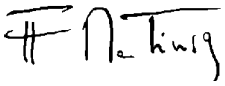
METHODS & TOOLS

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

Summer 2013 (Volume 21 - number 2) www.methodsandtools.com

Writing More Than Code in Software Development

The recent trend in software development has been to produce less documentation and written material. Many developers have interpreted the Agile Manifesto preference of "working software over comprehensive documentation" as the authorization to stop producing any document during a project. This has been translated in sentences like "the code is the documentation" or "the requirements are in the application". There have been certainly many projects that have produced documentation without purpose. You could have heard the famous tale about the 200 pages requirements document that contained a sentence like "the first one who would report reading this section will receive 100 dollars". Remember however that the Agile Manifesto says " while there is value in the items on the right, we value the items on the left more." There are many aspects of software development that will benefit from writing more than code and there is certainly value in doing this. If I didn't believe that written material had some value for software developers, Methods & Tools would contain only code listings. For me the first benefit of writing is to validate if things are clear in your mind. If you think you understand something, having difficulty to write it is a sign that maybe you should try to think more about it. The benefits are not in the results but in the process. It is similar when you try to define acceptance criteria for a requirement. If it is difficult to write some functional tests for it, then your requirement is maybe not so clear. There is also some specification of the software that cannot be found in the code. If the first screen of your web site or mobile application has to be loaded below a certain amount of time, you need a place to keep this information. If your screens have to respect specific accessibility standards, you also need a document to register this information. Agile recommends sometimes to use "soft" material like index cards or post-its to manage requirements information. However, good applications have a life span that is longer than their development time... and often their development team. I don't think that organizations will want to keep forever meeting room walls full of index cards. You need a support that can be available for a long period and that has search capabilities. Over the time, there will always be people asking questions about the features of the software. Having the original written requirements will allow you to answer if the application is working as specified or if you have introduced a bug in it. Yes, bugs still exist, even if you try to follow an Agile software development approach ;o) So my advice is that you should keep writing down requirements and other useful information about your application: do it for you and for your colleagues.



Inside

Testing Performance of Mobile Apps - Part 3: The Network

By Newman Yang and Phil Hamilton
newman.yang [at] xbosoft.com, philip.hamilton [at] xbosoft.com
XBOSoft, www.xbosoft.com

Today more and more users access the web via mobile devices. What is the difference in accessing a web app or web site from a mobile platform versus a desktop? From a desktop, if more users visit a www.mycompany.com web site than it was designed for, there may be slow server response or the server may crash. But what about mobile access? One report from Fonolo [1] showed that 63% of online visitors would be less likely to do business with a company via *any* channel after having problems with transactions on a mobile device, so performance for mobile applications seems to have a significant impact not only on that particular transaction but also on the user's overall impression or experience and remembrance of that company and its products or services.

When executing performance tests on a desktop, web, or client server platform, we commonly use tools such as LoadRunner, NeoLoad, LoadUI or Webload to emulate virtual users and simulate activities on the target web site or web app. These tools enable us to analyze each step or steps and potential scenarios where many users are executing different percentages of different activity sequences.

One common tool commonly used is Speedtest.net. This tool tests the user's current network situation. With a native mobile application for both Android and iOS, Speedtest.net executes tests from a local server depending on the location of your device. Figures 1 to 3 show some typical screenshots and results when using the tool.

Figure 1. SPEEDTEST.NET - Click Begin Test

Figure 2. SPEEDTEST.NET - 3 test measurements including Ping time, download speed and upload speed.

Figure 3. SPEEDTEST.NET - Results for each test cycle for comparison.

Examining the measurements provided by SPEEDTEST:

- **Ping time**: This indicates the initial time to send a message and hear back from the application server. In some instances we call this first look or the initial reaction time of the network for the first packets to come back to the device through the network.
- **Download speed**: This is the speed of downloading data, so especially for browsing or information related mobile applications such as news or video, this measurement can significantly affect the user experience of performance.
- **Upload speed**: This is the speed of uploading data, so depending on the characteristics of the mobile application, this measurement can significantly affect the user experience of performance or not at all.

When carrying out network performance tests, as usual it is important to have a controlled environment and systematic parameter alteration. Some of the parameters in the test environment for instance include: mobile device, mobile application (native or mobile web site), type of interaction, time of day, etc.

We used SPEEDTEST to carry out some simple network performance tests and get a comparison baseline. Firstly, we wanted to get some 3G network performance characteristics with one device, the HTC One (Android 4.1). We ran the test six times. Figure 4 shows the results.

Test cycle	ping time (10ms)	download speed (kbps)	upload speed (kbps)
1	41.1	26	13
2	70.3	61	151
3	17.700	32	48.000
4	25.6	33	100
5	23.1	98	66
6	44.1	29	22
StdDev	19.300	28	52.000
Avg	37	47	67
CV	0.522	0.596	0.776

Figure 4: Network -3G, Device: HTC One (Android 4.1), 6 test cycles

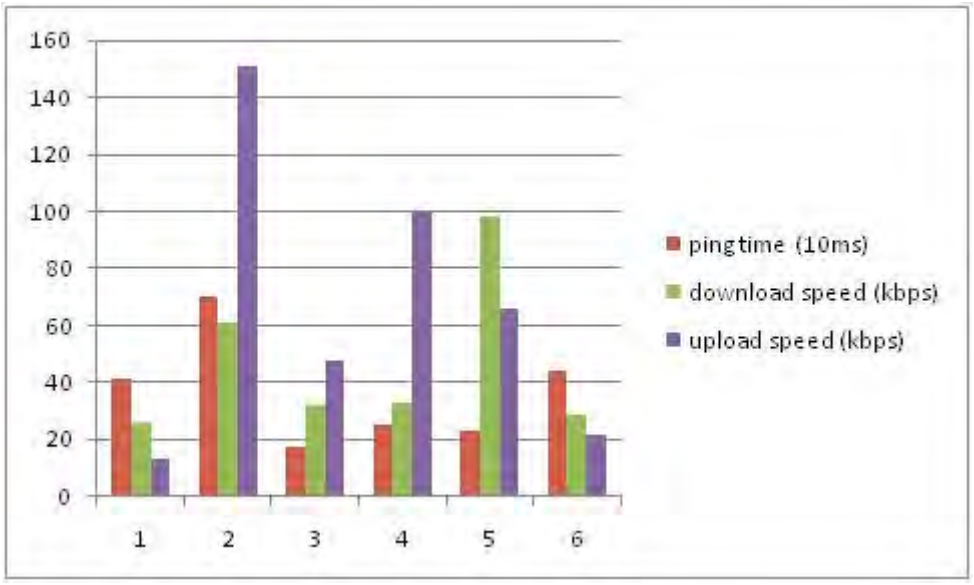


Figure 5. Graphic: Network -3G, Device: HTC One (Android 4.1), 6 test cycles

You can see from Figure 5 that there is a high variation in the results between the test runs. In particular the coefficient of variation (CV-standard deviation/mean) are over 50% for all measurements. This indicates that 3G users experience extreme variation in their application performance when accessing the network especially for upload speed.

Next, we wanted to see how this varied in different networks while keeping other variables constant.

Test cycle	ping time (10ms)	download speed (kbps)	upload speed (kbps)
1	45.8	8	7
2	36.8	12	13
3	24.9	15	9
4	44.9	14	8
5	51.5	15	11
6	44.1	9	13
StdDev	9.3	3	2.6
Mean	41.3	12	10
CV	0.225	0.250	0.260

Figure 6. Network -2G, Device: HTC One (Android 4.1), 6 times (test cycles)

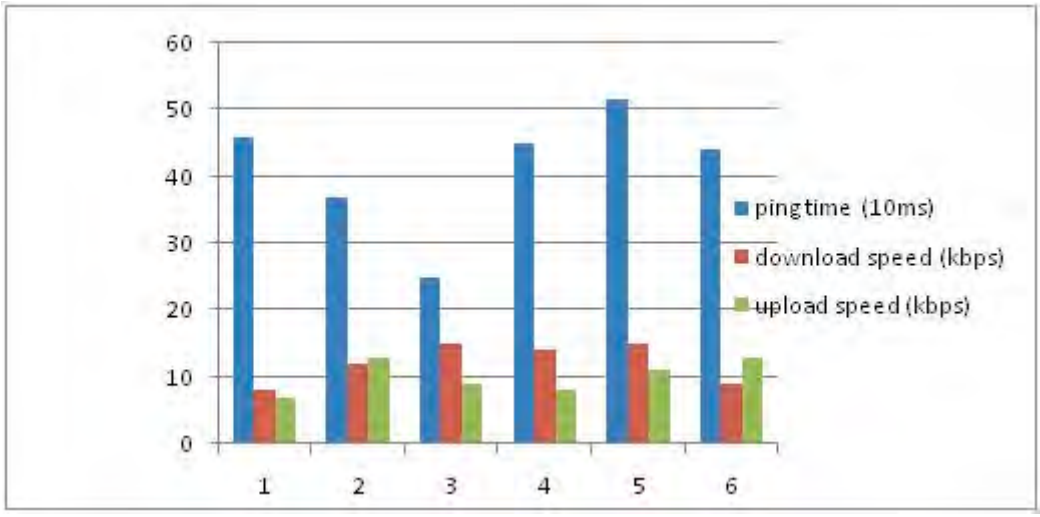


Figure 7. Graphic: Network -2G, device: HTC One (Android 4.1), 6 times (test cycles)

As shown in Figure 6, notice that although the download speeds were significantly slower with 2G versus 3G, the variability of the results as shown in Figure 7 is less than half of 3G network results. So although the speed is slower, the variability may impact applications that have 'bursty' traffic such as games. Now let's look at the WIFI results.

Test cycle	ping time (ms)	download speed (kbps)	upload speed (kbps)
1	96.0	960.0	430.0
2	87.0	930.0	470.0
3	99.0	940.0	470.0
4	64.0	970.0	480.0
5	51.5	970.0	460.0
6	88.0	970.0	460.0
StdDev	18.9	17.5	17.2
Avg	80.9	957.0	462.0
CV	0.234	0.018	0.037

Figure 8. Network -wifi, Device: HTC One (Android 4.1), 6 times (test cycles)

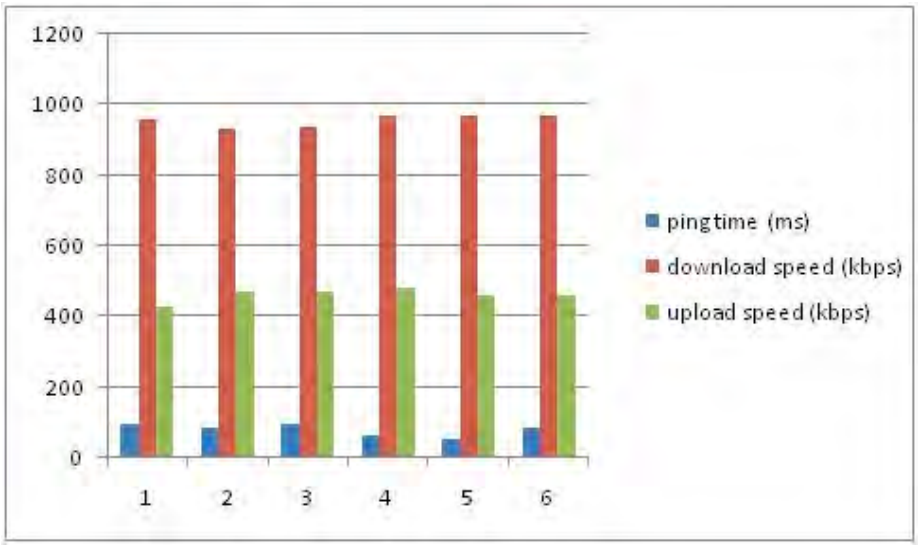


Figure 9. Graphic: Network -wifi, Device: HTC One (Android 4.1), 6 times (test cycles)

Just based on these simple tests, we can draw some conclusions. Yes, 3G is faster than 2G and of course WIFI is faster than 3G. But the real discovery here is the volatility and unstableness of the 3G network compared to 2G. Although it is significantly faster, the user experience may suffer due to its instability and performance variation.

I often make this choice in my daily life. Do I take the subway which is slow, yet predictable and deterministic but knowing I'll get there in 45 minutes. Or, do I drive and take the risk of traffic where I might get there in 10 minutes, but it could also be 30 minutes with 20 minutes of added frustration. With still some users on 2G networks, many on 3G, and now some users on 4G, mobile application developers need to consider the performance variability when designing their application not only for timeouts but the overall architecture and UI features for the optimal user experience.

Figures 10 and 11 summarize the averages of our experiment for 2G, 3G, and WIFI networks.

