: the sprint review (or demo) which focuses on getting product feedback, and the sprint retrospective which focuses on the team and the process used to deliver software. This article is about doing (Scrum) retrospectives, aiming to help teams to continuously improve their way of working.

The 12th agile principle states: "At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly". Agile retrospectives are a great way for teams to continuously improve the way of working. Getting workable actions out of a retrospective and getting them done helps teams to learn and improve.

To do agile retrospectives it is important to understand what they are and why you would want to do them. This helps you to facilitate valuable retrospectives and to "sell" retrospectives in your teams and motivate team members to actively and openly take part in them.

As a retrospective facilitator it is important to have a toolbox of retrospective exercises which you can use to facilitate a retrospective. This article describes some possible exercises that help you to facilitate retrospectives that deliver benefits to the teams that you work with.

This article is based on Getting Value out of Agile Retrospectives [1], a pocket book that contains many exercises that you can use to facilitate retrospectives, supported with the “what” and “why” of retrospectives, the business value and benefits that they can bring you, and advice for introducing and improving retrospectives.

You can download the book Getting Value out of Retrospectives free of charge from:

- InfoQ: http://www.infoq.com/minibooks/agile-retrospectives-value
- Leanpub: http://leanpub.com/gettingvalueoutofagileretrospectives

## What is an Agile Retrospective?

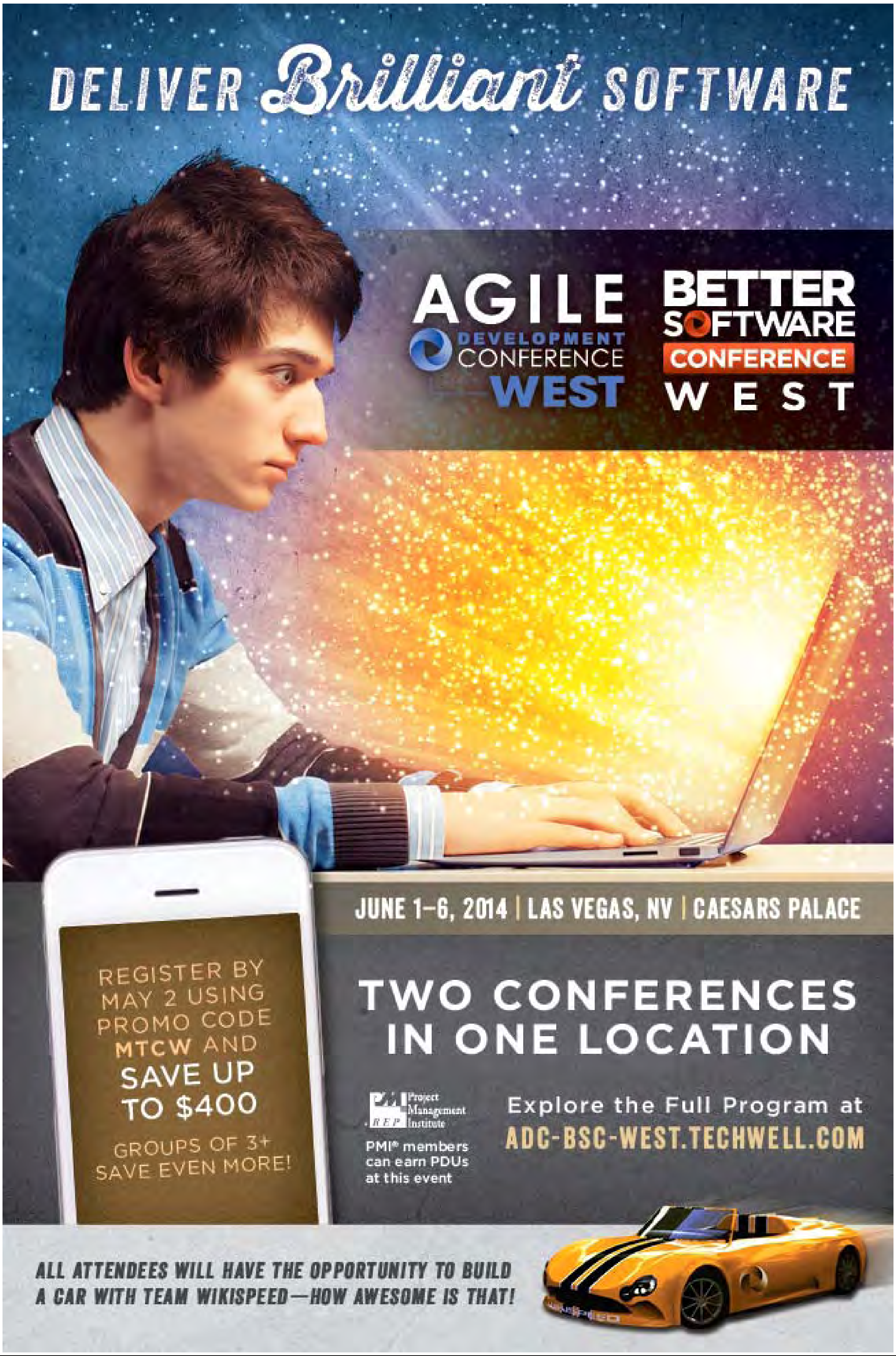
The agile manifesto proposes that a “team reflects on how to become more effective”. Teams use agile retrospectives to inspect and adapt their way of working.

A retrospective is normally held at the end of each iteration, but it can be done as often as a team needs it. It focuses on the team and the processes used to deliver software. The goal of retrospectives is helping teams to improve their way of working.

All team members attend the retrospective meeting where they “inspect” how the iteration has gone and decide what to improve and how they want to “adapt” their way of working and behavior.

Typically a retrospective meeting starts by checking the status of the actions from the previous retrospective to see if they are finished, and to take action if they are not finished and still needed. The actions coming out of a retrospective are communicated and performed in the next iteration.

Agile & Better Software Conferences West- Click on ad to reach advertiser web site


DELIVER Brilliant SOFTWARE
AGILE DEVELOPMENT CONFERENCE WEST
BETTER SOFTWARE CONFERENCE WEST
JUNE 1–6, 2014 | LAS VEGAS, NV | CAESARS PALACE
REGISTER BY MAY 2 USING PROMO CODE MTCW AND SAVE UP TO $400
GROUPS OF 3+ SAVE EVEN MORE!
TWO CONFERENCES IN ONE LOCATION
PMI Project Management Institute R.E.P.
PMI® members can earn PDUs at this event
Explore the Full Program at
ADC-BSC-WEST.TECHWELL.COM
ALL ATTENDEES WILL HAVE THE OPPORTUNITY TO BUILD A CAR WITH TEAM WIKISPEED—HOW AWESOME IS THAT!

To ensure that actions from a retrospective are done they can for instance be added to the product backlog as user stories, brought into the planning game and put on the planning board so that they remain visible to the team.

The retrospective facilitator, often the ScrumMaster, should have a toolbox of possible retrospective exercises and should be able to pick the most effective one given the situation at hand.

## Why Do We Do Retrospectives?

Organizations need to improve to stay in business and keep delivering value. Classical organizational improvement using (large) programs takes too long and is often inefficient and ineffective. We need to uncover better ways to improve and retrospectives can provide the solution. Many agile teams use retrospectives: to help them solve problems and improve themselves!

What makes retrospectives different from traditional improvement programs? It is the benefits that teams can get from doing them. The team owns the agile retrospective. They can focus where they see the need to improve and solve those issues that hamper their progress. Agile retrospectives give the power to the team, where it belongs! When the team members feel empowered, there is more buy-in from the group to do the actions which leads to less resistance to the changes needed by the actions coming out of a retrospective.

Another benefit is that the team both agrees upon actions in a retrospective and carries them out. There is no handover, the team drives their own actions! They analyze what happened, define the actions, and team members do the follow up. They can involve the product owner and users in the improvement actions where needed, but the team remains in control of the actions. This way of having teams leading their own improvement journey is much more effective and also faster and cheaper than having actions handed over between the team and other people in the organization.

How can you do retrospective meeting that delivers business value? A valuable agile retrospective identifies the most important things that a team wants to work on to improve their process. But what is most important? It can be the biggest, most current impediment your team has. You can do a root cause analysis to understand it and define effective actions. Maybe something is disrupting your team’s atmosphere and they can’t get a hold of it, in which case the “one-word retrospective” could help. Or it could be finding the reason why the current iteration failed, or why it was such a big success. You could investigate how to use the strengths that your professionals already have to improve further.

## Question Based Retrospectives

In this section we will discuss how to do a question based retrospective. The next sections will explore two more retrospective exercises: The Sail Boat Retrospective and the Strengths Based Retrospective.

Teams differ, and also the things that teams deal with can be different in each iteration. That is why there is no single retrospective exercise that always gives the best results. Also the risk exists that teams get bored when they are always doing retrospectives in a similar way. A solution to this is to introduce variation using different retrospective exercises. There are many different exercises that can be used in agile retrospectives, many of them are described in our book Getting Value out of Agile Retrospectives [1].

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

One exercise that is often used in agile retrospectives is to ask questions to the team, for instance about what went well and what can be improved. The answers to these questions can be clustered and used to define improvement actions that the team can do in the next iteration.

When doing a question based retrospective with a team that is new to retrospectives, you can use the four retrospective key questions [2]. These questions come from the book Project Retrospectives: A Handbook for Team Reviews, by Norm Kerth [3]. The questions are:

- What did we do well, that if we don't discuss we might forget?
- What did we learn?
- What should we do differently next time?
- What still puzzles us?