Test Network	ping time (ms)	download speed (kbps)	upload speed (kbps)
2G	413	12	10
3G	370	46.5	67
Wifi	81	957	462

Figure 10. Summary-Same device HTC One (Android 4.1), different network: 2G, 3G, Wifi (on average)

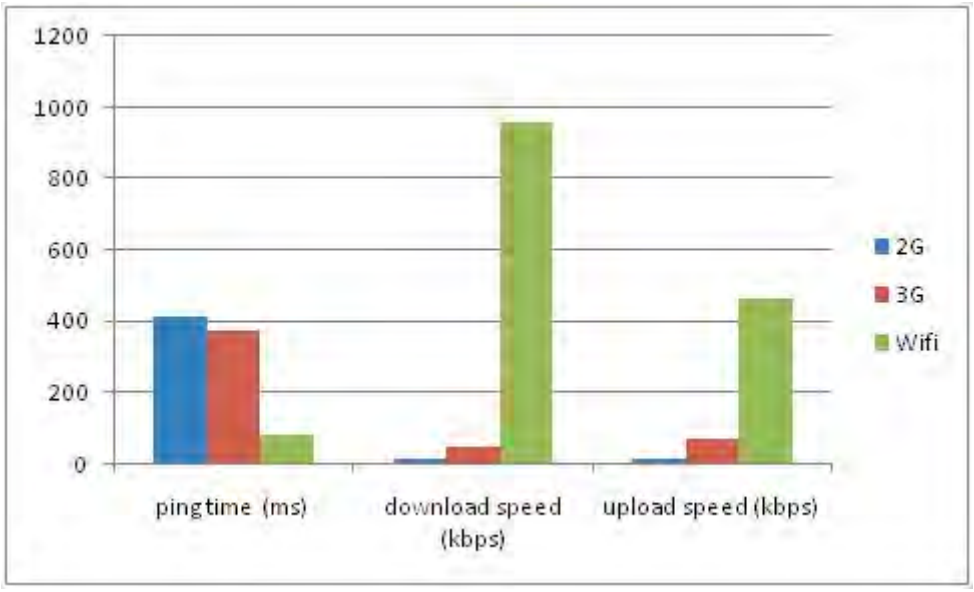


Figure 11. Graphic: Summary-Same device HTC One (Android 4.1), different network: 2G, 3G, Wifi (on average)

Obviously, different networks have different performance but the question is how the performance can influence user experience. Next we wanted to see how different device influence the user's experience of network performance. So we took the best performing network (WIFI) and tested it with 3 additional devices.

Devices	ping time (ms)	download speed (kbps)	upload speed (kbps)
HTC One (Android 4.1)	94	965	450
iPhone4S (IOS 5.01)	83	960	460
Motorola ME860 (Android 2.2)	90	950	430
iPhone5 (IOS 6.0)	88	960	450

Figure 12. Same network -WIFI, Different device: iPhone4S (iOS5.0), HTC One (Android4.1), Motorola ME860 (Android 2.2)

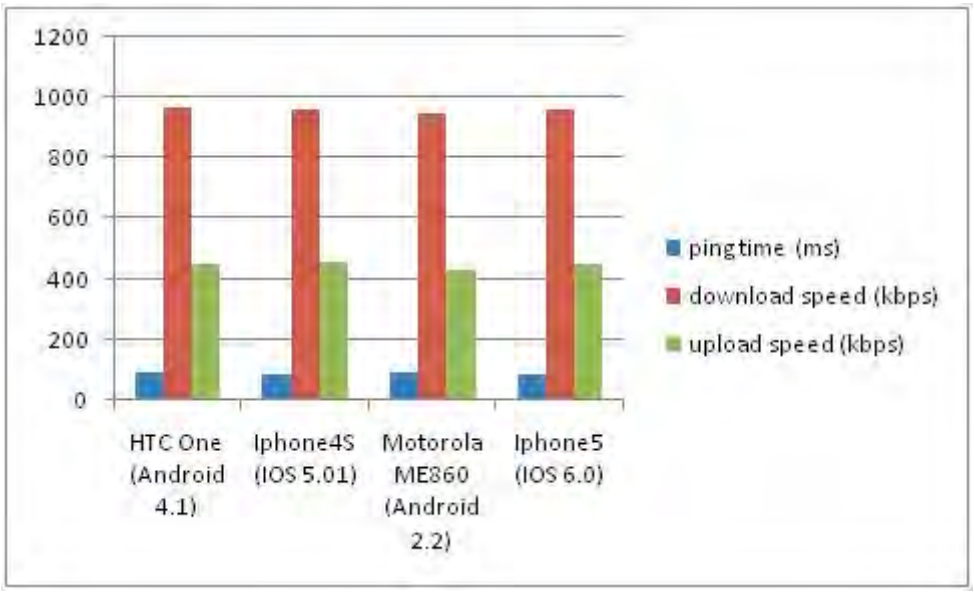


Figure 13. Graphic: Same network -WIFI, Different device: Iphone4S (iOS5.0), HTC One (Android4.1), Motorola ME860 (Android 2.2)

We can see it is very close. What is most interesting is that you often see advertisements for how fast a new smartphone is. Perhaps the phone does have great performance in that it allows users to swap between many applications quickly or that native mobile applications execute much faster than older model phones. But in reality, there are still many mobile applications that are web-based and not native applications. This means that much of the application's performance and user experience will be dependent on the network's performance and as we've shown, that can vary widely and is independent of the phone itself.

However, the network performance is still impacted by the application and the way it sends data over the network. Although our simple tests indicated that the device is not important regarding being influenced by network performance, different applications will send different data packet structures to the server and these data packet structures are also treated differently by the network.

Before testing your application's load on different networks, it is still a good idea to investigate your users' device models as different device models will send different amounts of data, even for the same application or web site accessed. Figure 14 shows a market share as of 4Q 2012, but it differs between different products in different countries. For example, if your application or site is accesses mostly in a developing country, maybe the IOS percentage will lower than indicated by aggregated statistics as shown.

Top Five Smartphone Operating Systems, Shipments, and Market Share, 4Q12
(Units in Millions)

Operating System	4Q12 Unit Shipments	4Q12 Market Share	4Q11 Unit Shipments	4Q11 Market Share	Year over Year Change
Android	159.8	70.1%	85.0	52.9%	88.0%
iOS	47.8	21.0%	37.0	23.0%	29.2%
BlackBerry	7.4	3.2%	13.0	8.1%	-43.1%
Windows Phone/ Windows Mobile	6.0	2.6%	2.4	1.5%	150.0%
Linux	3.8	1.7%	3.9	2.4%	-2.6%
Others	3.0	1.3%	19.5	12.1%	-84.6%
Total	**227.8**	**100.0%**	**160.8**	**100.0%**	**41.7%**

Source: IDC Worldwide Mobile Phone Tracker, February 14, 2013

Figure 14. Smartphone OS shipment and market share

Device and Application effect on Network Performance

So given a particular device, if you just take into account pure network speed, it makes sense that the results would be close to identical as shown in Figures 12 and 13. However, what happens for the different devices but for the same application on an identical network. Does the application behave differently and display different performance characteristics with different devices even with the same network? To investigate this, we loaded our own home page with 3 simulated devices using Mobitest, a tool developed by Akamai.

http://www.xbosoft.com	iPhone 4, iOS 5.0	Nexus S, Android 2.3	iPad 2, iOS 5.0
Load Time	8.46s	5.13s	3.64s
Bytes	344.62kb	263.88kb	891.52kb

Figure 15. Same application, same network, different device

So, what is interesting in this analysis is that our site, www.xbosoft.com, can sense a mobile device, a phone, and take the user automatically to the mobile site. As you can see, the mobile site is optimized with a totally different application and different functionality and screen, thus the different amount of bytes loaded. What is notable is that even though the Android 2.3 phone and the iOS phone render the same site, m.xbosoft.com, their performance is significantly different. We ran this test many times with consistent results where the bytes transferred was about 30 percent more for the iPhone, and the load time was 30 to 50 percent faster for the Android device. This indicates that the application and device does have an impact on the network performance due to different packet sizes and thereby amount of data transferred by the particular device for that application.

Troubleshooting Poor Performance

So, once you do discover that there is a potential problem with an application on a particular device, and/or a particular you can dig deeper into the data being transferred. See our previous article on mobile server performance: Testing Performance of Mobile Apps - Part 2: A Walk on the Wild Server Side: http://www.methodsandtools.com/archive/mobileserverperformance.php.

In the previous article, we discussed how load times of the data should be examined via content download metrics. However, you can dig deeper to discover further information that can help you troubleshoot based on network performance issues.

One way to dig deeper is to use a HAR (HTTP Archive) viewer. An HTTP Archive is a common format for recording HTTP tracing information. This file contains a record of each object being loaded and the time. Information contained includes:

- time to fetch the DNS information
- time each object takes to be requested
- time to connect to the server
- time to transfer from the server to the browser of each object
- whether an object is blocked

With a HAR viewer, you can examine these measurements in detail for each object. Looking at these measurements, our previous article focused on the time to transfer from the server to the browser of each object, hence download times in that article shown in Figure 8: Waterfall View of time of each object downloaded which can be partially solved by caching. However, other measurements such as time the object takes to be requested, and time to connect to the server are more indicative of poor network performance. To examine the data, the HAR file format can be read by many viewers, each with a different format and view of the data. The Mobitest tool has a built in viewer that enables us to look at particular images and http requests that could be causing the delays as shown in Figure 16.

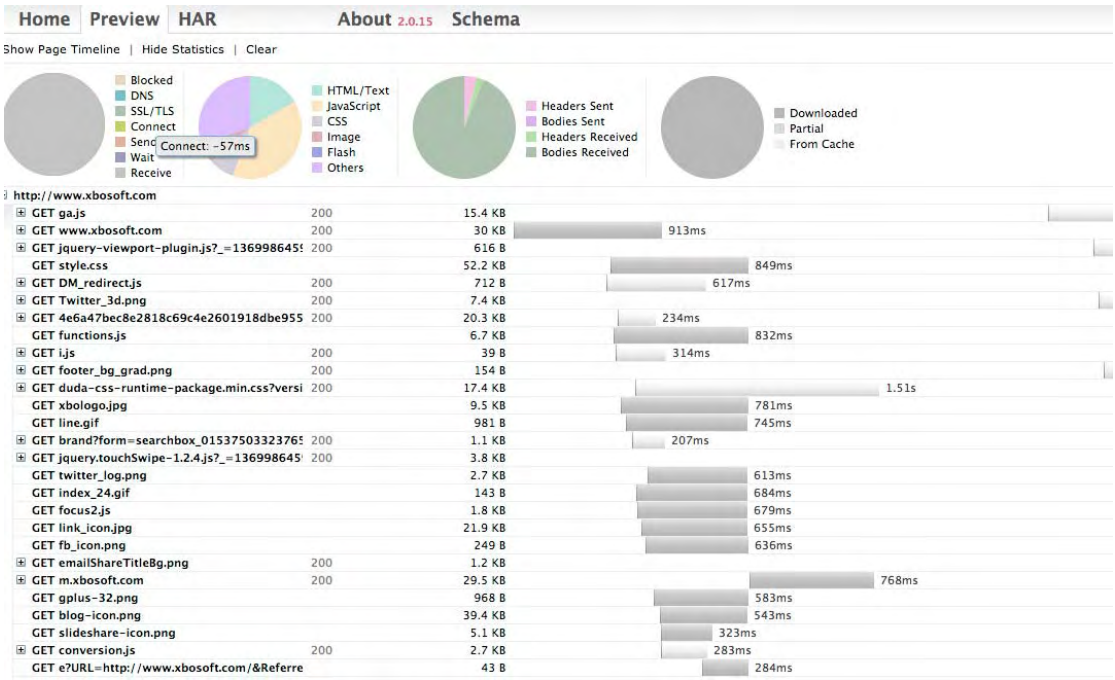


Figure 16. Mobitest HAR Viewer showing results from iPhone4 with iOS 5.0.

As you can see, it shows each object and the load time along with other statistics shown in the pie chart. On the pie chart, when you mouse over the different types of delays (blocked, DNS, SSL, etc.), you can see the time delay. In this instance, we can see that the connect time is 57 ms. This is much different than just examining download speeds because the connect times which are the time that the server establishes communication with the browser, can be

influenced by protocols and packet sizes as they initially begin to start communicating. This is more dependent on the network itself and its configuration rather than the pure speed of the network itself. In this way, you can see if your application is suited for, or can be tuned to communicate better depending on the network it is predominately used on. Looking down further in the chart, we also see that m.xbosoft.com was loaded about halfway down the page with a download time of 700+ ms. With this kind of information, you can really dig into what could be slowing things down and possibly examine the order of things that could effect the user experience.

Just for fun, we decided to use another HAR viewer and another web site to see the results from using a different tool. Shunra's NV also contains a HAR viewer. We tested ABCNews.com for both abcnews.go.com/m and abcnews.go.com. As with www.xbosoft.com, for mobile users, after inputting abcnews.com on a mobile browser, the browser will send a request that includes the browser's information to the abc news server. Then the server will judge if the request is from a mobile or desktop platform and send the mobile user to abcnews.com/m, and desktop user to abcnews.go.com.

Let's do some analysis. We used Shunra's NV to get the following figures that show a waterfall timeline view for http://abcnews.go.com/m tested with3G on Samsang Galaxy 3 (Android 4.1 platform).