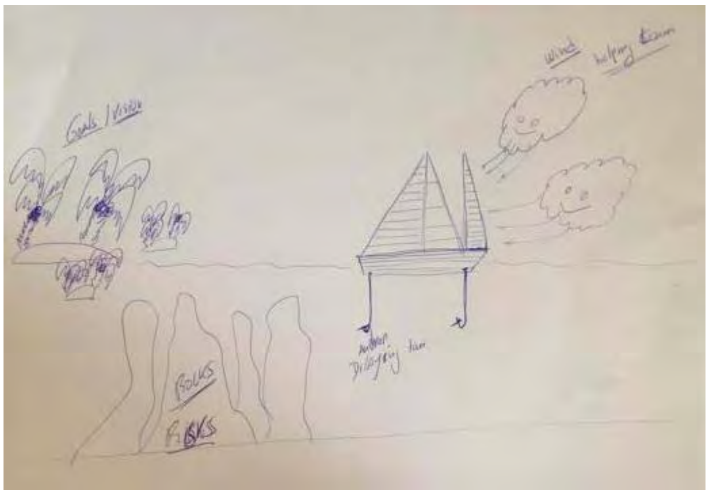
Asking questions is an exercise that is easy to learn, but the effectiveness depends on the questions that you ask to the team. The trick is to pick the questions that help the team to gain insight into the primary issues that they are having, and questions that help them to visualize their improvement potential.

Use open questions to elicit answers that provide more information, and use follow up questions to help teams get insight into what happened. Ask for examples to make situations concrete, summarize answers to build a shared understanding in the team and come to actions that the team will do.

**Sail Boat Retrospective**

This retrospective exercise allows a team to think about its own objectives, impediments, risks, and good practices, in a simple piece of paper. The members can define a vision and identify what slows them down and what actually helps them to achieve their objectives.

You start a sail boat retrospective by drawing a boat, rocks, clouds, and couple of islands as shown in the picture.



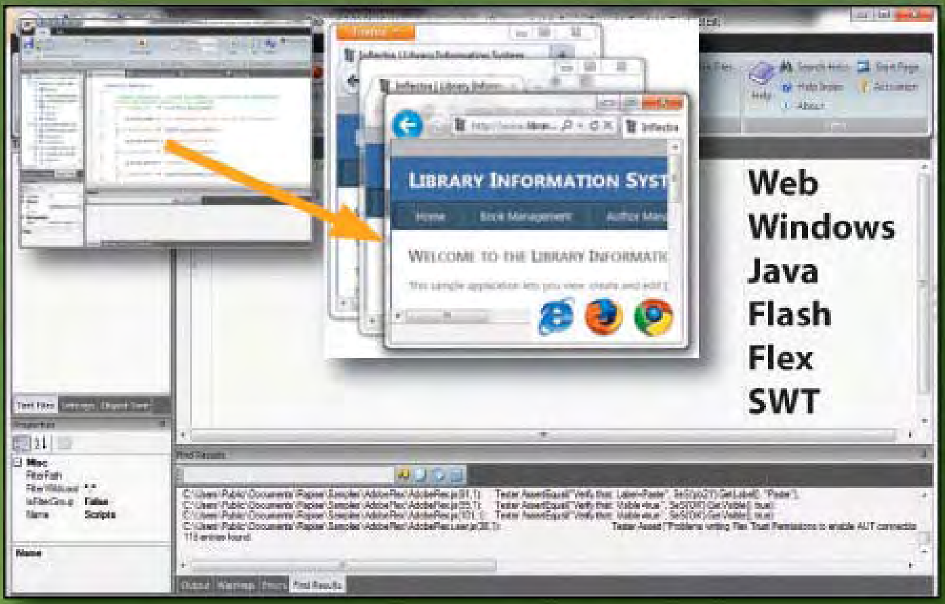
Sail Boat

The islands represent the team's goals/vision. They work every day to reach these islands. The rocks represent the risks they might encounter along the way. The anchor on the boat is everything that slows them down on their journey. The clouds and the wind represent everything that helps them to reach their goal.

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

Need to Test Your Application on Multiple Environments?

Writing Test Scripts Too Slow?



It's time to try a better way.

Rapise

RAPID & FLEXIBLE TEST AUTOMATION

Learn more at: inflectra.com/rapise

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

inflectra®

With the picture on the wall, write down the team visions or goals. Start a brainstorming session during which the team dumps their ideas into the different areas according to the picture. Give the team 10 minutes to write their ideas down on post-its. Afterwards, give each person five minutes to read their ideas out loud.

At this point, discuss with the team how can they continue to practice what is written on the clouds/wind area. These are good ideas that help the team and they need to continue with them. Next, discuss how the team can mitigate the identified risks.

Finally, let the team choose the most important issue that is slowing them down. If there is disagreement within the team about which topic to tackle, you can use vote dots. At the end, the team can define the steps to take to fix the problem and the retrospective can conclude.

## Strengths-Based Retrospective

How can you become an excellent team that is able to deliver and exceed customer expectations? By continuously becoming better in the things that you are great at. This can be accomplished using a strengths-based retrospective with a solution focused approach based on Solution Focused Therapy [4]. This kind of therapy does not focus on the past but instead focuses on the present and future. It examines what works in a given situation, and uses that to address existing problems. It is a positive way of improving, exploring possibilities and revealing strengths that people and teams may not be aware of.

In retrospectives, teams use an exercise to reflect on the work that they did, analyze what happened and why, and define improvement actions for the next iteration. These actions imply that they will change their way of working. A strength-based retrospective is a different approach. Instead of coming up with a list of actions to start doing new things (of which you might not be capable of doing), your actions result in doing more of the things that you are already doing and which you are good at.

A strength-based retrospective consists of 2 steps: discovering strengths, and then defining actions that use them. Both steps consist of retrospective questions that team members ask themselves.

Discovering strengths: Think of something that succeeded in this iteration that the team managed to accomplish beyond expectation, and which produced benefits for you, the team, and/or for your customers. Now ask yourself and your team the following kinds of questions:

- How did we do it?
- What did we do to make it successful?
- What helped us do it?
- Which expertise or skills made the difference?
- Which strengths that you possess made it possible?
- How did being part of a team help to realize it?
- What did team members do to help you?
- Which strengths does your team have?

The questions are based on Appreciative Inquiry [5], an approach that focuses on value and energy. These questions give visibility to good things that happened and explore the underlying strengths that made it possible.

Defining actions: Think of a problem that you had in the past iteration, one that is likely to happen again. For example a problem that is keeping you and your team from delivering benefits for your customers? Now ask:

- How can you use your individual strengths to solve this problem?
- What would you do more frequently that would help prevent the problem from happening again?
- Which actions can you take, that you are already capable of?

Again, this applies appreciative inquiry by envisioning what can be done using the previously discovered strengths and giving energy to the team members to carry it out.

**Starting with Retrospectives**

Just as implementing any other agile practice, adopting retrospectives is an organizational change where professionals adapt their way of working, their behavior. It won't just happen, and if not properly supported, it may take much time, or even fail. So when you start with retrospectives, make clear what their purpose is, and set up a team of capable retrospective facilitators. Then start doing them with your agile teams, and reflect on how they are going

I started by doing agile retrospectives in stealth mode. I didn't use the term retrospective but called it an evaluation. My reasoning for using retrospectives was to help my projects with frequent evaluations and improvement actions, thus reaping the benefits of the retrospective during the project.

Becoming agile is hard work and you may have to deal with resistance to change. Once you have become more agile, things will get easier. When you have developed an agile culture and mindset, things will start to fall into place and decisions on Do's and Don'ts do often come easier. Frequently reflecting on your agile journey helps you stay agile.

**Valuable Agile Retrospectives**

Agile retrospectives are a great way to continuously improve the way of working. We hope that this article helps you and your teams to conduct retrospectives effectively and efficiently to reflect upon your ways of working, and continuously improve them!

Whatever way you choose to do agile retrospectives, ensure that you keep on doing them. Even if things seem to be going well, there are always ways to improve! Getting actions out of a retrospective that are doable, and getting them done helps teams to become agile in an agile way.

Our book Getting Value out of Agile Retrospectives [1] and related articles like this one are the beginning of a journey. We are growing a small ecosystem to release more exercises in the future, How To´s, retrospectives´ advices and many other things. If you want to stay up to date, the best way is to subscribe to our Valuable Agile Retrospectives mailing list [6].

## References and Links

[1] Getting Value out of Agile Retrospectives - A Toolbox of Retrospective Exercises.
Book Pages: http://www.benlinders.com/getting-value-out-of-agile-retrospectives/ and
http://lmsgoncalves.com/getting-value-out-of-agile-retrospectives/.
Download InfoQ: http://www.infoq.com/minibooks/agile-retrospectives-value
Download Leanpub: http://leanpub.com/gettingvalueoutofagileretrospectives

[2] The four Retrospective Key Questions:
http://www.retrospectives.com/pages/RetrospectiveKeyQuestions.html

[3] Project Retrospectives: A Handbook for Team Reviews, by Norm Kerth:
http://www.dorsethouse.com/authors/kerth.html

[4] Solution Focused Therapy: http://en.wikipedia.org/wiki/Solution_focused_brief_therapy