Legend: (DNS = Domain Name System)

First View:

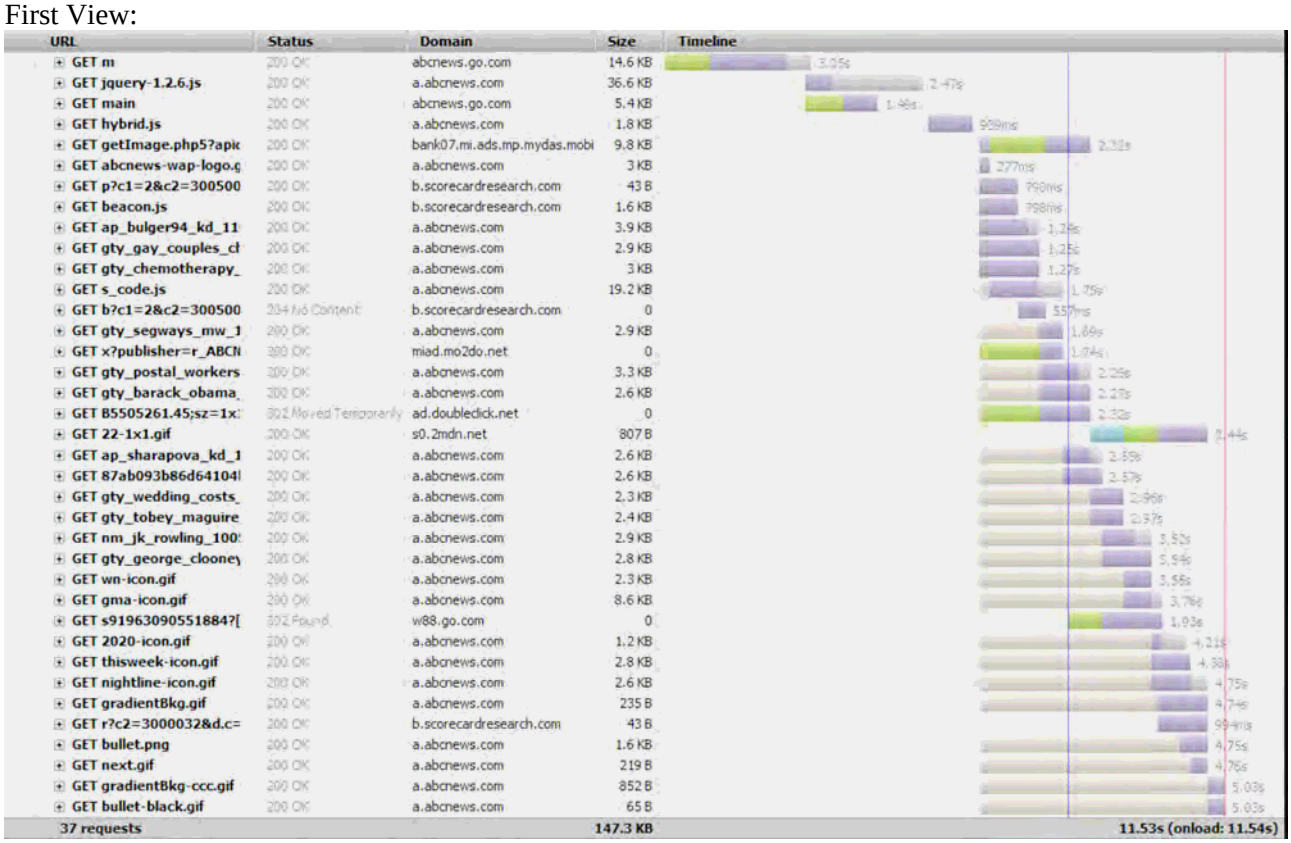


Figure 17. Step HTTP Download Graph via Shunra's NV HAR Viewer

From Figure 17, we can see which parts need to be improved or optimized. As can be seen on bottom right corner, total download time was greater than 11.5 seconds (download 147.3KB data 37 requests sent). We can also see that some image files (gif and png files) take a long time to download, so the developer should consider making the images smaller. Especially for a site optimized for mobile, many forget that the screen sizes are so small, thus high resolution images are rarely needed.

Mobile Network Performance Affects Your Application

These days, many companies operate both a mobile site, or site specialized for mobile access and a normal web site for desktop access. Adobe did a survey that indicated that 45% [2] of web sites only have a desktop site currently, but that mobile devices still can visit their web site. As desktop web sites are not specially designed for mobile, they usually open slowly and have UI issues with some components (menus, pictures) sometimes displayed out of order in addition to being very slow as we've shown above in our analysis. Because of this, if you don't have a specialized mobile site, then you need to design your desktop site with considerations for mobile users. In doing so, you not only have to design the screens and workflows more concisely to accommodate slower network speeds, but you also need to design for different and complex mobile scenarios. These scenarios and the user experience depend not only on your mobile software and its UI design, but also its technical adaptations to potential network performance issues.

[1] http://fonolo.com/blog/2012/03/customer-experience-statistics-2012/
[2] Adobe 2013 Digital Marketing Optimization Survey

Agile Scrum Sprint Length: What's Right for You?

Brian Haughton, Director - Consulting, Advanced Solutions Practice
Cognizant, www.cognizant.com

When a team begins a project using the Scrum approach, one of the first questions that often arises is, "how long should our sprints be?" After the collective team understands what a sprint is and how Scrum works, several discussions on the topic usually ensue. This paper reviews the purpose of the Scrum timebox, the reason why two weeks has become the industry standard and some considerations for when another timebox may be appropriate for your particular situation.

Timebox Duration Considerations

Teams new to Scrum struggle with questions of duration and expectations of what can be accomplished in a Scrum sprint. Typical questions often arise: "How long should it be? How can you expect me to get all that done in that short amount of time? How do we know we've chosen the right length?"

You will hear Scrum experts say that a sprint can vary from one to four weeks, depending upon the project and the team members involved. What's appropriate for your project?

In Scrum development, it's common for new teams to pick a timebox at the longer end of the scale because they don't understand how they could perform design, development and testing within a shorter period and produce a quality product.

Their thinking is, if they have more time, they will be able to accomplish what they commit to. For teams new to Scrum, there might be reasons for choosing a longer duration. Perhaps they would like some padding, or they lack confidence in their ability to deliver in a shorter amount of time. Perhaps they are not sufficiently insulated from the enterprise and have many dependencies on outside teams that cannot respond quickly. ("The DBAs have a 36-hour SLA, so we'll need to wait…")

Whatever their reasoning, let's review the variables and consequences driving the decision.

The Measuring Checkpoint

Sprints in Scrum are unchangeable timeboxes that provide the team with a "measuring checkpoint" for gathering information about progress and making adjustments to scope, staffing, forecasts, etc. With this purpose, the more frequent the checkpoint, the more often metrics are gathered and the faster the cycle time for the team to inspect and make adjustments, leading to a more efficient process more quickly (e.g., do they need a DBA to join the team?). Based on what the team has accomplished, how should the release plan (the forecast) be updated? What can management do to help the team achieve a faster velocity?

Note that this checkpoint has nothing to do with the type of project or the amount of work the team chooses to commit to. Also, for a large company's projects where a project management office must be included, this provides the PMO and leadership with business intelligence data from which they can help the team accomplish its goals.

The Sprint Review (a.k.a. Demo)

The sprint review meeting provides an opportunity for the team to hear feedback from product owners, peers and anyone else who attends. The more often a sprint review occurs, the more often feedback can be provided, incrementally improving the product. If this occurs on a shorter cycle time, the cumulative effort of the small improvements resulting from a series of frequently held reviews will yield a better product than that of a longer cycle time. In fact, late in a project, the team may wish to consider a move to one-week sprints to increase the opportunities for feedback as it prepares for a production release.

The Retrospective

The retrospective is a key aspect of the inspect-and-adapt Scrum cycle. The more often a team pauses to consider how to streamline its process, the sooner it will identify process issues, attempt to address them and reach terminal velocity. Frequent process improvement meetings also allow for a team to provide feedback to management on productivity drags, such as spending time creating status reports that no one will use or creating a business requirements document that no one will read. Not unlike the fixed political timebox of U.S. elections, the retrospective may be used as an outlet for individuals' process frustrations, not necessarily to vote anyone out of office, but to allow people to voice their opinion.

Sprint Planning

The perception that moving to longer duration sprints will reduce pressure and provide breathing room without a cost is a "false truth." Indeed, it may provide breathing room at the cost of team velocity. Consider the time change during Daylight Saving Time, when people actually have one day a year with 25 hours. How do they spend their "extra" hour? Some may choose to work more, others may choose to sleep or rest more, and others may choose to do more. The point of this simple analogy is that people will adjust to fill whatever time they have been given.

We postulate that the smaller the sprint duration, the faster the engine can run. Smaller durations squeeze out the fat and can only run lean. Consider that if the team knows it has up to three weeks to perform a task, it may spend time researching a better solution than its "first thought" design. However, if the team only has one week, it must quickly implement what can be accomplished in the sprint.

The Commitment

An argument for shorter duration sprints is the issue of how the team handles the absence of a key team member. Regardless of the reason for the absence, the general rule is that the team acts as a unit and must pick up the slack to meet its commitment. This places a significant burden on the remaining team members for the duration of the sprint once the absence begins. In longer duration sprints, the team has committed substantially more points than a shorter duration sprint, and if a person is out for a portion of that longer sprint, the remaining team members must either carry the load longer before they can resize velocity or de-scope items to which they've committed. We've found that, in shorter duration sprints, when a team member is absent, teams are more likely to complete the sprint with the original scope than de-scope it. The reason: Teams can "endure pain" for a few days.

The Unbroken Think-Stream

One consideration for longer duration sprints is that software development is, to some degree, more art than science, and it requires a level of creativity that cannot be rushed. Certainly, it seems obvious that if the team performs one-week sprints, it has only three actual work days — the first day is spent planning and designing, and the last day is spent on the review and retrospective. On longer sprint durations, say four weeks, the team has 18 unbroken days for the team think-stream. They have time to dream, be creative, identify high-risk design issues and address them, toss out their first or even second version of an idea and restart again and again before bringing their ultimate idea to the sprint review. This degree of creativity and the pursuit of perfection are not to be found in deadline-driven software development.

Human Nature and Reality

The boots-on-the-ground reality is that the same thing is going to happen at the end of the three-week iteration that happens at the end of a one- or two-week iteration: Time allocated for testing and preparing for the sprint review will be squeezed. What we've found is that it's more a matter of the team adjusting what it can do during the iteration duration and setting expectations appropriately among themselves than the alternative, which is trying to lengthen the iteration so the team can finish the stories they committed to. The "better" solution to this point is to reduce the amount committed to, break the stories into smaller tasks, pick up more stories opportunistically, deliver functionality earlier in the iteration and allocate time to test and prepare for the sprint review from the start.

Procrastination

One aspect of human nature is for people to postpone and procrastinate on work that they know they need to do. While this must have a scientific name, let's affectionately call it "college term-paper" thinking. Why? The stereotypical college student who is assigned a research paper at the start of a semester will think he doesn't need to start on it right away; the end of the semester (or "term") is many weeks away. He does other things and comes back to it about three weeks before it's due, only to realize that he is going to need more hours than are left to produce a good paper. His procrastination will result in a lower quality paper, with less content or depth on the topic assigned.

The same may happen with a Scrum team with a longer duration sprint. It may not "feel" like a sprint if the duration is four weeks. Instead of diligently using the unbroken time for a think-stream and following a rapid creative-destruction cycle, the team slowly starts on the stories, building on its previous sprints' work but realizing with only a few days left that what it's built is not adequate.

Visibility into Team Challenges

Professional developers are smart translators that like to solve puzzles. In the mindset of a stereotypical developer, when a challenge is encountered in the course of developing a component, they put all their energy into solving it. In this scenario, one possible path has been taken many times by developers. The developer becomes consumed by the challenge and doesn't report it as a block to her Scrum master; after all, she's smart and, in the past, has solved more "hairy" challenges than this one.

The catch is that, in a short-duration sprint cycle, the challenge must become visible to the Scrum master. This is against the developer's nature - she doesn't want to bother the Scrum master with a "non-issue." So, what happens? Often, the developer works to solve the challenge right up to the deadline but fails to finish it per the acceptance criteria, and the product owner is surprised that the functionality was not completed in the sprint review. There is a danger here; if this behavior occurs in the context of a longer sprint duration, the opportunity cost will be quantified in the form of lower velocity.

The Story Creation Debacle

Another consideration for longer duration sprints is the time it takes to flesh-out an epic into a family of fully developed stories and document the subtasks for each. The reality is that some topics require lots of time from the subject matter experts and may need several "reviews" with the product owner, his peers or other stakeholders before the acceptance criteria for a family of stories is ready for developers and testers to start translating that story into functionality.

This is where we need to recognize that it will take time to draft, socialize and refine stories as the team moves through the story creation process, and we should forecast upfront that a particular family of stories will need more than one sprint to complete. For some stories, three weeks won't be enough. In previous projects, we've found that some story families can take up to eight weeks to produce because of the subject matter experts and/or the complex nature of the topic.

With that said, it is not wise to move the sprint review, retrospective and "metrics gathering checkpoint" just because a "story author" team member needs more time to finish a family of stories. A better approach is to extend the time to create that family of stories across sprints and provide updates at the end of each sprint as to where the group is in the process.

Another approach is to establish a checkpoint with one or more business "peer reviewers" that would verify that a story is ready for development. This validation step could also act as a measurement checkpoint for the rate of story creation, which would provide one more data point for management and the team to understand its velocity and productivity.

Some Metrics to Consider

Let's assume for a moment that your Scrum team is struggling through two-week sprints but is able to produce 100 story points of velocity for each sprint. They are struggling because of pressure to complete the stories to which they committed, but they're not working weekends to complete them. However, the testing activities are squeezed, and the team doesn't feel it has the proper time to prepare for the sprint review. At one sprint retrospective, a member proposes that the team switch to three-week sprints to reduce the pressure, perform appropriate testing and prepare for the sprint review. The real issue is that the team is committing to deliver too many stories in one sprint, and they need to adjust the number downwards. However, let's consider this proposal from a simple metrics perspective.

We suggest that moving to three-week sprints does not actually solve the underlying issue. Our rationale: It is false to extrapolate that the team will now average 150 points in the new three-week sprints. Typically, the team will drop back a little because it perceives that the pressure is less, say, to 130 points in the three-week sprint. The team may now "feel" it has breathing room, but it was bought with a 13% reduction in throughput.

In terms of velocity capital over a six-week period, if the team could average 300 points across three two-week iterations, and if they move to two three-week iterations, the forecast is that the team will only be able to produce 260 points. Over this timeframe, the reduction in velocity capital will result in 40 points of less functionality or fewer defects fixed in the final product. For a team with an average of four story points per story, that means it will complete 10 fewer stories over a six-week period. That represents a significant difference in functionality. A product owner would probably balk upon hearing that news.