[5] Appreciative Inquiry: http://en.wikipedia.org/wiki/Appreciative_inquiry

[6] Valuable Agile Retrospectives Mailing List: http://eepurl.com/Mem7H

[7] Blog Ben Linders: http://www.benlinders.com/

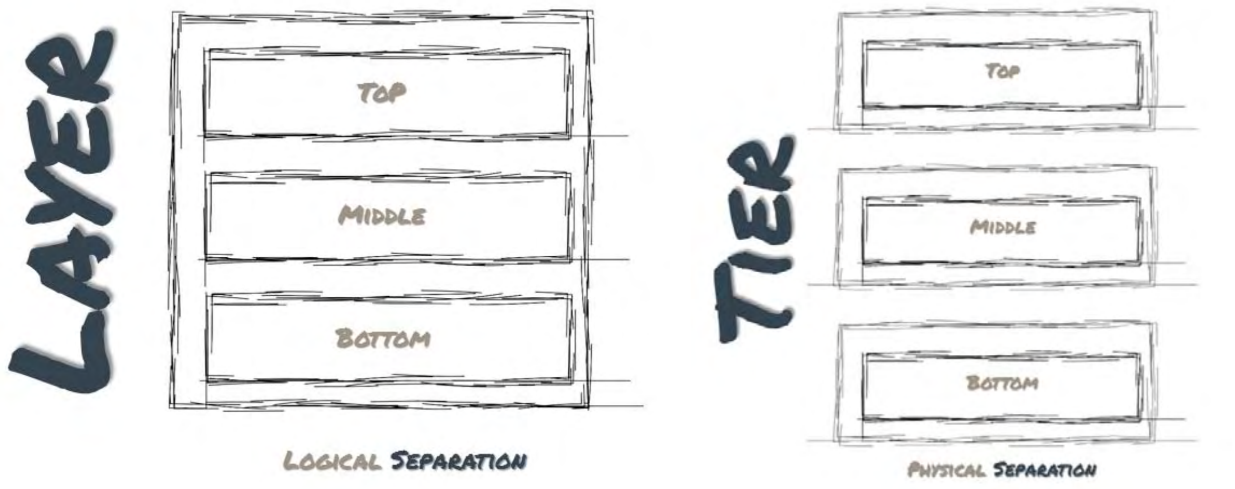
[8] Blog Luis Gonçalves: http://lmsgoncalves.com/

# Chop onions instead of layers

Daniel Marbach, @danielmarbach, http://www.planetgeek.ch

Chopping onions usually makes you cry. This is not the case in software architecture. On the contrary! The onion architecture, introduced by Jeffrey Palermo, puts the widely known layered architecture onto its head. Get to know the onion architecture and its merits with simple and practical examples. Combined with code structuring by feature your software is easy to understand, changeable and extendable. Turn your tears of sorrow into tears of delight. For a very long time the standard answer to the question how components and classes should be organized in the software architecture was layers. Before we explore the promised benefits of layers and how they represent themselves in software architecture, we need to get rid of a common misconception regarding layers vs. tiers.

## Layers vs. Tiers

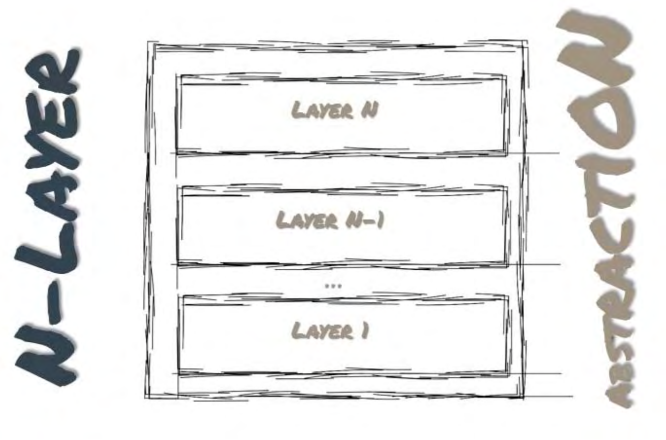


When we talk about layers, we mean the logical separation or division of components and functionality and not the physical location of components in different servers or places. The term "tiers" refers to the physical distribution of components and functionality in separate servers, including the network topology and remote locations. Tiers are usually used to refer to physical distribution patterns such as “2 Tier”, “3 Tier” and “N Tier”. Unfortunately, both layers and tiers often use similar names. In this article, we talk about layers and not tiers! Nevertheless, what is a layer besides a logical separation and when was it introduced?

In the year 1996 Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal analyzed different software systems. They asked themselves what patterns make software systems successful and allow us to evolve systems without developing a big ball of mud. Their knowledge was published in a book called Pattern-oriented Software Architecture – A System of Patterns. [12]

In that book they came to the conclusion that large systems need to be decomposed in order to keep structural sanity. The so-called Layer pattern should help to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction. The initial inspiration came from the OSI 7-layer Model defined by the International Standardization Organization. This inspired the original N-Layer model.

## N-Layer model



The layer higher in the hierarchy (Layer N+ 1) only uses services of a layer N. No further, direct dependencies are allowed between layers. Therefore, each individual layer shields all lower layers from directly being access by higher layers (information hiding). It is essential that within an individual layer all components work at the same level of abstraction. This approach is also called **strict layering**. The **relaxed** or **flexible layering** is less restrictive about the relationships between layers. Each layer may use the services of all layers below it. The advantage of this approach is usually more flexibility and performance (less mappings between layers) but this is paid for by a loss of maintainability.

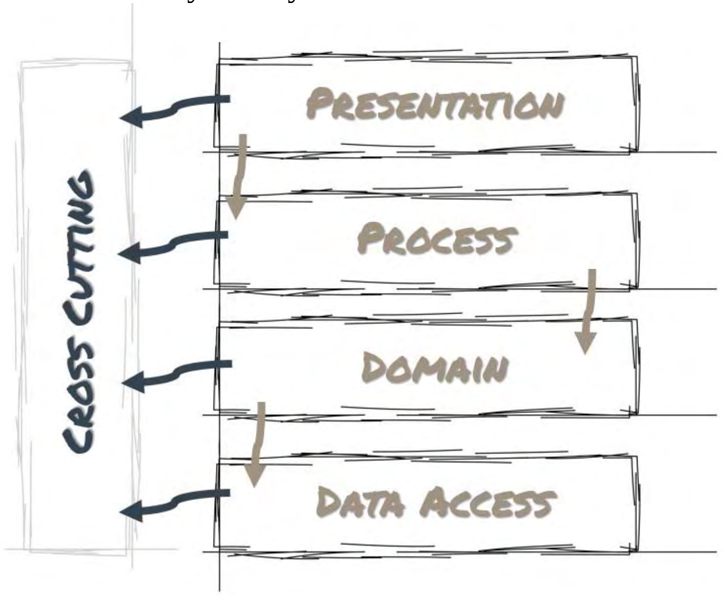
OnTime Scrum Project Management Tool - Click on ad to reach advertiser web site

OnTime Scrum
Agile project management & bug tracking software
The Scrum project management tool your development team will love to use.
Easily manage product backlogs
Automate your workflow process
Project visibility with burndowns
Get a Free 30-day Trial now.
Visit OnTimeNow.com
axosoft

In order to be able to define layers and put component into layers you have to define the abstraction criterion. For example, the lower levels can be defined by the distance from the hardware on the upper levels by the conceptual complexity. Possible layering could be chosen (top to bottom):

- User-visible elements
- Specific application modules
- Common services level
- Operating system interface level
- Operation System
- Hardware

The common layers they defined were



Other books or articles may name it differently but we will stick to that definition. We have the presentation or client layer, the process or service layer, the domain or business logic layer, the data access or infrastructure layer. Sometimes you see the layers above extended with another layer sitting on the left side spawning all layers. This layer is often called crosscutting layer which handles tracing, logging and more.

The advantages of this approach are:

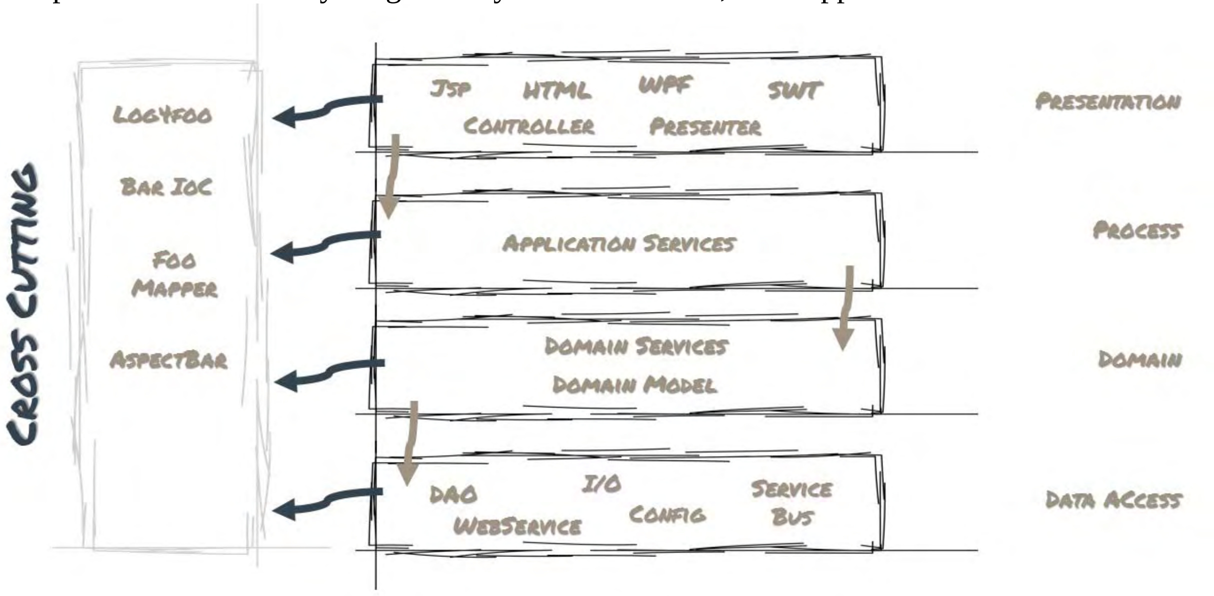
**Layer reuse**
If an individual layer embodies a well-defined abstraction and has a well-defined and documented interface, the layer can be reused in multiple contexts. Naturally the data access seems a nice fit for layer reuse.

**Supports standards**
Clearly defined and commonly accepted levels of abstraction enable the development of standardized tasks and interfaces. After many years of layered architecture a lot of tools and helpers have been invented to automatically map from one layer to another for example.

**Local dependencies**
Standardized interfaces between layers usually confine the effect of code changes to the layer that is changed.

**Layer exchange**
Individual layer implementations can be replaced by semantically equivalent implementations without too great of an effort.

Upon first sight the layer model seems straightforward and easy to adopt. Unfortunately developers often take the layering literally. Sometime later, this happens…



Instead of getting the best out of the benefits of the layered architecture style, we end up with several layers dependent on the layers below it. For example giving the previous layering structure the presentation layer depends on the application layer and then on the domain layer and finally on the database layer. This means that each layer is coupled to the layers below it and often those layers end up being coupled to various infrastructure concerns. It is clear that coupling is necessary in order for an application to be able to do anything meaningful but this architecture pattern creates unnecessary coupling.

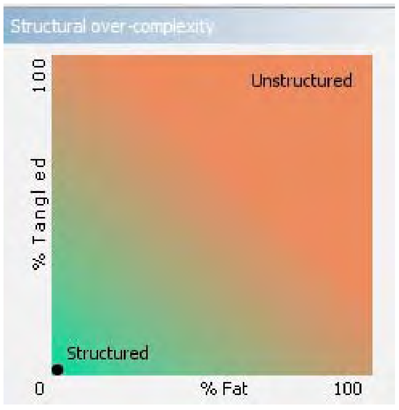
The biggest offender is the coupling of the UI and business logic to the data access. Wait a moment. Did I just say that the UI is coupled to the data access? Yes indeed. Transitive dependencies are still dependencies. No matter how anyone else tries to formulate it. The UI cannot function if the business logic is not available. The business logic in return cannot function if the data access is not available. We gracefully ignore the infrastructure because typically it varies from system to system. When we analyze the architecture above in retrospective, we detect that the database layer is becoming the core foundation of the whole application structure. It is becoming the critical layer. Any change on the data access / infrastructure layer will affect all other layer of the application and therefore changes ripple through from the bottom to the top of the application.

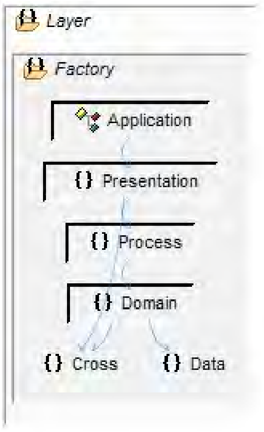
This architecture pattern is heavily leaning on the infrastructure. The business code fills in the gaps left by the infrastructural bits and pieces. If a process or domain layer couples itself with infrastructure concerns, it is doing too much and becomes difficult to test. Especially this layer should know close to nothing about infrastructure. Infrastructure is only a plumbing support to the business layer, not the other way around. Development efforts should start from designing the domain-code and not the data-access, the necessary plumbing should be an implementation detail.

Agile 2014 Conference - Click on ad to reach advertiser web site

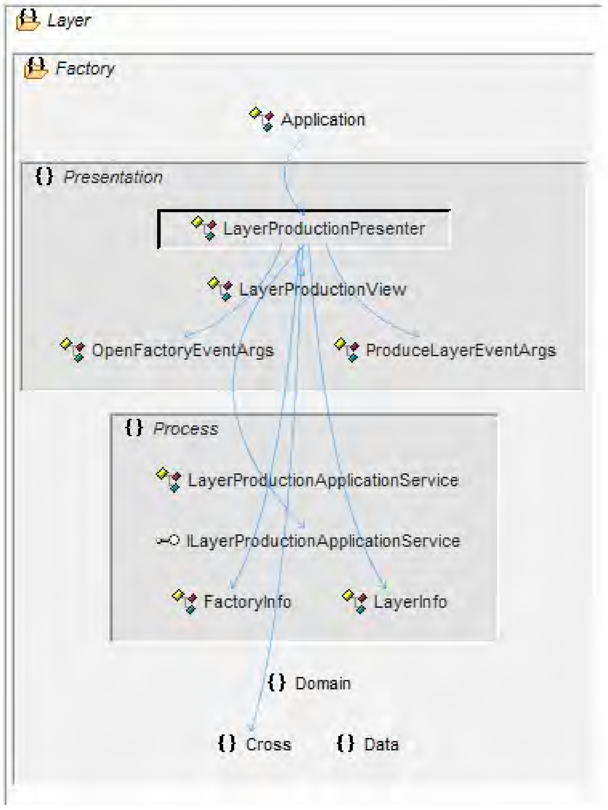
## Layer.Factory sample

To illustrate the layer architecture, we will analyze a Layer.Factory code sample that is available on github [13]. The Layer.Factory sample is a very simple Domain Driven Design sample application which follows the layered architecture pattern. The idea is that the domain model behind it represents a factory which produces layers (what a coincidence). In order to be able to create layers a factory responsible for creating layers must be created first. In order to analyze our sample application, we use a tool called Structure101 Studio. Structure101 Studio is a commercial tool which helps to visualize and organize large code bases in various programming languages. When we analyze the sample application with Structure101 Studio we see that the application is well structured. It contains no namespace or class tangles and no fat namespaces or classes. From a structural complexity perspective our application is in good shape. The dependencies are only going down from layer to layer. Therefore the sample adheres the strict layering principles.



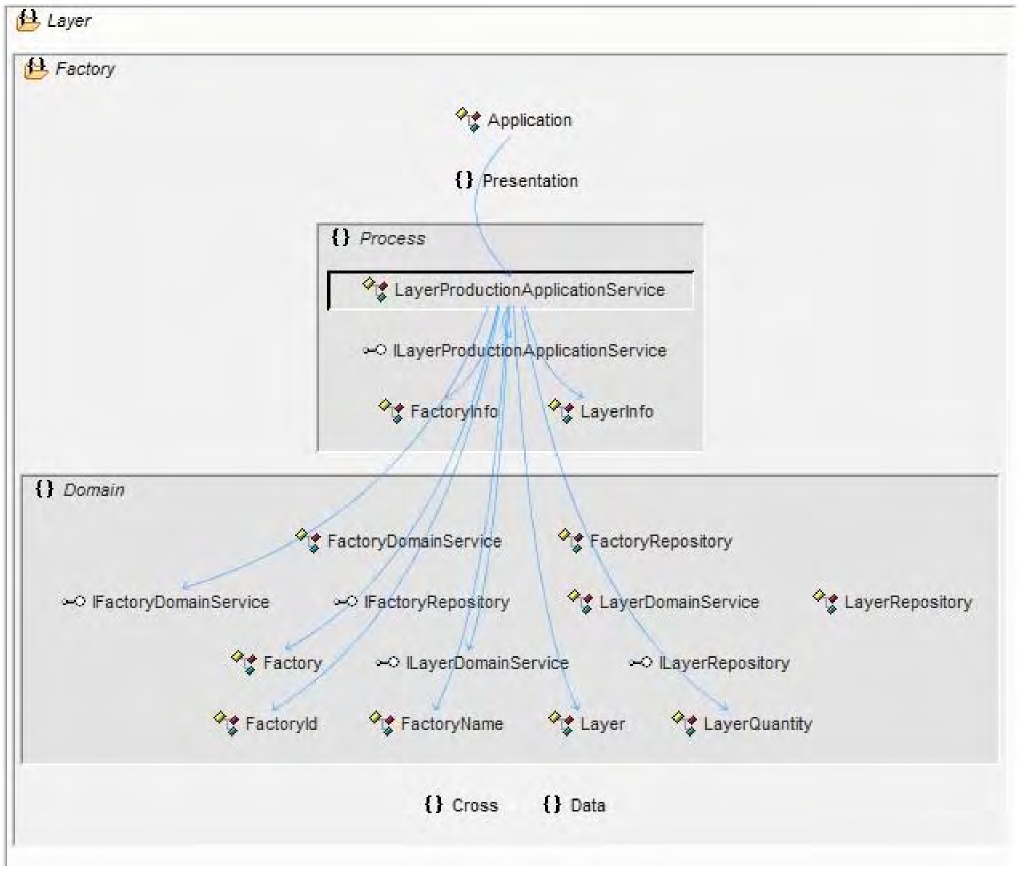


In order to see how the application structures itself internally we need to drill down deeper.



The presentation layer entry point is the *LayerProductionPresenter*. The *LayerProductionPresenter* uses the *ILayerProductionApplicationService* to open a factory and produces layers by using the previously opened factory. Because the application follows the strict layering pattern, the process layer has to translate domain objects into data transfer objects residing in the process layer (*FactoryInfo* and *LayerInfo*). The presentation layer can only use these data transfer objects to present information on the views. Data held by the domain objects has to be translated from layer to layer.

Drilling down deeper into the domain layer makes this issue more apparent. The *LayerProductionApplicationService* uses a set of Domain Services. These Domain Services implement the core business logic of the application and directly expose the domain model's aggregates, entities and value objects (i.e. *Factory*, *FactoryId*, *FactoryName*, *Layer*, *LayerQuantity*).