In a baseball analogy, the difference between an average hitter and a great hitter is only one additional hit per week. If a team can do the same — that is, commit to one additional story per week — it will raise the bar and complete the project with a much better product than what an "average" team would have produced.

Should the team keep up the pressure, it may be able to produce the same as it would in the two-week sprints, but it is unlikely it will exceed that. In other words, on this point, there's little upside to switching to three weeks but a significant potential on the downside.

Two Weeks: The Standard?

The two-week sprint duration (10 business days) has become the de facto industry standard. Why?

The two-week sprint has the best balance of all the above factors. It allows for a team to have a small amount of creativity with eight days of unbroken think-stream, and it provides a near-term deadline that kills procrastination and forces developers' challenges to the surface more quickly. Moreover, this approach is often enough to gather feedback from the sprint review and reflect on the process. Finally, it sets a pace for measuring checkpoints twice a month (leadership usually likes that), keeps the pressure up and "feels" like a sprint.

What's right for your project? That's a decision you'll need to make. Perhaps you are in a situation where your team members lack experience or a key skill in a particular area, or your team is struggling with breaking down the stories into tasks, and you conclude that two weeks is just too short. (For a summary of various sprint lengths and their challenges, see Figure 1.) The key with longer duration sprints is making them "feel" like a sprint by keeping up the pressure; keeping them lean; making sure the team has challenged themselves; avoiding the pitfalls of procrastination, lack of visibility into developers' challenges and the absence of team members; and allowing story creation to cross sprint boundaries.

Sprint Length Comparison

Challenges	1 week	2 weeks	3 weeks	4 weeks
In a four-month project, "ceremonial days" for sprint planning, review and retrospective	32 days	16 days	11 days	8 days
In a four-month project, minimum number of days spent testing	~16 days	~8 days	~8 days	~12 days
Impact on other team members when a member falls absent in the middle of the sprint – length of the "endurance test"	A couple of days	Several days	More than a week	A couple of weeks
Unbroken thinking time, allowing for the team to address challenges as they arise	3 days	7 days	12 days	16 days
Visibility into team challenges	Best	Second best	Fair	Poor
Likelihood of falling into the "procrastination trap"	None	Low	Medium	High
Adaptation to uncertainty in story creation	Many stories cross sprint boundary	Some stories cross sprint boundary	Few stories cross sprint boundary	Rare for stories to cross sprint boundary
Application feedback	Best	Better	Good	Fair
Process improvement cycle	Fastest	Fast	Frequent	Slow
Adaptation to F500 enterprises that still operate in traditional models	Unsustainable without isolation strategy	Difficult due to many escalations required	Escalations required but less frequently	Best; few escalations required

Figure 1

The challenge with shorter duration sprints is breaking the stories into small tasks and reassembling them into a unit of work that will be meaningful to the product owner. Consider this: In a one-week sprint, the team will have three productive days to design, build, test and polish a component. Given the uncertainty in software development, this doesn't allow for much time for recovery when the unexpected occurs, causing velocity to appear uneven across sprints. Also if the increment produces too small a delta for the product owner to notice the update or change from the last sprint review, then your sprints may be too short.

Sprinting Ahead

Consider carefully for what purpose you might deviate from what has become the two-week standard. Note that there is a series of reasons why two weeks has become the default. You should ask the question, “if we change to a different length sprint, what problem does that solve?” Are there underlying problems, like personality or behavioral issues that are manifesting themselves in other ways that would be best addressed in another way than changing the sprint duration? Generally speaking, it is better to set the cadence and have the team adjust to it rather than the other way around. No matter what sprint duration your team chooses, make sure everyone buys in, then run it as lean as you can, and keep in mind the considerations raised in this paper.

Software Developer Business Patterns

Allan Kelly, Software Strategy Ltd, allan [at] softwarestrategy.co.uk
http://www.softwarestrategy.co.uk/allankelly
- Twitter: @allankellynet

Building software is hard: successfully bringing a new software product from conception to market is harder. Building a successful software company that develops and markets multiple software products is harder still.

The days of 'build it and they will come' are over. Simply creating a great piece of software and waiting for the customers to knock on your door no longer works – if it ever did.

Nobody should ever think that building software is easy: some small application may indeed be easy to build, but meaningful software applications and platforms are extremely complicated things to create. However, building software is at least a reasonably well-defined task, which isn't true of many of the other things that need to be done. Indeed it isn't even clear what needs doing.

Creating a successful software product, and building a successful software company, involve a myriad of other activities that are far less easy to define and put boundaries around. The first of these is just deciding what it is that should be built. But perhaps the most difficult task of all is bringing all these activities – well-defined, poorly defined and never defined alike – into alignment so that they move towards a common goal. This is akin to an exercise in formation flying in which the individual pilots have their own opinion on where they should be going. The bigger the company, the more complex the environment, the more difficult the task.

Still, every seven years or so the industry comes up with a new paradigm which allows small pieces of software, sometimes written by one person, to advance the ecology itself. A few become millionaires while many have walk on parts.

Apps are the latest iteration of this cycle. Even the word *application* is too big for the small pieces of software we download and install on our phones, tablets or other gadget. iOS Apps, Android Apps, Facebook Apps and now Apps for televisions. We've been here before, last time it was the web, and before that Windows, before that MSDOS and before that Apple IIs and TSR-80s.

That is the way the software industry is: high growth, high failure, dynamic, loved of politicians and venture capitalists. The barriers to entry are shockingly low: as long as you can code, have an idea and can live on credit cards for a few months you can play too.

Yet the barriers to success are extremely high. Just because you can play doesn't mean you will succeed. Infant mortality is shocking. And we repeat the same mistakes over and over again. Perhaps the biggest mistake is to believe that if we build it they will come. Those days are over, if they ever existed in the first place.

Successful software companies know about marketing, product strategy, roadmaps, customers and many other things. In fact, successful software companies have a lot in common. There are, just as in code, patterns of success, patterns of business.

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

Over the last eight years I have collected many of these patterns. Often identifying and writing one pattern leads to the discovery of several more. All patterns are available for download from my website - www.allankelly.net - and 36 were published as a book in early 2012, *Business Patterns for Software Developers*. In the rest of this piece I'd like to discuss a few of the patterns and how they fit together.

Same Customers, Different Product

Once your business has a relationship with your customers they will trust you, satisfied customers have a positive impression of your company and your. Sales staff already have a relationship with the customers and the company is in a good position both to understand their needs and to supply them

Modern consumers are faced with a bewildering number of brands and product variations. Choosing a trusted brand is one way to navigate the jungle.

Nor is it just individual consumers who return to existing suppliers and brand. Business customers find it easier to buy from suppliers they have worked with before; and the promise of a compatible product is a big plus.

So rather than continually seek out new customers for the product you have consider the customers – rather than your products – as the asset. Expand your product offering so you have more products to sell to your existing customers. Do this by focusing on the customer

This approach has the additional benefit that by deepening your relationship with the customer not only will you sell more, the customer will get a better solution and the relationship will be strengthened further.

Some sales will build directly on the products you have already sold – sometimes called "plus one" sales: more licenses for the same software, upgrades to enhanced versions new versions of existing software, and add-on modules are all possible. The next sale could be a service: support contracts, training and installation are highly applicable to existing products. In fact, when you think about it there are actually four possibilities for sales, as shown in Figure 1.

Same Customer	Consumables, e.g. repeated razor blades or printer ink. Sell to other groups within the same corporate.	Customer is the asset, find or develop new products to sell to the customer.
Different Customer	Product is the asset view.	Diversification: a new product in a new market.
	Same Product	**Different Product**

Figure 1 - Four growth options

Different Customer, Same Product is probably the default position of most companies – "We have a product, we need to find more customers!" Yet making sales can be very expensive in itself, customers might already be using a competitor product or new sales channels might be needed to reach new customer groups.

Same Customer, Same Product is normally found with consumable products - Gillette razor blades or HP ink jet cartridges but it could be software upgrades, I must have bough five or six copies of Microsoft Office in my life.

When products last, like software, ways need to be found to entice the same customer to buy more of the same product. If the customer is a large organization then it might be possible to move from one group within the company to another selling licenses as you go. While the paying customer might be the same the end user will be different.

For home users it might be an upgrade cycle – new OS, new PC, new Office version. Although even this might be considered a Same Customer, Different Product model depending on how you view upgrades.

Diversification - Different Customer, Different Product – is inherently risky because it is, in effect, establishing a new business. Sometimes companies pull this off, like the way Caterpillar extended its brand into clothing. Perhaps more often this is one jump too many, look at the way many tobacco companies tried to move into food only to retreat later.

Same Customers, Different Product view customers, not the product, as the asset. Rather than sell an existing product to new customers you sell existing customer a new product. When done well this can be highly effective at low risk. Think of Oracle building on the database customer base to sell enterprise software and middleware, or the IBM Rational developer toolset.

Nor does the different product need to be your own product. You could use the patterns Value Added Reseller or White Label to offer products to your existing customer base under your own name. Offering another product as a complement to your own can reduce costs and eliminate competitive tension – something described in the pattern Complementor Not Competitor.

Of course it is sometimes difficult from the outside to tell what a company's strategy actually is. Consider Dell's recent acquisition of Quest Software. This might be a diversification move – Different Customer, Different Product – or attempt to sell software to existing hardware customers, or possibly vice versa.

Homogenous and Segmented Customers

In the beginning many companies assume – by conscious decision or through naivety – that all customers are homogenous. Indeed, this approach has some advantages: it reduces time to market entry and therefore costs. By entering the market it puts the company in the best possible position to understand what customers want and what is possible. A homogenous customers approach has worked well for the likes of Microsoft – with MSDOS – and even Google's early search product. If you consider all customers to be the same then one product is all you need to enter the market – thus keeping costs down and time short.

In the beginning companies may simply lack an understanding of potential customers. Sometimes the best way to gain knowledge about customers is simply to get into the market. However, sooner of later companies start segmenting their customer base so to serve customers better, and to extract more money.

Segmenting makes sense: customers are not homogenous. Different customers want different things, some value speed or ease of use, others value low price. Many a software development team has been lead astray by focusing on the needs of one customer too closely and producing a product which has little, or no, use to others.

The trick is to use Customer Understanding (another pattern in the book) to segment customers into different groups and address the needs of each group separately. Segment groups are typically defined on discernable attributes and characteristics that allow differentiation of one group from another; another approach is to segment on the tasks to be performed. Either way, working with definable groups avoids generalizations that do not accurately describe any one group.

You may choose not to meet the needs of some groups if doing so would compromise the needs of another. When resources are limited is it better to target resources than spread them thin. Segmenting your customers will allow you to segment your market. As you decide which customer groups to serve you will define your market position relative to your competitors. You are also deciding whom you will not serve. In some cases you may need to extricate yourself from some existing group to pursue your new targets.

Segmentation can help avoid situations where more attention is paid to the customer who shouts loudest. Strategy is as much, or even more, about what you will not do, whom you will not serve, as it is what you will do and whom you will serve.

Those you decide to serve will benefit because they will get a product that more specifically fits their needs. A deliberate choice not to serve some groups allows you to serve others better. Sub-dividing your customer base will also help you spot opportunities to serve customers better.

But customer segmentation can go too far. While for customers the sheer choice of products on offer under the same brand can be overwhelming. Last year I needed to buy a new car GPS system, confronted with what seemed like several thousand products on Amazon I stuck with the brand I knew, Tom-Tom, and bought the cheapest product rather than spend time considering the merits of each product. (True, I momentarily considered a Garmin but was again overwhelmed by product choices.)

More dramatically contrast the approach of Nokia and Apple to the phone market. Nokia markets an amazing range of different phones. Conceivably there is a Nokia for every customer segment – indeed there might well be more than one. There comes a point were the costs of such a diversified product range become self-defeating.

Now consider Apple's approach to the same market: there is one. There was one iPhone to start with and although Apple continues to offer older versions, and some Simple Product Variations (another pattern) there is essentially still one iPhone.

Apple too have segmented the market but having done so they decided to only address the premium smart phone market. Even here segmentation is limited. A quick look at Nokia's website one day in June 2012 showed four different Lumia Smartphones, six other smart phones and at least a dozen simpler phones.

On the face of it addressing a segmented customer base with a range of products targeted at each segment should be a more effective approach than homogenous customers. But as the Apple v. Nokia story shows things aren't always that simple.

Sequences and competitive advantage

Patterns don't exist in isolation. If one company faces a problem and solves it using the same pattern as another it is quite likely to find the next problem is also quite similar and the corresponding solution is also similar.

Figure 2 shows one such pattern sequence. The start of this sequence is the desire to reach more customers. After segmenting the customer base the company engages Simple Product Variations pattern – maybe a range of colors products al la the iMac or iPad covers – and Core Product Only. (While Core Product Only works in some domains it doesn't have great history in the IT world. Consider Microsoft Works, it has most of the Office functionality most users need but has never sold well. Even Linux seldom appears on its own, it is usually the core of a distribution or baked into a product like Android.).

Consequently the company has a Product Portfolio. In order to reach the different customer segments – and limit competition with itself – the company then engages a variety of distribution channels.

While one pattern may address one set of problems right here and now in resolving the question new issues arise. Thus patterns tend to lead to other patterns – a pattern of patterns usually know as a *pattern sequence*.