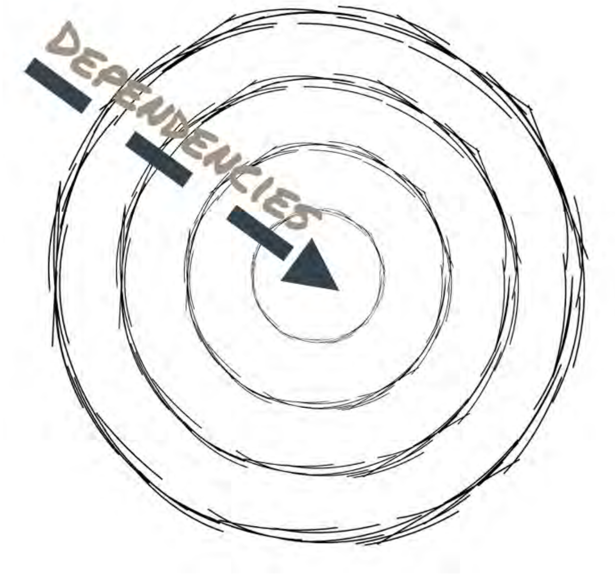


As we can see, there is a bunch of translation from top to bottom and from bottom to top going on. Input information floating down from the presentation to the domain layer has to be translated from the presentation data transfer objects to the process data transfer objects and ultimately to the domain objects. Output information going up from the domain to the process layer has to be translated from the domain objects to the process data transfer objects and ultimately to the presentation data transfer objects. As long as only data is transferred the mapping process is tedious but manageable. As soon as the presentation layer would like to reuse business rules from the core domain model this approach's drawbacks outweigh its benefits.

How do we get around the drawbacks of the layered architecture? With onions! What else?

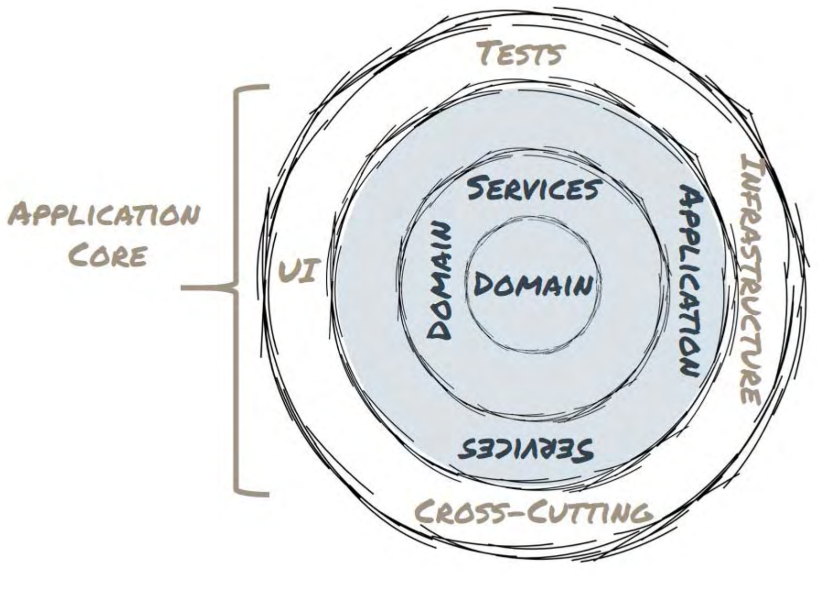
## Onion Architecture

We simply move all infrastructure and data access concerns to the external of the application and not into the center. Jeffrey Palermo proposed this approach called Onion Architecture on his blog 2008. The approach is nothing new. However, Jeffrey liked to have an easy to remember name, which allows communicating the architecture pattern more effectively. Similar approaches have been mentioned in Ports & Adapters (Cockburn), Screaming Architecture (Robert C. Martin), DCI (Data Context Interaction) from James Coplien, and Trygve Reenskaug and BCE (A Use Case Driven Approach) by Ivar Jacobson. Let us depict the onion architecture.



The main premise is that it controls coupling. The fundamental rule is that all code can depend on layers more central, but code cannot depend on layers further out from the core. In other words, all coupling is toward the center. This architecture is unashamedly biased toward object-oriented programming, and it puts objects before all others.

Furthermore the Onion Architecture is based on the principles of Domain Driven Design. Applying those principles makes only sense if the application has a certain size and complexity. Be sure to reflect properly on that point before jumping blindly into the Onion Architecture. Let us see what Onions combined with Domain Driven Design produces.



In the very center, we see the Domain Model, which represents the state and behavior combination that models truth for the organization (everything unique to the business: Domain model, validation rules, business workflows). The number of layers in the application core will vary, but remember that the Domain Model is the very center, and since all coupling is toward the center, the Domain Model is only coupled to itself.

The first ring around the Domain Model is typically where we would find interfaces that provide object saving and retrieving behavior, called repository interfaces. The object saving behavior is not in the application core, however, because it typically involves a database. Only the interface is in the application core. Out on the edges we see UI, Infrastructure, and Tests. The outer rings are reserved for things that change often. This approach to application architecture ensures that the application core doesn't have to change as: the UI changes, data access changes, web service and messaging infrastructure changes, I/O techniques change.

The Onion Architecture relies heavily on the **Dependency Inversion principle**. The application core needs implementation of core interfaces, and if those implementing classes reside at the edges of the application, we need some mechanism for injecting that code at runtime so the application can do something useful. So tools like Guice, Ninject etc. are very helpful for those kinds of architectures but not a necessity.

The application is built around an independent object model. The whole application core is independent because it cannot reference any external libraries and therefore has no technology specific code. The inner rings define interfaces. These interfaces should be focusing on the business meaning of that interface and not on the technical aspects. So the shape of the interface is directly related to the scenario it is used in the business logic. The core takes ownership of these interfaces. Outer rings implement interfaces, meaning all technology related code remains in the outer rings. The outermost ring can reference external libraries to provide implementations because it contains only technology specific code. This allows pushing the complexity of the infrastructure (which has nothing to do with the business logic) as far outwards as possible and therefore, the direction of coupling is toward the center.

That approach makes us independent of several infrastructure and crosscutting concerns:

- **Database**: The business rules do not depend upon the database (so storage can be swapped out)
- **UI**: The UI can change without changing the rest of the system
- **Frameworks**: The architecture does not depend on the existence of some library. This allows you to use frameworks as tools rather than having to cram your system into their limited constraints
- **External agency**: Business rules do not know anything about the outside world.

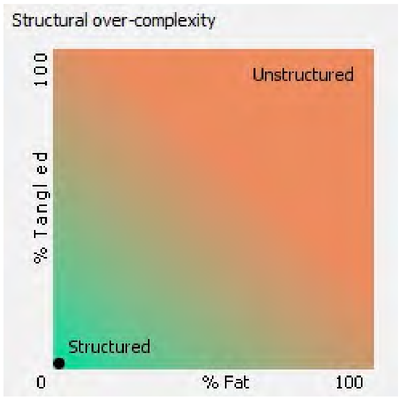
Which leads us to the ultimate benefit of this architecture. The application core is 100% testable.

**Onion.Factory sample**

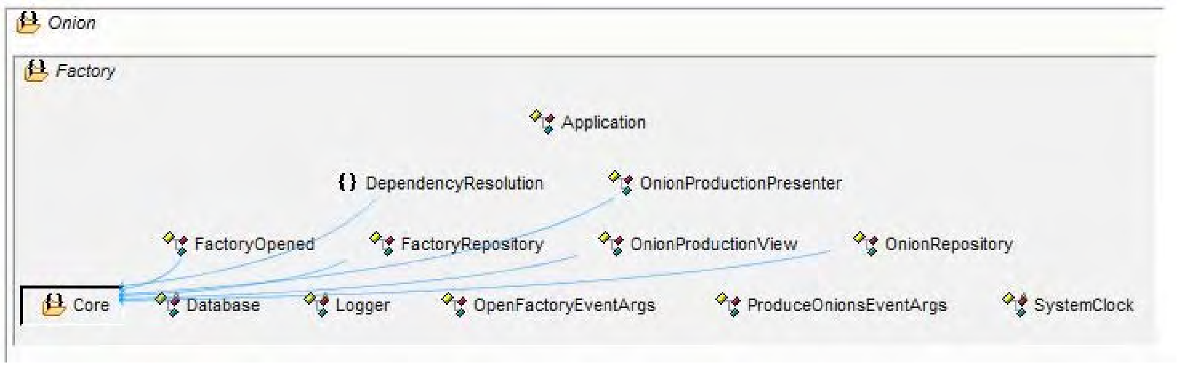
To illustrate the onion architecture, we will analyze an Onion.Factory code sample that is available on github [14]

The Onion.Factory sample is a very simple Domain Driven Design application which follows the onion architecture pattern. The idea is that the domain model behind it represents a factory which produces onions (what a coincidence). In order to be able to create an onion, a factory responsible for creating onions must be created first.

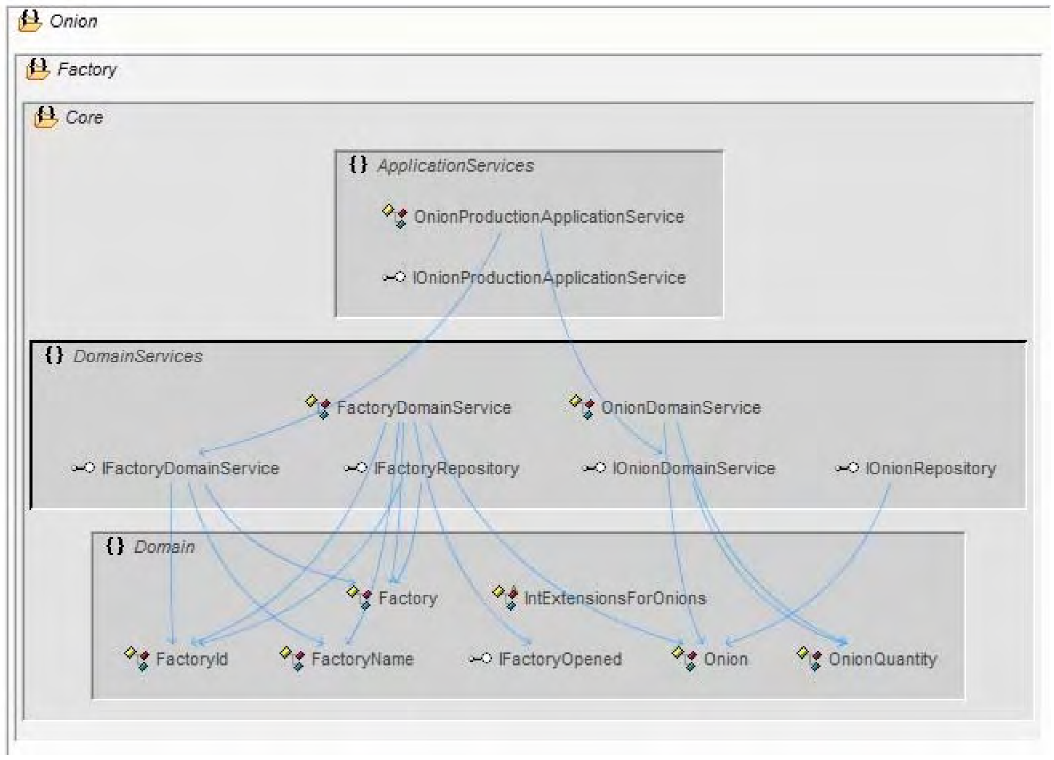
When we analyze the sample application with Structure101 Studio we see that the application is well structured. It contains no namespace or class tangles and no fat namespaces or classes.



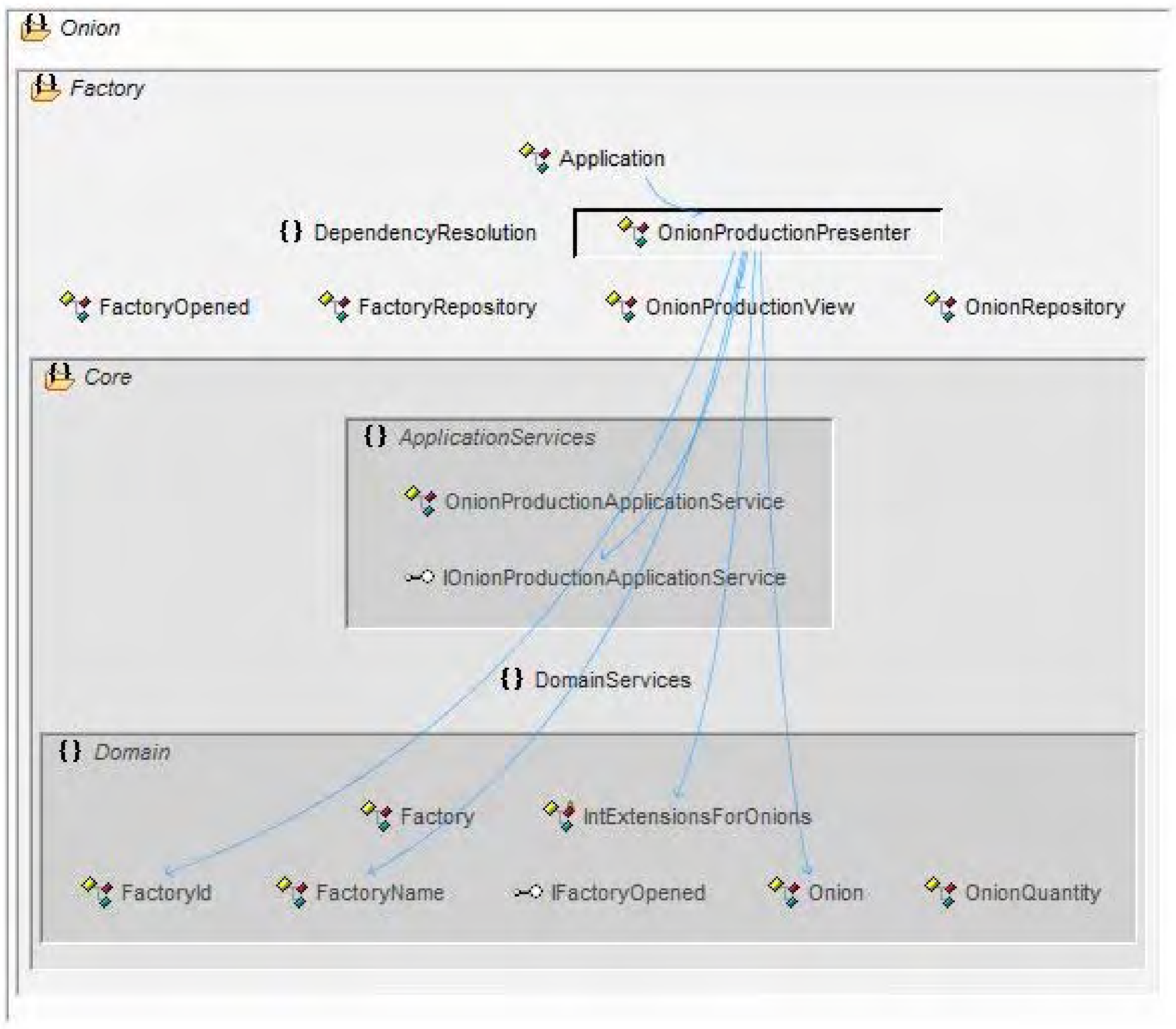
All classes which have dependencies to concrete things like the database, the file system, the view and more reside in the outer rings. All dependencies are floating towards the core. No dependency points from the core into the outer rings.



In order to see how the application structures itself internally we need to drill into the core. The core is shown isolated without the outer rings.



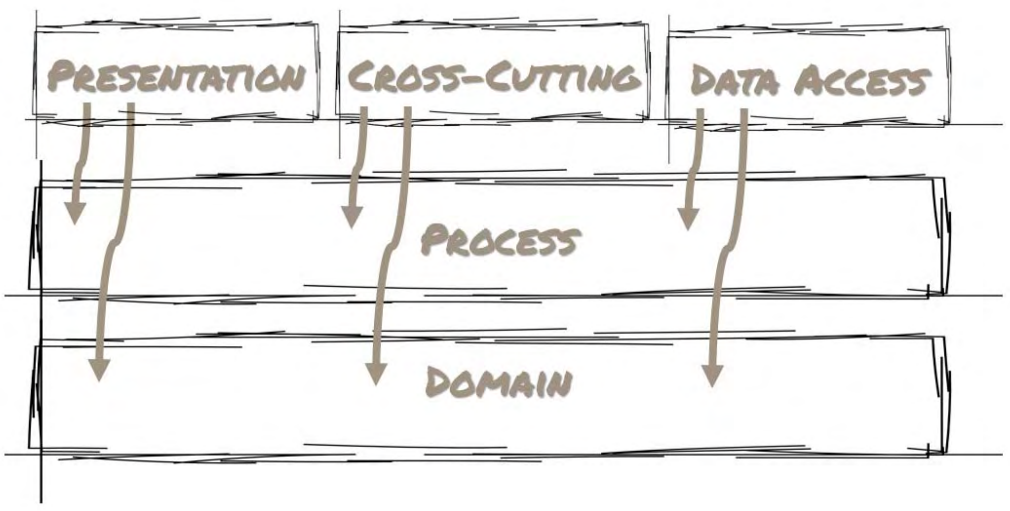
The core itself is pure domain driven design and contains only what is near and dear to the heart of the business: the business domain and rules! When we reflect again about the layered architecture we remember that we needed to map from data transfer object to data transfer object over all layers. How does the presenter and its dependencies look like with the onion architecture?



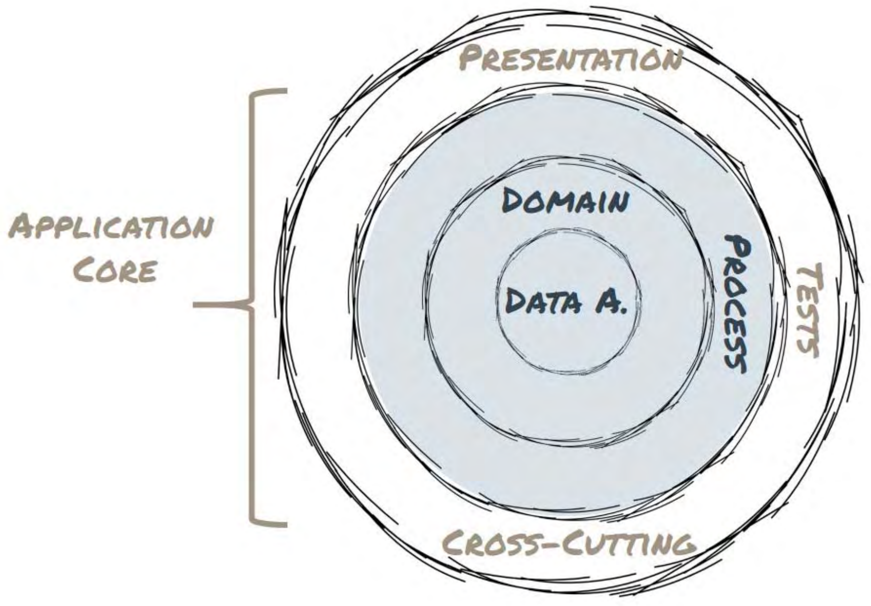
The *OnionProductionPresenter* uses the *IOnionProductionApplicationService*. This is exactly the same dependency we've previously seen in the layered architecture. The real difference becomes apparent in the area where the presenter directly uses the domain model. The presenter uses and display information available on the domain model and can reuse business rules implemented in the domain model behind aggregates, entities and value objects. We don't need to reinvent the wheel!