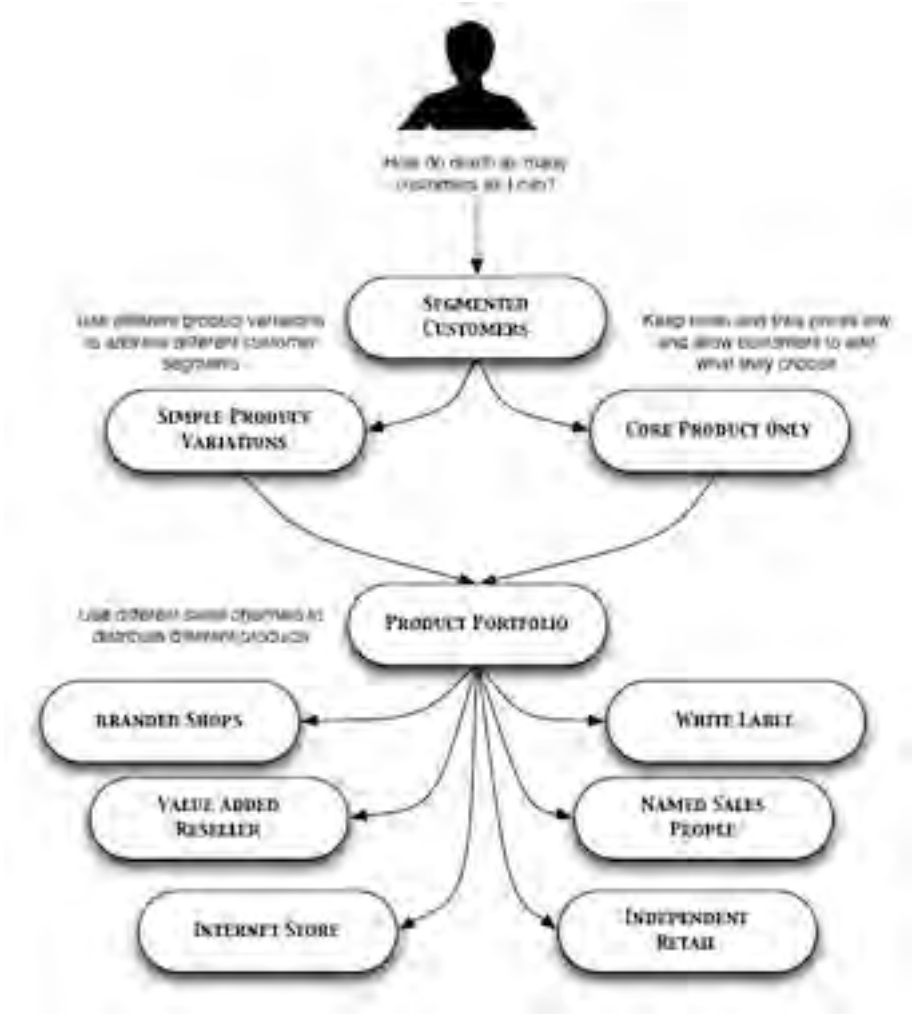


Figure 2 - Pattern Sequence for reaching more customers

That different companies follow the same patterns and same sequences isn't a bad thing, in fact it validates the whole approach. For many companies the product, not the process or organization, is the competitive advantage. Why reinvent the process wheel when plenty of others have already shown how to play the game?

That said, for some reinventing the wheel is the competitive advantage. The way the product or service is delivered, or the way the company works, could well be the things that make the company a winner. The actual product may be very similar to the competition but delivery very different. For these companies business patterns serve not as a template for what to do but rather a description of what not to do.

Finally

The software industry, indeed the wider technology, industry doesn't stand still. As the patterns were re-edited for *Business Patterns for Software Developers* it became clear that some updates were needed. For example, Nokia was cited as an example in several patterns. In some case the pattern and example still held. In others, the events at Nokia lead to new insights into the business of technology.

A year after publication, nearly two years since the most of the text was complete the patterns still stand up. I believe the patterns themselves will stand the test of time but I expect some of the details and examples will change. I also expect that using this common languages for much of what our industry does will allow software developers and entrepreneurs to come up with new variations, new combinations and completely new approaches.

References

Kelly A., Business Patterns for Software Developers, 2008, John Wiley & Sons.

Git-TFS - Work with your Team (Foundation Server) with Git

Matt Burke, spraints [at] gmail
GitHub, http://pickardayune.com/

Git-tfs is an open source two-way bridge between Team Foundation Server (TFS) and git, similar to git-svn. It fetches TFS commits into a git repository and lets you push your updates back to TFS.

Web Site: http://git-tfs.com/
Current Version: 0.17.1
License & Pricing: Open Source (MIT), Free
Documentation:
Wiki: https://github.com/git-tfs/git-tfs/wiki
Support:
Mailing list: http://groups.google.com/group/git-tfs-dev/
Issues: https://github.com/git-tfs/git-tfs/issues
IRC: #git-tfs on freenode
Project : https://github.com/git-tfs/

I am a source control geek. Keeping versions of my code and being able to reliably synchronize it between computers is something I can't do without. So it shouldn't be a big surprise when I admit that, once upon a time, I was excited to use CVS. I was subsequently thrilled to use Clear Case, Subversion, and TFS.

When I first learned about distributed version control tools (for example Mercurial or Git), I was quickly won over by the ability to version anything, anywhere, without ever needing to set up a server. This was exactly what I needed for my side projects. It allowed me to keep a history of changes regardless of whether I was online, and I could synchronize my changes later.

Being able to work offline got me interested in Mercurial and Git, but being able to quickly and easily experiment was what really hooked me. One of the big advantages of a distributed version control system (DVCS) is its ability to sanely handle branches. Every time you create a workspace, you have created a branch. When everything is a branch, there is a lot more focus on making merging as frictionless as possible. This makes "what if?" experiments cheap and easy.

Introducing git-tfs

While I was learning about Mercurial and Git, my employer was using Team Foundation Server (TFS). The git-tfs project was born as an experiment to see if I could use Git while my co-workers continued to use TFS.

`git-tfs` is a client-side command-line tool, inspired by `git-svn`. `git-tfs` provides a two-way bridge between a local Git repository and a TFS server. `git-tfs` allows you to do your local development in a Git repository, and still synchronize your work with a TFS server.

In this article, I'll walk through some of the things that you can do with git-tfs. I assume that you use or are familiar with TFS source control.

Installing git-tfs

Before you start, you'll need to install the Team Foundation client tools (Team Explorer). You probably already have this installed if you're working with source code in TFS. git-tfs works with the VS2008, VS2010, and VS2012 versions of the Team Foundation client tools.

If you use the chocolatey package manager, you can simply run `cinst gittfs`. This takes care of downloading git-tfs and its dependencies, and making them available at a command prompt.

You can also download the latest release as a ZIP file. The prerequisites for `git-tfs` are the .NET 4 Framework and Git (msysgit is recommended). Download the latest release of `git-tfs` from git-tfs.com. Extract the ZIP file, and add the resulting directory to your `PATH` environment variable. Verify that git-tfs is installed correctly by opening a new command prompt and type `git tfs --version`.

The source code is available on GitHub.

Clone

Start by cloning your TFS project into a Git repository. You'll need to know the TFS project collection URL (if your server is TFS 2010 or later) or the TFS server URL (if your server is TFS 2008 or 2005). Git-tfs also needs to know the path in TFS that you want to clone. It can clone any path (other than the root), so you can clone an entire TFS project, or just one subdirectory of it. Use this command to create a Git repository in `local-dir`.

```
> git tfs clone https://your-server/tfs/collection $/Project/Path
local-dir
```

The new clone includes the full history of `$/Project/Path`.

If you have a large TFS repository, a full clone will take a long time to create. To get started more quickly, you can create a Git repository with a snapshot of the latest version from TFS.

```
> git tfs quick-clone https://your-server/tfs/collection
$/Project/Path local-dir
```

Both `clone` and `quick-clone` produce a normal Git repository. Run `git log` to see the imported commits. While you're developing in this repository, use normal Git tools to commit, branch, etc.

Fetching new changesets

As other developers check in new source code to TFS, you'll want to fetch the changes.

```
> git tfs fetch
```

The above command will fetch changes for the current TFS server. It stores all TFS changes on a branch named `tfs/default`. You can merge the changes from TFS with `git merge tfs/default`.

To fetch and merge all in one command, run

```
> git tfs pull
```

Share using a shelveset

If this is the first time that you're trying out git-tfs, make a commit or two in your Git repository and try creating a TFS shelveset from git-tfs.

```
> git tfs shelve this-comes-from-git-tfs
```

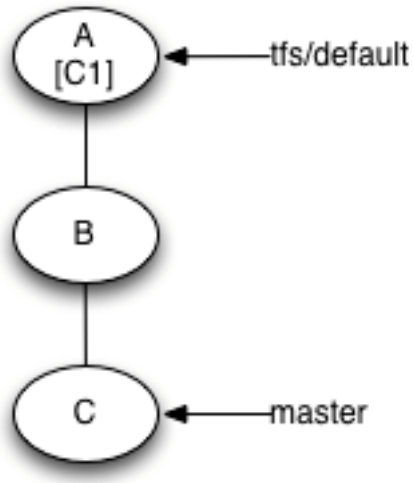
This tells git-tfs to collect the changes you've made into one shelveset. You can view the contents of the shelveset using the normal TFS tools.

This is an easy way to get started with git-tfs, because it doesn't create a permanent record (i.e. a changeset), but you can still get a good idea of what it is doing.

Share a single changeset

Git-tfs includes two ways to convert Git commits into TFS changesets. The first is `git tfs checkin`. This command will create a single TFS changeset with a squashed diff of all of your local Git commits.

For example, consider a repository with one commit (A) that git-tfs created from a TFS changeset (C1), and two commits (B and C) created locally with Git.



A(C1) -- B -- C

To check in the cumulative diff between A and C:

```
> git tfs checkin -m "made some changes"
```

Git-tfs can include checkin meta data. For example, to add a work item reference, use the `-w` flag.

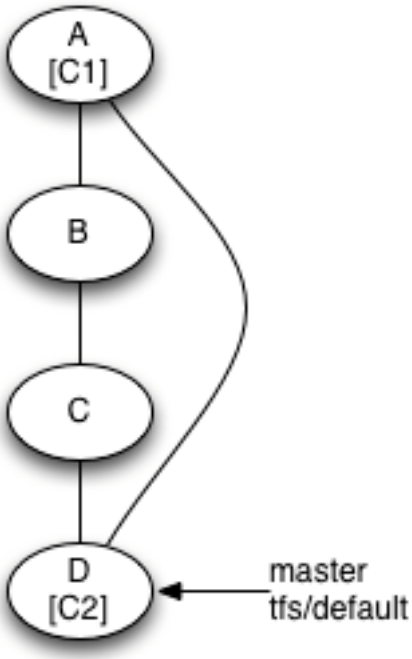
```
> git tfs checkin -w 1234 -m "made some changes"
> git tfs checkin -w 1234:associate -m "made some changes"
```

More options can be found by running git tfs help checkin.

To use TFS's graphical checkin dialog:

```
> git tfs checkintool
```

The `checkin` or `checkintool` command will add a TFS changeset (C2) and a corresponding Git commit (D).

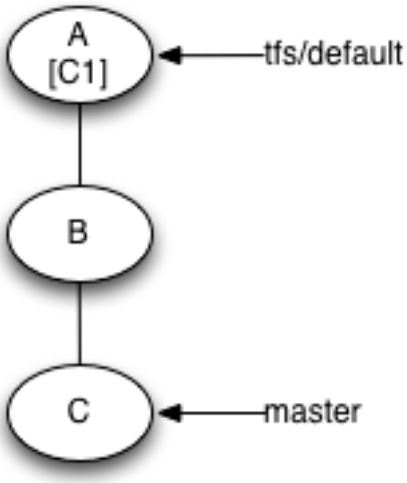


A(C1) -- B -- C -- D(C2)

Share all your commits

The second way to convert Git commits into TFS changesets is `git tfs rcheckin`. This command will create a TFS changeset for each Git commit that you have made locally.

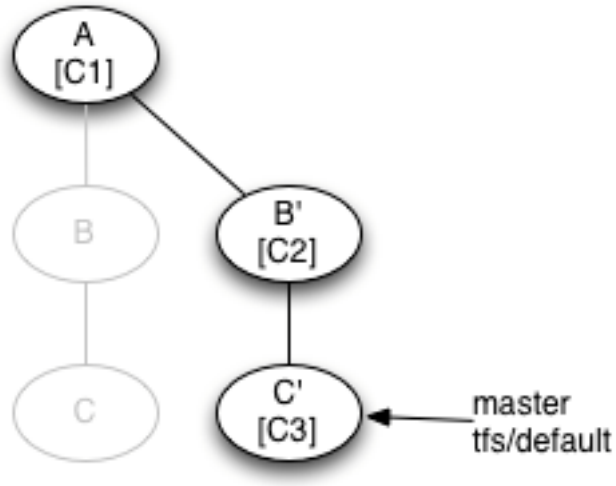
Git-tfs creates the checkins in a single command invocation, so it relies on information in the commits' messages for other commit metadata. The Git commit message becomes the TFS checkin comment. Other metadata can be provided on lines with certain keyword prefixes. For example, a line that starts with `git-tfs-work-item:` specifies the work item number to link to the new TFS changeset. See the git-tfs wiki for an example and for a list of other fields that can be provided.



A(C1) -- B -- C

Given the previous example of one TFS commit and two local Git commits, you can run this.
```
git tfs rcheckin
```

Git-tfs creates TFS checkins from your local commits, and updates your local branch to point to these new commits.



A(C1) -- B(C2) -- C(C3)

TFS branches

TFS source control has branching support. In TFS, a branch looks like another folder in your project. Git-tfs includes the ability to map TFS branches to Git branches. The easiest way to get started with this is to clone the root TFS branch, along with the option that tells git-tfs to retrieve all child branches:
```
> git tfs clone --with-branches https://tfs/collection $/project
local-dir
```

The repository that git-tfs will create is still a normal Git repository, and you should be able to interact with TFS in the same way that you have before, with the exception that you will need to tell git-tfs which branch you're working with more of the time. For example, to fetch changesets for all of the TFS branches:
```
> git tfs fetch --all
```

To fetch changes just for one branch:
```
> git tfs fetch -i yourbranch
```

To rcheckin or checkin just one branch:
```
> git tfs rcheckin -i yourbranch
> git tfs checkin -i yourbranch
```

Sharing a git-tfs repository

If Git starts to catch on with your team, you may find yourself sharing a repository that you created with git-tfs. When you do this, your teammates may also need to use git-tfs to interact with TFS. To enable this in a cloned Git repository, git-tfs can look through the history of the HEAD branch to find TFS configuration information.
```
> git tfs bootstrap
```

If bootstrap fails to configure the TFS connection correctly, you can configure it manually with `git tfs init`. You can provide the same options to `git tfs init` that you would provide to `git tfs clone`. For example:

```
> git tfs init http://server/collection $/project/folder
```

Conclusion

If you're using TFS and are considering Git, git-tfs can help you get started without a wholesale conversion. If part of your team wants to collaborate with each other using Git, git-tfs can provide a link back to the rest of the team that is using TFS. If your entire team wants to move to Git, but you still have tooling or processes that require the use of TFS, git-tfs can provide the link. And if you end up switching completely to Git, git-tfs makes the transition significantly simpler.

Apache Isis - Developing Domain-driven Java Apps

Dan Haywood dan@haywood-associates.co.uk
Haywood Associates Ltd, http://danhaywood.com

Apache Isis software is a framework for rapidly developing domain-driven apps in Java. Write your business logic in entities, domain services and repositories, and the framework dynamically generates a representation of that domain model as a webapp or a RESTful API. For prototyping or production. Apache Isis and Isis are trademarks of The Apache Software Foundation.

Website: http://isis.apache.org
Version presented: Isis Core v1.2.0 (with v1.0.3 quickstart archetype)
System requirements: Java, 1Gb memory (Windows, Mac or Linux)
License & pricing: Open source, licensed under Apache License, v2.0.
Support: Mailing lists, see http://isis.apache.org/support.html

To stop himself from procrastinating in his work, the Greek orator Demosthenes would shave off half his beard. Too embarrassed to go outside and with nothing else to do, his work got done.We could learn a lesson or two from old Demosthenes. After all, we're forever taking an old concept and inventing a new technology around it (always with a new acronym, of course); anything to stop getting down to the real work of solving business problems.

Domain-driven design (hereafter DDD) puts the emphasis elsewhere, “tackling complexity in the heart of software”. And Apache Isis - an open source Java framework at the Apache Software Foundation – helps you build your business applications with ease. No beard shaving necessary.

In this article we’re going to quickly review why a domain-driven approach is important, and then we’ll see how Isis supports those principles, and how to rapidly build apps either for prototyping or production.

Understanding Domain-Driven Design

We developers love the challenge of understanding and deploying complex technologies. But understanding the nuances and subtleties of the business domain itself is as great a challenge; more so. Putting our efforts into understanding the domain and addressing its subtleties leads to clean, maintainable software, that does a better job for our stakeholders. Our stakeholders would probably thank us for that.There are two central ideas at the heart of domain-driven design:

- The *ubiquitous language* is about the whole team (both domain experts and developers) communicating transparently using a domain model.

Ubiquitous Language

Build a common language between the domain experts and developers by using the concepts of the domain model as the primary means of communication. Use those terms in speech, in diagrams, in writing, when presenting.

If an idea cannot be expressed using this set of concepts, go back and extend the model. Look for and remove ambiguities and inconsistencies.

- *Model-driven design* is about capturing that model in a very straightforward manner in code.

Model-Driven Design

Use a straightforward and very literal way to represent the domain model in terms of software. Changing the code means changing the mode; refining the model requires a change to the code.

But how can a software framework help? Apache Isis does so by adopting the *naked objects* (NO) architectural pattern. The fundamental idea of NO is to infer both the structure and behaviour of the domain objects from the code, allowing the end-user to interact with the domain objects directly through a user interface generated dynamically (at runtime) from those domain objects. Let me emphasise the "and behaviour" bit here. The naked objects pattern is sometimes characterized merely as a means to build CRUD applications. And while you can indeed build CRUD apps with an NO framework such as Isis, the real value comes from the fact that domain object behaviour is also exposed by the framework.

For example, suppose you work for an estate management company where the lease agreements with tenants can be based on a multitude of factors. For index-linked leases, an invoice must be sent out each quarter and an adjustment must be made retrospectively when an index rate is updated. The non-trivial logic to recalculate and reissue invoices can all be exposed as behaviour on the business object. Moreover, the different calculations required for other lease types can be invoked in the same way: polymorphic operations, and at the same abstraction level that the domain expert thinks!

Another reason why you should consider using Isis to build your DDD app – perhaps the killer reason – is the speed of development that it allows. To build an application in Isis means writing the classes in your domain model, but it does *not* require you to write any UI logic. No controllers, no view models, no HTML pages with markup, no DTOs… nothing. You can build stuff very rapidly indeed. More subtly, because it doesn't cost much in terms of time to build the app, you're much more likely to experiment and try out other designs. Each time you iterate you'll gain greater insights into the business domain, and all the time you'll be extending and enriching your ubiquitous language.

How does Apache Isis compare to other frameworks?

Many other frameworks promise rapid application development, so how do they compare with Apache Isis?

For many developers, the most obvious difference of Apache Isis is the absence of controllers or views. Instead, both state and behaviour is automatically exposed in its generic object-oriented UI.

This UI can then be customized in various ways. Isis supports a number of different viewers; the Wicket viewer (based on Apache Wicket™ [2]) allows components to be registered that will render objects in a calendar view, or in a google map, or as a chart, or whatever.

Another viewer is the RestfulObjects viewer which exposes a full REST API from the domain object model. Using this API you can if you wish hand-craft your own UI, with Isis taking care of the entire back-end.

Jolly good. But I suspect by now you'll want to see some code. Let's see how to build an application in Isis.

Yet another "ToDo" app…

While the "ToDo" app seems to have replaced "Hello World", we make no excuses for using this domain as the basis for Isis' quickstart archetype. Its lack of complexity makes it easy for you to refactor towards your own requirements.

If you go to the Isis website then you'll find instructions and screencasts describing how to run the quickstart (Wicket/Restful/JDO) archetype [3]. All the example code and discussion that follows relates to that app. Do run the archetype yourself and follow along.

Domain Services vs. Domain Entities

The quickstart application consists of just two domain classes. The `ToDoItem` entity is obvious enough – it represents a single item on your todo list – while the `ToDoItems` class is a domain service acting as both repository (to lookup existing `ToDoItem`s) and as a factory (to create new `ToDoItem`s).

To see how Isis automatically builds a UI from the code, look at the signature of the methods in the `ToDoItems` service in listing 1.

```
@Named("ToDos")
public class ToDoItems {
 ...
 @MemberOrder(sequence = "1")
 public List<ToDoItem> notYetComplete() { ... }

 @MemberOrder(sequence = "2")
 public List<ToDoItem> complete() { ... }

 @MemberOrder(sequence = "3")
 public ToDoItem newToDo(
   @RegEx(validation = "\\w[@&:\\-\\,\\.\\+ \\w]*")
   @Named("Description") String description,
   @Named("Category") Category category,
   @Optional @Named("Due by") LocalDate dueBy,
   @Optional @Named("Cost") BigDecimal cost ) { ... }

 @MemberOrder(sequence = "4")
 public List<ToDoItem> allToDos() { ... }
 ...
}
```

Listing 1: ToDoItems domain service's methods

And in the user interface this is rendered as shown in figure 1:

Figure 1: ToDoItems domain service's methods rendered in the UI as actions

The `newToDo()` method in listing 1 takes parameters. If its menu item is selected then Isis will, as figure 2 shows, render a page prompting for argument values.

Figure 2: ToDoItems newToDoItems action takes arguments

There's enough information within the `ToDoItems` class for Isis to build this UI; each method in the class corresponds to an action in the UI. The menu item names are inferred directly from the method names, while annotations – such as `@MemberOrder` – provide additional rendering hints. Similarly the `@Named` annotation is used in the menu ("TODOS"); otherwise the menu label would be derived from the class name.

We sometimes hear complaints that a programming model heavy with annotations pollutes the domain model with UI concerns. However, the way that Isis builds up its internal metamodel is completely extensible; you could if you wished use some other mechanism, for example reading some external XML file.

Domain Service Implementations

So far we've only discussed the method signatures of a domain service, so let's now look at the implementations. Listing 2 shows the `newToDo()` method/action (with some helper methods inlined to simplify the discussion).

```
public class ToDoItems {
 ...
 @MemberOrder(sequence = "3")
 public ToDoItem newToDo(
   ... String description,
   ... Category category,
   ... LocalDate dueBy,
   ... BigDecimal cost) {
  final String ownedBy = container.getUser().getName();
  final ToDoItem toDoItem =
     container.newTransientInstance(ToDoItem.class);
  toDoItem.setDescription(description);
  toDoItem.setCategory(category);
  toDoItem.setOwnedBy(userName);
  toDoItem.setDueBy(dueBy);
  toDoItem.setCost(cost);

  container.persist(toDoItem);
  return toDoItem;
 }
 ...
}
```

Listing 2: ToDoItems newToDo() method

Note the `container` field, used to obtain the current user, to instantiate and finally to persist the `ToDoItem`. The `container` is of type `DomainObjectContainer`, defined in Isis' application library, and represents the domain objects' only hard dependency upon the framework. Even then, `DomainObjectContainer` is an interface, easily mocked out during unit testing.

Persistent agnostic entities

In Isis entities are perhaps best described as "persistent agnostic" rather than "persistent ignorant"; they do know *that* they are persisted (an object might behave differently if persisted or if not), but they don't know (nor care) *how* they are persisted.

The container is injected into the domain service, as shown in listing 3.

```
public class ToDoItems {
 ...
 private DomainObjectContainer container;
 public void setContainer(DomainObjectContainer container) {
  this.container = container;
 }
}
```

Listing 3: DomainObjectContainer injection

Moreover, because the `ToDoItem` is instantiated through the container, the container is `also` injected into the `ToDoItem` entity. In fact, all domain services are injected into domain entities automatically. `DomainObjectContainer` is really just a built-in general purpose domain service.

Extending the reach of Domain Entities

Domain services can do far more than simply finding existing objects or creating new ones. For example, they can:

- call out to other systems via web services
- publish events onto an enterprise service bus
- generate representations of themselves as word documents or PDFs,
- attach barcodes to such a PDF so that a document can be scanned
- send out emails, text messages or tweets

The (Java) interface of these is part of the domain object model, while the implementation considered part of the infrastructure. During testing, each of these interfaces can be easily mocked out, with the service implementations themselves tested independently.

Let's inspect another of `ToDoItems` methods, `notYetComplete()`,shown in listing 4.

```
public class ToDoItems {
 ...
 @ActionSemantics(Of.SAFE)
 @MemberOrder(sequence = "1")
 public List<ToDoItem> notYetComplete() {
  List<ToDoItem> items = doNotYetComplete();
  if(items.isEmpty()) {
   container.informUser(
    "All to-do items have been completed :-)");
  }
  return items;
 }
 protected List<ToDoItem> doNotYetComplete() {
  return container.allMatches(ToDoItem.class,
   new Filter<ToDoItem>() {
    @Override
    public boolean accept(final ToDoItem t) {
     return ownedByCurrentUser(t) && !t.isComplete();
    }
   });
 }
}
```

Listing 4: The ToDoItems repository's implementation of the notYetComplete() action method

Once more the method delegates to the `DomainObjectContainer`, and (in the `doNotYetComplete()` hook method) calls its `allMatches()` method. This returns all instances of the specified type meeting some criteria. We call this a "naïve" implementation of the repository because, although easy to write, this isn't code that would scale to large volumes; the filtering is done client-side. We trade off the speed of development against performance.

As your domain model starts to stabilize though, you'll want to replace this naïve implementation with one that *will* scale. Apache Isis has a pluggable persistence layer, and the implementation most commonly used for production systems is the JDO objectstore with DataNucleus [4] as the underlying implementation. Although JDO isn't as well known as JPA, DataNucleus is a very capable ORM; JDO is also used within Google App Engine [5].

Thus, the `ToDoItemsJdo` subclasses `ToDoItems` and overrides the repository hook methods with implementations that delegate the querying to the database. Listing 5 shows the corresponding method, with supporting annotations appearing on the `ToDoItem` entity, in listing 6.

```
public class ToDoItemsJdo {
 ...
 protected List<ToDoItem> doNotYetComplete() {
  return container.allMatches(
    new QueryDefault<ToDoItem>(ToDoItem.class,
      "todo_notYetComplete",
      "ownedBy", currentUserName()));
 }
}
```

Listing 5: The JDO-specific ToDoItemsJdo's implementation of doNotYetComplete() action method

```
@PersistenceCapable(identityType=IdentityType.DATASTORE)
@Queries( {
 @Query(
  name="todo_notYetComplete", language="JDOQL",
  value="SELECT FROM dom.todo.ToDoItem "+
    "WHERE ownedBy == :ownedBy && complete == false")
} ... )
public class ToDoItem {
 ...
}
```

Listing 6: The JDO annotations on ToDoItem entity

The domain services are registered to Isis through the `WEB-INF/isis.properties` file, using the `isis.services` key. This can be seen in listing 7.

```
isis.services = objstore.jdo.todo.ToDoItemsJdo,\
    fixture.todo.ToDoItemsFixturesService
```

Listing 7: The JDO annotations on ToDoItem entity

As you can see, there are in fact two services registered; `ToDoItemsJdo` (already discussed), and with `ToDoItemsFixtureService`. Each is represented as a menu item in figure 1.