**Layers vs. Onions**

If we apply the principles of the Onion Architecture to the layered architecture, we need to turn the layer diagram upside down.



The key difference is that the Data Access, the presentation and the cross-cutting layer along with anything I/O related is at the top of the diagram and not at the bottom. Another key difference is that the layers above can use any layer beneath them, not just the layer immediately beneath. At least this approach could be achieved by using relaxed layering.



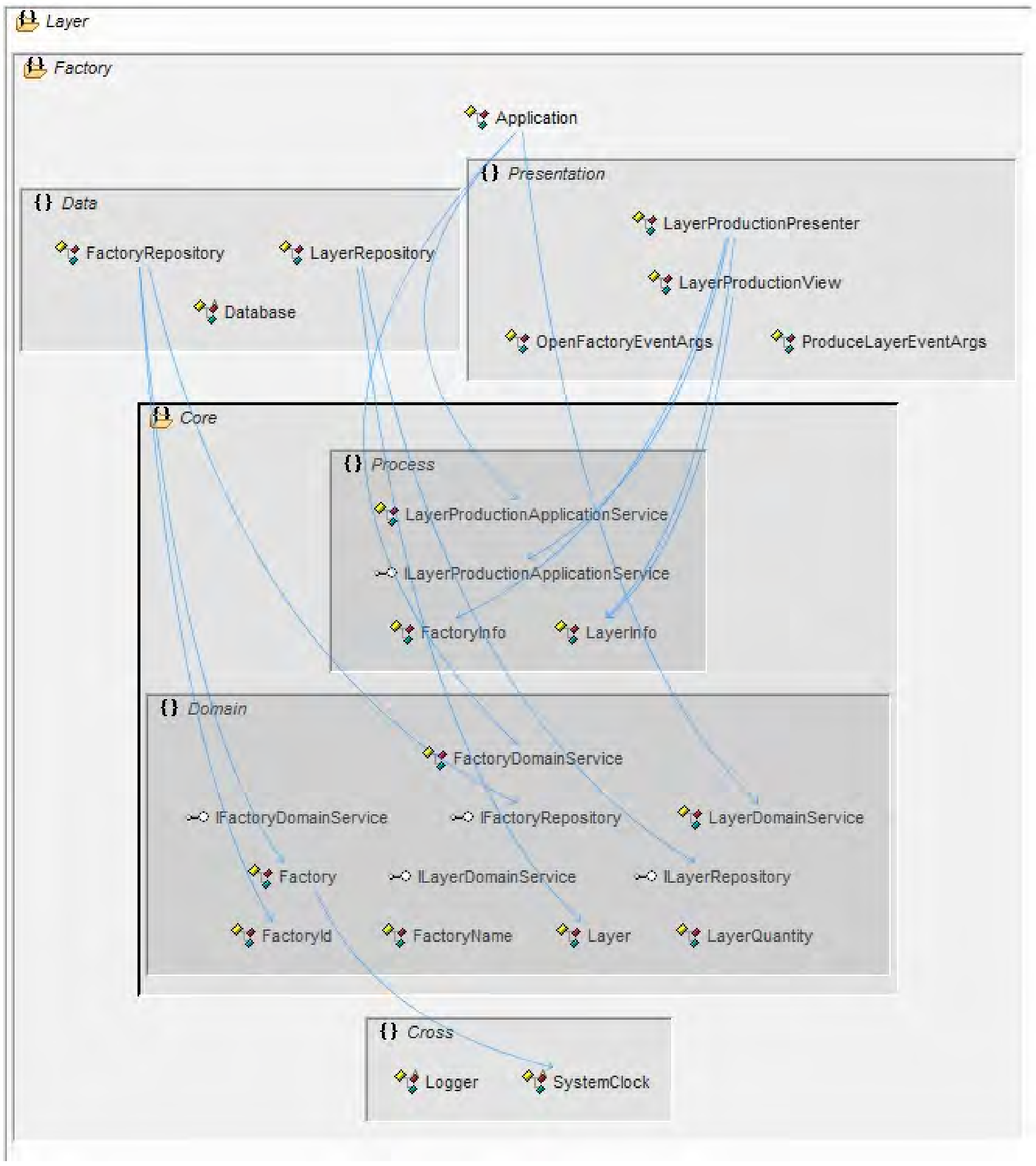
If we put the traditional layered architecture in concentric circles we clearly see the application is built around data access and other infrastructure. Because the application has this coupling, when data access, web services, etc. change, the business logic layer will have to change. The world view difference is how to handle infrastructure. Traditional layered architecture couples directly to it. Onion Architecture pushes it off to the side and defines abstractions (interfaces) to depend on. Then the infrastructure code also depends on these abstractions (interfaces). Depending on abstractions is an old principle, but the Onion Architecture puts that concepts right up front.

## Refactoring from Layers towards Onions

Now that we know the difference between layers and onions, let us try to refactor the Layer.Factory sample towards an onion architecture. The first refactoring approach we'll take will try to leave as many of the original namespaces intact, remember they are used in the sample to simulate layers. After applying the following steps

- Introduce *Core* namespace
- Move *Process* and *Domain* namespace into *Core*
- Move *FactoryRepository* and *LayerRepository* into *Data* namespace

The code structures itself like the following:



We can see in that simple example that refactoring from a layered architecture to an onion architecture is not enough in order to get a true onion architecture. We actually need to redesign the software. The newly introduced *Core* still has a dependency outgoing. In order to get rid of this dependency we would need to introduce a small abstraction (interface) defined by the needs of the *Core*.

The introduced interface needs then to be implemented by the outer rings (in the sample above by the *SystemClock*. Furthermore we need to get rid of the unnecessary abstractions introduced by the layered architecture in example by removing the unwanted data transfer objects.

## Summary

Let us summarize what we learned. As easy as it may sound in the beginning, strictly following the layered architecture approach can lead to a big dependency tangle. Using the onion architecture approach makes you think about getting the dependencies right from the start. Combined with the introduction of necessary abstractions you achieve an independent application core which is fully testable. This allows you to maintain, protect and evolve what matters the most to your business: your domain logic! Chop onions instead of layers and turn your tears of sorrow into tears of delight.

## References

1. Alistair Cockburn Hexagonal Architecture http://alistair.cockburn.us/Hexagonal+architecture
2. The Clean Architecture Uncle Bob http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html
3. Screaming architecture Uncle Bob http://blog.8thlight.com/uncle-bob/2011/09/30/Screaming-Architecture.html
4. Growing Object Oriented Software Guided by Tests, Steve Freeman & Nat Pryce, Addison-Wesley - Designing for Maintainability Page 47-49
5. Ports and Adapters With No Domain Model http://www.natpryce.com/articles/000786.html
6. Improve Your Software Architecture with Ports and Adapters http://spin.atomicobject.com/2013/02/23/ports-adapters-software-architecture/
7. Implementing Domain Driven Design by Vaughn Vernon published by Addison-Wesley Professional - Chapter 4 Hexagonal or Ports and Adapters
8. The Onion Architecture : part 1 to part 4 http://jeffreypalermo.com/blog/the-onion-architecture-part-1/
9. Creating N-Tier Applications in C# by Pluralsight http://pluralsight.com/training/courses/TableOfContents?courseName=n-tier-apps-part1 http://pluralsight.com/training/courses/TableOfContents?courseName=n-tier-apps-part2
10. The Onion Architecture by Matt Hidinger http://matthidinger.com
11. Software development fundamentals part 2 Layered architecture by Hendry Luk http://hendryluk.wordpress.com/2009/08/17/software-development-fundamentals-part-2-layered-architecture/
12. Pattern-oriented Software Architecture – A System of Patterns Volume 1 by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal published by Wiley Series
13. The Layer.Factory example https://github.com/danielmarbach/Layer.Factory
14. The Onion.Factory example https://github.com/danielmarbach/Onion.Factory

# Ranorex - Automated Testing Tool for Desktop, Web & Mobile Applications

Franco Martinig, Methods & Tools

Ranorex is a software testing tool that allows you perform functional testing on desktop, web or mobile applications. It supports many user interface (UI) technologies that includes Java, HTML, C#, Flex/Flash, Android, iOS and Silverlight.

**Web Site:** http://www.ranorex.com/
**Version tested:** Ranorex Studio 4.1.5.17134 in February-March 2014 on Windows 8
**System requirements:** Windows XP, 7, 8, Windows Server 2003-2012
Microsoft .NET Framework 3.5 SP1 or higher installed for running Ranorex Studio
**License & Pricing:** from 690 to 3.490 euros depending on the license type.
More information on http://www.ranorex.com/purchase/buy-now.html
**Support:** support requests, on-line forums, on-line documentation.
More information on http://www.ranorex.com/support/support-center.html

## Installation

After downloading the setup program, the process uses a classical windows install program and goes smoothly. Several Ranorex icons are added to your Windows pane.

- Ranorex Studio
- Ranorex Spy
- Ranorex Test Suite Runner
- Ranorex Instrumentation

Ranorex Studio is available on two different editions. The runtime only edition allows standalone test execution and the premium edition provides all features with node locked or floating licenses. The node locked license type is bound to the machine's host name. A floating user license can be shared between different machines.

## Documentation

The complete documentation for the Ranorex suite is available on-line and consists of a user guide and screencasts. There is also a PDF version of the user guide that you can download on your computer. The open source tool KeePass is provided with Ranorex as an example application that is referenced in the documentation on how to use Ranorex.

## Configuration

The global settings window of Ranorex is accessible from the different tools. These windows allow managing different parameters for the recording and replaying activity like the delays between each action or the number of snapshots captured.