Optional Services

In fact, there are rather more than two services registered in `isis.properties`; the two discussed in the main text are the only ones that have actions that are visible in the UI.

The other services implement optional SPIs for the framework: auditing, the automatic publication of events, and strategies for recognizing and handling exceptions. Some of these capabilities are very powerful, but are, unfortunately, out of scope of this article.

The `ToDoItemsFixtureService` service provides a convenient way to install sample ("fixture") data, for prototyping with users or for testing. This service isn't intended to be deployed to production, but when prototyping we usually use a non-persistent data configuration (such as the JDO objectstore configured against an in-memory HSQLDB database). The service helps us quickly set up some realistic data.

Time now to turn our attention to domain entities.

Domain Entities

While domain services only expose behaviour through actions, domain entities also hold state: properties and collections.

In terms of code, think of a domain entity as a javabean/pojo "on steroids". It uses the getters and setters to identify the properties and collections, with any remaining public methods taken to be actions of the entity.

For example, listing 9 sketches out the properties of the `ToDoItem` class:

```
public class ToDoItem ... {
 public static enum Category { Professional, Domestic, Other }

 @RegEx(validation = "\\w[@&:\\-\\,\\.\\+ \\w]*")
 @MemberOrder(sequence = "1")
 public String getDescription() { ... }
 public void setDescription( ... ) { ... }

 @javax.jdo.annotations.Persistent
 @MemberOrder(name="Detail", sequence = "3")
 @Optional
 public LocalDate getDueBy() { ... }
 public void setDueBy( ... ) { ... }

 @MemberOrder(sequence = "2")
 public Category getCategory() { ... }
 public void setCategory( ... ) { ... }

 @Hidden
 public String getOwnedBy() { ... }
 public void setOwnedBy( ... ) { ... }

 @Disabled
 @MemberOrder(sequence = "4")
 public boolean isComplete() { ... }
 public void setComplete( ... ) { ... }
```

```
 @javax.jdo.annotations.Column(scale = 2)
 @Optional
 @MemberOrder(sequence = "4.1")
 public BigDecimal getCost() { ... }
 public void setCost(final BigDecimal cost) { ... }

 @Hidden(where=Where.ALL_TABLES)
 @Optional
 @MultiLine(numberOfLines=5)
 @MemberOrder(name="Detail", sequence = "6")
 public String getNotes() { ... }
 public void setNotes( ... ) { ... }

 @javax.jdo.annotations.Persistent(defaultFetchGroup="false")
 @Optional
 @MemberOrder(name="Detail", sequence = "7")
 @Hidden(where=Where.STANDALONE_TABLES)
 public Blob getAttachment() { ... }
 public void setAttachment( ... ) { ... }

 @Hidden(where=Where.ALL_TABLES)
 @Disabled
 @MemberOrder(name="Detail", sequence = "99")
 @Named("Version")
 public Long getVersionSequence() { ... }
}
```

Listing 9: The ToDoItem entity's properties

If you invoke one of the `ToDoItems` domain service actions such as `notYetComplete()`, then a list of matching `ToDoItem` instances are rendered in a table. This is shown in figure 3.

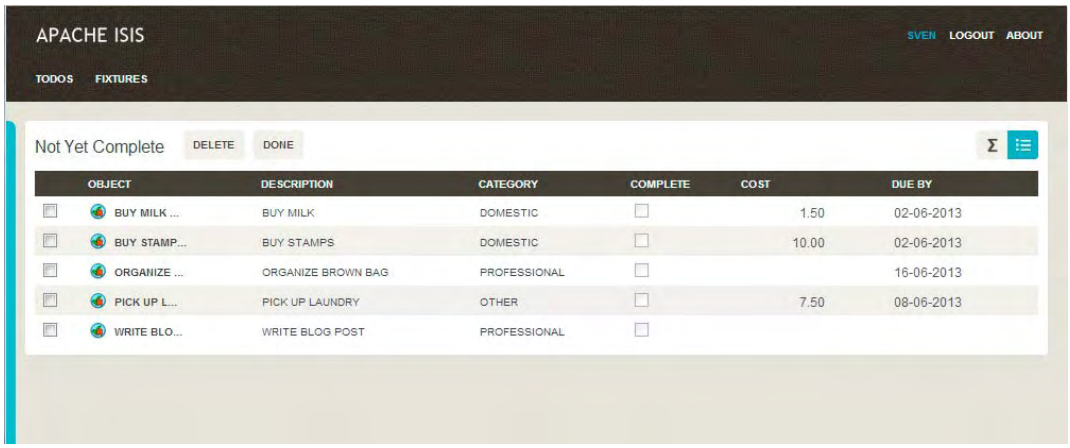


Figure3: A collection of ToDoItems is automatically shown in a table

Comparing this to the code, you can see that the table automatically renders each property appropriately (for example, a checkbox for the `complete` property, right aligned for the cost). Not every property is shown though; those that are annotated as `@Hidden` or `@Hidden(where=..._TABLES)` are excluded. Visibility is just one of three business rules that Isis supports on class members, each of which can be specified declaratively (as here) or imperatively. We'll see examples of the other business rules and of imperative business rules shortly.

Another thing you may have noticed is that every entity is identified by a title and an icon (truncated for space in the table view). The title itself is specified either using the `@Title` annotation, or, as in this case, using a `title()` method. This is shown in listing 10.

```
public String title() {
 final TitleBuffer buf = new TitleBuffer();
 buf.append(getDescription());
 if (isComplete()) {
  buf.append(" - Completed!");
 } else {
  if (getDueBy() != null) {
   buf.append(" due by ", getDueBy());
  }
 }
 return buf.toString();
}
```

Listing 10: The title of the ToDoItem is provided by the title() method

It's possible to specify the icon image using a similar mechanism, but more commonly this is done just by naming convention. In the various screenshots the icon being rendered is in a file `ToDoItem.png`.

Let's get back to looking at the rest of the `ToDoItem` entity though. Listing 11 shows (an outline of) the collections (the remaining getters) and actions that make up `ToDoItem`.

```
public class ToDoItem ... {

 @javax.jdo.annotations.Persistent(table="TODO_DEPENDENCIES")
 @javax.jdo.annotations.Join(column="DEPENDING_TODO_ID")
 @javax.jdo.annotations.Element(column="DEPENDENT_TODO_ID")
 @Disabled
 @MemberOrder(sequence = "1")
 @Render(Type.EAGERLY)
 public SortedSet<ToDoItem> getDependencies() { ... }
 public void setDependencies( ... ) { ... }

 @MemberOrder(sequence = "5")
 @NotPersisted
 @Render(Type.LAZILY)
 public List<ToDoItem> getSimilarItems() { ... }

 @Named("Done")
 @PublishedAction
 @Bulk
 @MemberOrder(name="complete", sequence = "1")
 public ToDoItem completed() { ... }

 @Named("Undo")
 @PublishedAction
 @MemberOrder(name="complete", sequence = "2")
 public ToDoItem notYetCompleted() { ... }

 @Named("Update")
 @MemberOrder(name="cost", sequence = "1")
 public ToDoItem updateCost( ... ) { ... }
```

```
 @PublishedAction
 @MemberOrder(name="dependencies", sequence = "3")
 public ToDoItem add( ... ) { ... }

 @MemberOrder(name="dependencies", sequence = "4")
 public ToDoItem remove( ... ) { ... }

 @Named("Clone")
 @MemberOrder(sequence = "3")
 public ToDoItem duplicate( ... ) { ... }

 @Bulk
 @MemberOrder(sequence = "4")
 public List<ToDoItem> delete() { ... }
}
```

Listing 11: The ToDoItem entity's collections and actions

Clicking on any of the links in the table (figure 3) will take us to a page showing a single entity (figure 4).

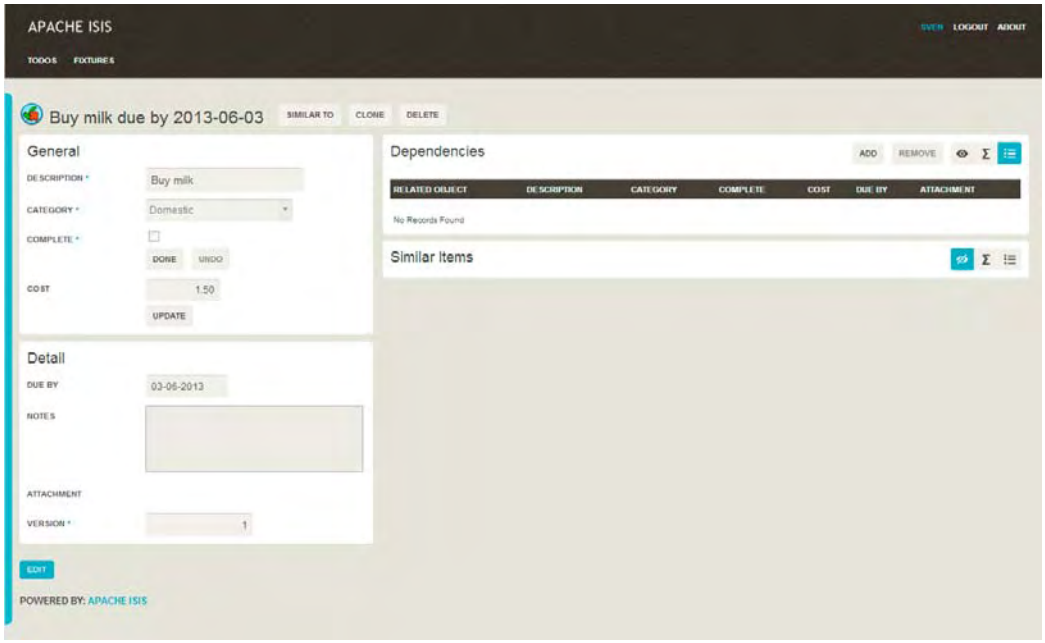


Figure 4: A single ToDoItem entity, as rendered in Isis

On the left hand side are ToDoItem's properties, on the right are collections, throughout are buttons for the actions. As previously, the `@MemberOrder` annotation specifies the relative order of class members; it can also be used to group them together ("General", "Detail"). `@MemberOrder` is also used to associate actions with properties or collection; thus the `done` and `undo` actions are shown by the `complete` property, and the `add` and `remove` actions are shown within the `dependencies` collection.

On the `notes` property, the `@MultiLine` annotation causes the property to be rendered using a textbox. The `category` property meanwhile is rendered as a dropdown, its values taken from the `Category` enum. Pressing "Edit" allows the user to update the properties of the entity. Non-mandatory properties are annotated with `@Optional`. Those annotated `@Disabled` – such as the `complete` property - remain read-only. Derived properties also cannot be edited; an example is the `version` property.

After visibility/invisibility, enablement/disablement is the second type of business rules we can apply to class members.

Finally, you'll see that the `description` property is annotated with `@Regex`, a regular expression. This is one of a handful of annotations that can validate a property making sure its value is correct; another commonly used one is `@MaxLength`.

Validation is the third of the business rules that Isis lets us apply to class members.

In all we therefore have three types of rules: visibility rules, enablement rules and validation rules. Or, if you prefer: think of them as the "see it, use it, do it" rules.

Declarative and also Imperative Business Rules

As well as using annotations; we can also specify business rules imperatively, using a number of supporting methods.

For example, listing 12 fleshes out the `dueBy` property.

```
private LocalDate dueBy;
@javax.jdo.annotations.Persistent
@MemberOrder(name="Detail", sequence = "3")
@Optional
public LocalDate getDueBy() {
 return dueBy;
}
public void setDueBy(final LocalDate dueBy) {
 this.dueBy = dueBy;
}
public void clearDueBy() {
 setDueBy(null);
}
public String validateDueBy(final LocalDate dueBy) {
 if (dueBy == null) {
  return null;
 }
 return isMoreThanOneWeekInPast(dueBy) ?
  "Due by date cannot be more than one week old" : null;
}
```

Listing 12: Supporting methods for the ToDoItem entity's dueBy property

Both the `validateDueBy()` and the `clearDueBy()` are supporting methods for the `dueBy` property; Isis matches them by naming convention. In the case of `validateDueBy()`, this is called whenever the user enters a new value, prior to actually calling the setter. If the method returns a non-null string then (as shown in figure 5) this is used as the error message in the UI.