## Features

The Ranorex Studio is composed of different tools that provide their own features. All tools are integrated in the Ranorex Studio framework. The Ranorex Studio tools allow you to test applications running the following platforms:

- Winforms, C#, VB.NET
- Web (IExplorer, Firefox, Chrome, Safari)

- WPF, Silverlight, Qt
- SAP, Oracle Forms, MS Dynamics
- Flash/Flex
- Java
- Android
- iOS

In addition to the technologies Ranorex also supports:
- Capture & Replay
- Keyword-Driven Testing
- Data-Driven Testing

To support the technologies of the applications under test, Ranorex uses an instrumentation system which is similar to the plugin system used by other software. This provides specific features for the technology tested. There is for instance an extension to install if you want to test a web application in Firefox or some files to include in the Java Swing and Java AWT applications if you want to test Java Swing and Java AWT applications.

Ranorex allows you to manage your test activity in a concept called solution where you can manage your testing projects. The project contains testing suites that are composed of testing scripts and code modules. A test suite allows creating new test cases by combining existing automation modules. You can also specify global parameters and data connectors to set up data-driven test cases.

The testing scripts can be created with Ranorex Recorder, but you can also write your testing code independently for more complex needs. This will allow you for instance to have test code using variables that allow running the same code for different test cases. Ranorex testing code can be managed with Subversion and Team Foundation Server.

Ranorex tests can also be data driven, using a simple internal data table or getting data from external sources like Excel files, CSV files or SQL Databases.

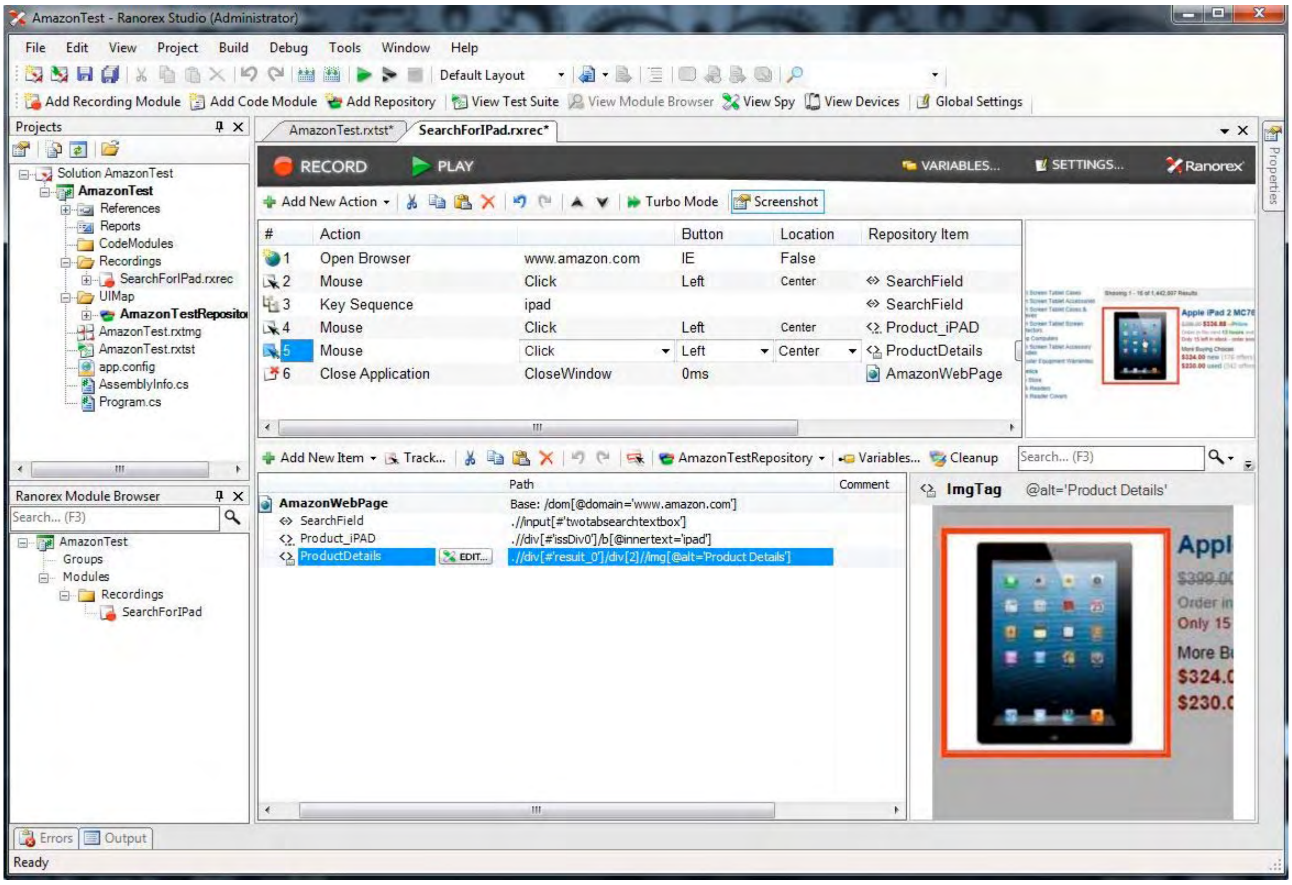
One of the core strength of the Ranorex tools is its repository. This tool provides a mapping between the user interface elements of the tested application and the testing framework. This reduces the efforts to maintain your tests as the items to manipulate in test are identified directly and not just by their placement on the user interface. In complex systems, the elements of the interface can be organized in folders, grouping for instance all the buttons. The repositories are stored in xml files that can be edited and managed like any other source code. A repository can be automatically generated during a recording session or manually populated. Ranorex Spy can also be used for this task. Multiple repositories can be used if many software testers are working on the same project.

## Ranorex Studio

Ranorex Studio acts as the central tools for the Ranorex test suite. It is the tool that allows to manage all the testing items like the projects or tests suites. You can also start the other tools like the Spy and the Recorder that are totally integrated in the Studio. The Studio has three views

- The project view (top left) shows all files and references currently associated with the project.

- The 'Module Browser' view (bottom left) lists all available modules based on the project's code files and module groups based on the projects module group file.
- The file view (large window right) shows the content of the selected item from the two other windows. You can create or adapt a recording module, write code modules or view a test report. In the example below, the upper part shows the test actions and the lower part the repository items for a test on the Amazon.com web site.



| # | Action | | Button | Location | Repository Item |
|---|---|---|---|---|---|
| 1 | Open Browser | www.amazon.com | IE | False | |
| 2 | Mouse | Click | Left | Center | SearchField |
| 3 | Key Sequence | ipad | | | SearchField |
| 4 | Mouse | Click | Left | Center | Product_iPAD |
| 5 | Mouse | Click | Left | Center | ProductDetails |
| 6 | Close Application | CloseWindow | 0ms | | AmazonWebPage |

| | Path | Comment |
|---|---|---|
| AmazonWebPage | Base: /dom[@domain='www.amazon.com'] | |
| SearchField | .//input[#'twotabsearchtextbox'] | |
| Product_iPAD | .//div[#'issDiv0']/b[@innertext='ipad'] | |
| ProductDetails | .//div[#'result_0']/div[2]//img[@alt='Product Details'] | |

Figure 1. Ranorex Studio

## Ranorex Recorder

The Ranorex Recorder is a tool used to record and replay the keyboard and mouse actions during a manual test of the user interface. You can start the recorder independently or from the Studio interface. The recorder operates from a small window with controls that you can activate with your mouse or with shortcuts. During the recording you can create checkpoint with the validate action. These validations can be done with the status of the elements of the user interface or you can validate images or screen regions of an application.

Compared to other Capture & Replay approaches, Ranorex recorder automatically separates UI actions – including the script behind – from the object identification layer. UI elements in Ranorex repositories are automatically created during recording or reused in case of existence. Based on the RanoreXPath object identification mechanism used in the repositories, recordings become a lot more robust against UI changes in the system under test.

## Ranorex Spy

The Ranorex Spy provides the tool needed to explore and analyze your code and user interface whether it is a Windows, mobile or web application. You can for instance examine the different elements of a web page that you are testing and take snapshots. The Spy feature can be directly called from the Studio screen. Ranorex supports a large list of user interface technologies that are listed on http://www.ranorex.com/automate-testing-of-desktop-web-mobile-software.html

## Ranorex Test Suite Runner

The Ranorex Test Suite Runner allows you to run the test suite independently from Ranorex Studio that can be done with only a runtime floating license. Each run of a test suite produces a test report. Tests can also be run from the command line if you want to integrate them with other tools like test management tools (HP Quality Center, etc.) or continuous integration servers.(TeamCity, Jenkins, etc.).

## Conclusion

The Ranorex Studio is a powerful software testing tool that supports a large number of applications technologies on the Windows, Web and mobile platforms. One of its core strength lies in its user interface items repository that separates the test logic from the user interface display, thus allowing to keep the same tests even when the interface changes. If the overall user interface and the usage main features are very intuitive, this tool requires a certain amount of time to benefit from its full power for your specific application development technology. This learning curve is however made easier by the support provided by the documentation, screencasts and user forums.

A 30 days free trial version of Ranorex can be downloaded from
http://www.ranorex.com/download.html

**Better Software & Agile Development Conference West | June 1–6, 2014 | Las Vegas, NV, USA**

Attend the premier collocated conferences for software professionals, and learn about the most up-to-date information, tools, and trends regarding agile, project management, people and teams, software testing and QA, software requirements, process improvement, metrics, design, and many more. Join industry experts and peers for a week jam-packed with learning sessions that will help you make a powerful impact in your job role and for your company.

Enjoy an exclusive discount for up to $200 off when you use the code MTCW.  Plus, if you register before May 2, you can take advantage of Early Bird pricing and save another $200 more! Go to http://vlt.me/previewmt32314

---

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 a free software development magazine 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