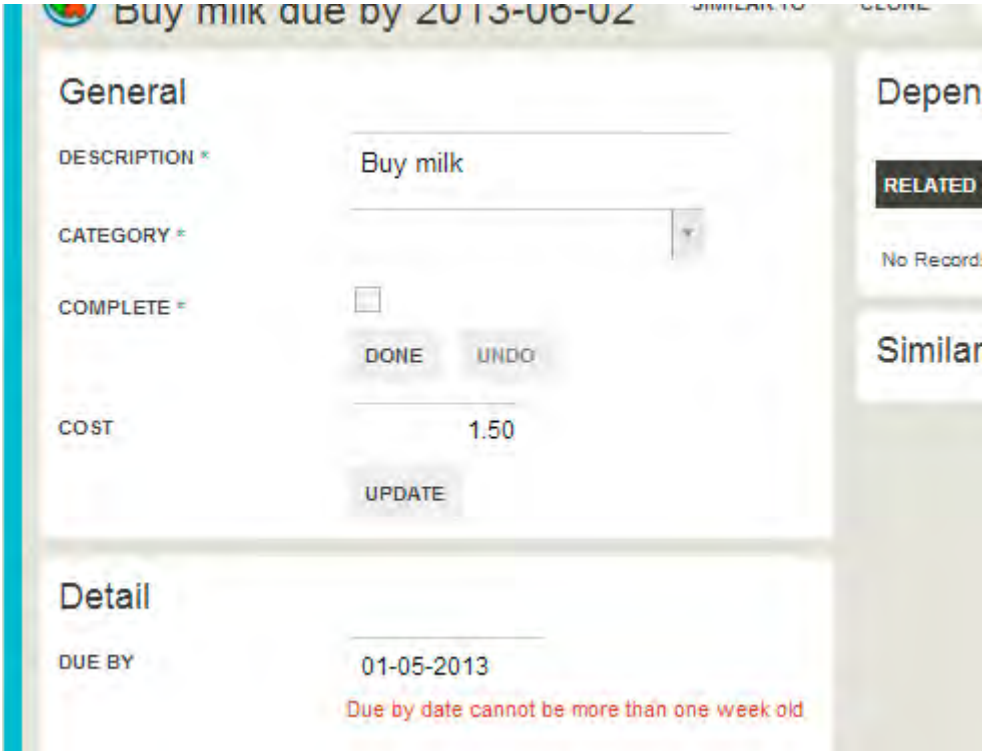


Figure 5: Isis renders validation error messages

Another example of a supporting method, this time to disable an action, can be seen in the `completed` action, per listing 13.

```
@Named("Done")
@PublishedAction
@Bulk
@MemberOrder(name="complete", sequence = "1")
public ToDoItem completed() {
 setComplete(true);
 return this;
}
public String disableCompleted() {
 return complete ? "Already completed" : null;
}
```

Listing 13: Supporting methods for the ToDoItem entity's completed action

As we see in figure 6, this `disable` method causes the action's button to be greyed out for a `ToDoItem` where its `complete` property is true.

Figure 6: Isis automatically disables the completed action

You might also have noticed the `@Bulk` annotation, which can be put onto any no-arg action, allowing that action to be invoked against all selected objects at once. If you look back you'll notice the "Done" button above the table in figure 3. The `@PublishedAction`, meanwhile, is in support of Isis' `PublishingService` (out of scope for this article, I'm afraid).

Subtractive Programming

In other frameworks the functionality of the app must be built up piece by piece. With Isis it is to some extent the other way around: defining the domain class structure gives us basic CRUD behaviour, and then we apply the "see it, use it, do it" rules to *subtract* functionality.

On top of this basic CRUD behaviour we then identify and implement the domain object actions that provide the main value-added business logic. But we can get to that point very quickly, and won't have exhausted ourselves writing reams of boilerplate in the meantime.

Usability

As well as implementing business rules, supporting methods also make the application more usable, providing defaults for arguments when invoking actions , and/or in providing a selection of choices.

For example, figure 7 shows this in the case of the `ToDoItems`' `newToDo` action.

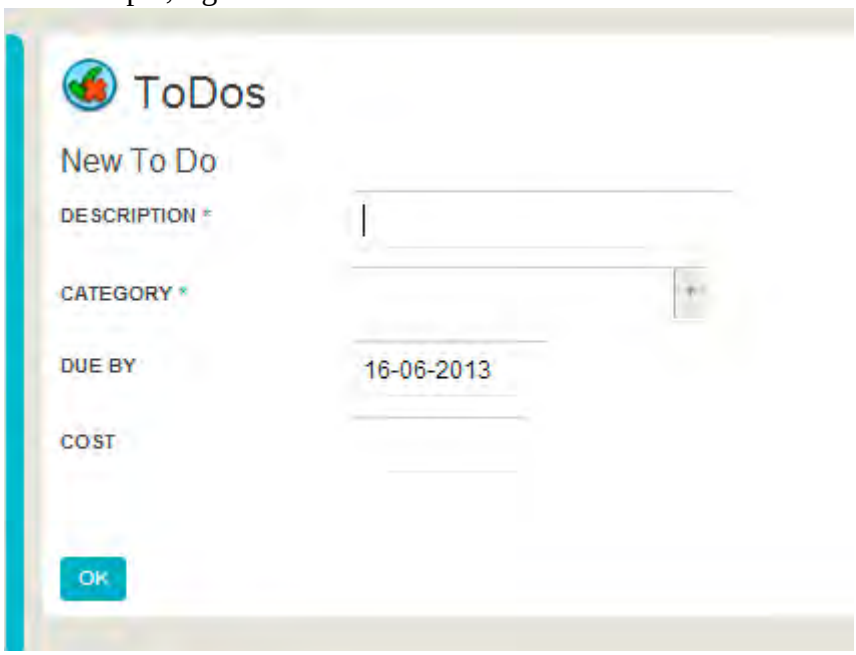


Figure 7: Defaults are provided for the clone action

Listing 14 shows the corresponding code. The default for the dueBy parameters is provided in the supporting `default2NewToDo()`method:

```
public ToDoItem newToDo(
  ... String description,
  ... Category category,
  ... LocalDate dueBy,
  ... BigDecimal cost) {
 ...
}
public LocalDate default2NewToDo() {
 return new LocalDate(Clock.getTime()).plusDays(14);
}
```

Listing 14: The supporting methods for the clone action that provide the defaults

Let's look at the choices supporting method, as seen in the `remove` action. If dependencies have been added to a `ToDoItem`, it makes sense only to offer those dependencies for removal. This is shown in figure 8.

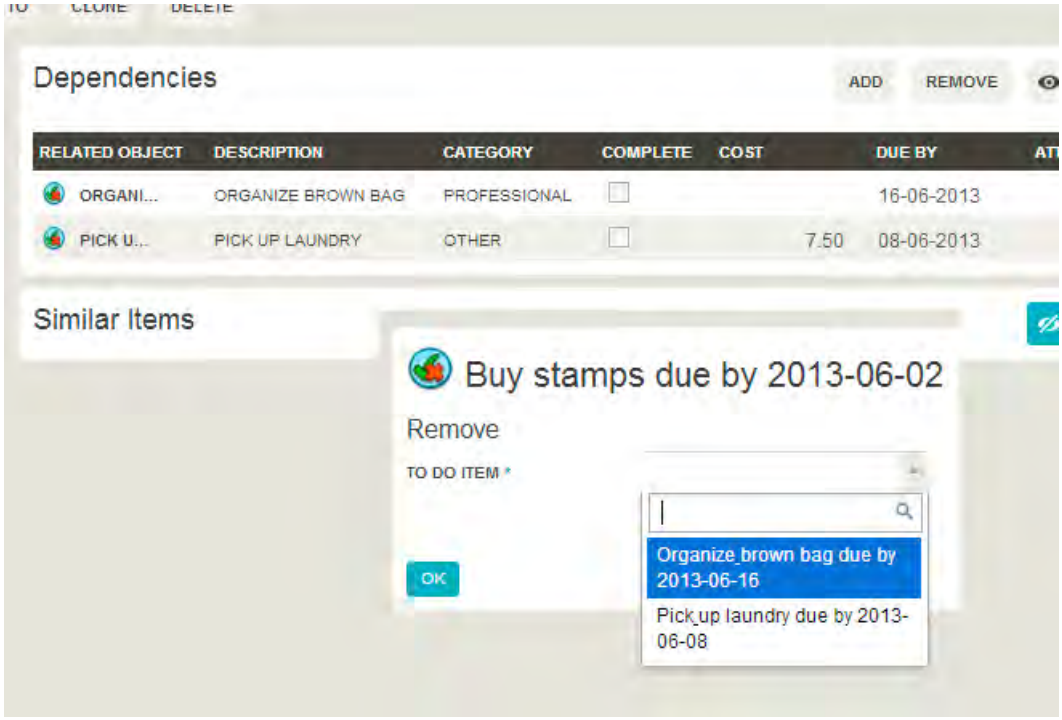


Figure 8: Choices are provided for the remove action

The supporting methods in this case are shown in listing 15 (there are also examples of `disableXxx()` and `validateXxx()` methods).

```
public ToDoItem remove(final ToDoItem toDoItem) {
 getDependencies().remove(toDoItem);
 return this;
}
public String disableRemove(final ToDoItem toDoItem) {
 return getDependencies().isEmpty()?
  "No dependencies to remove": null;
}
public String validateRemove(final ToDoItem toDoItem) {
 if(!getDependencies().contains(toDoItem)) {
  return "Not a dependency";
 }
 return null;
}
public List<ToDoItem> choices0Remove() {
 return Lists.newArrayList(getDependencies());
}
```

Listing 15: The supporting methods for the remove action

We haven't actually looked at the `ToDoItem`'s `add` action method, yet. The Java method for it is simple enough, as shown in listing 16.

```
public ToDoItem add(final ToDoItem toDoItem) {
 getDependencies().add(toDoItem);
 return this;
}
public String validateAdd(final ToDoItem toDoItem) {
 if(getDependencies().contains(toDoItem)) {
  return "Already a dependency";
```

```
  }
  if(toDoItem == this) {
   return "Can't set up a dependency to self";
  }
  return null;
 }
```

Listing 16: The ToDoItem add action method takes a reference to another entity

But the way that Isis renders this is interesting because, as shown in figure 9, the user can specify the `ToDoItem` for the action by just typing its title.

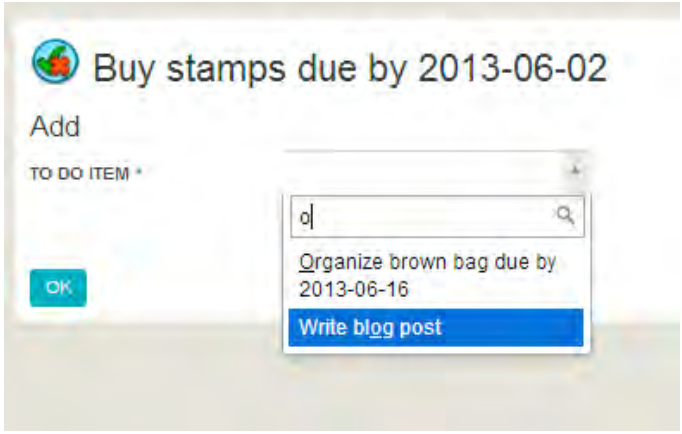


Figure 9: The add action's parameters can be searched for using autocomplete

Isis provides this because the `ToDoItem` class is annotated with `@AutoComplete` annotation specifying an action on the `ToDoItems` repository/domain service. This can be seen in listing 17.

```
@AutoComplete(repository=ToDoItems.class, action="autoComplete")
public class ToDoItem ... {
 ...
}

public class ToDoItemsJdo {
 ...
 @Hidden
 public List<ToDoItem> autoComplete(final String description) {
  return ...
 }
}
```

Listing 17: Autocomplete is provided through an annotation and a method

With all of these supporting methods, if a class member were renamed then there's a potential risk that its supporting method might not be. To protected against this, Isis validates the metamodel that it builds up internally; any orphaned methods are flagged when Isis starts. This metamodel validation is also extensible; if you want to register your own checks (perhaps to enforce some project-specific conventions), then you can.

Customizing the User Interface

While the UI provided by Isis is completely generic, it is still, we think, reasonably usable. But there will be times when you want to extend it.

Isis' Wicket viewer (as shown in the screenshots), provides an extensible API allowing different renderings of any of the components on the UI, leveraging Apache Wicket's own `Component` interface. For example, one extension [6] automatically renders entities on a map, as per figure 10.

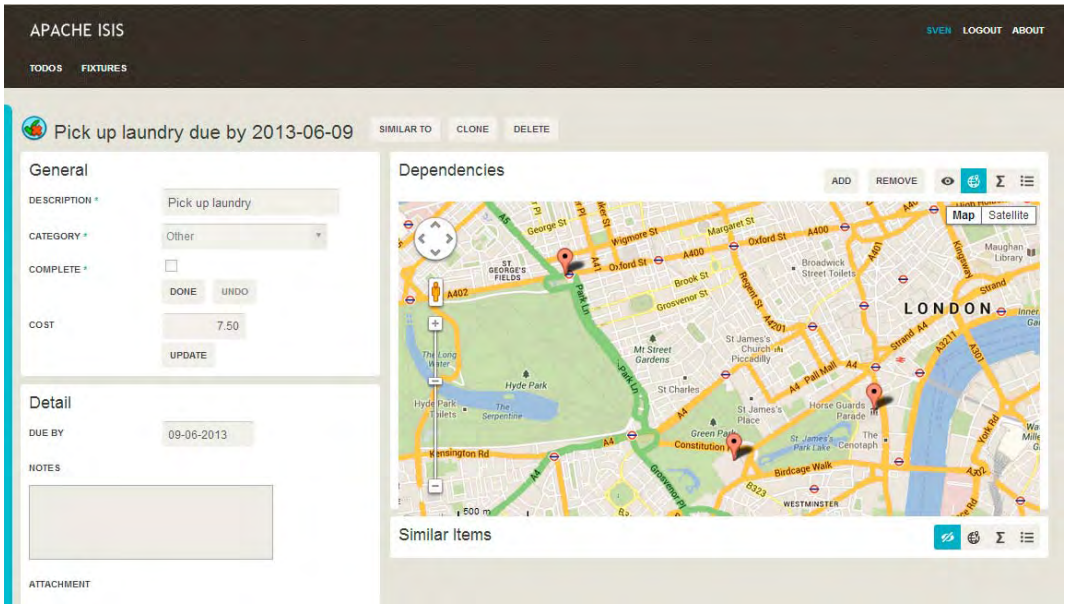


Figure 10: Entities can be rendered within a Google map

Supporting this is a matter of implementing a `Locatable` interface (listing 18).

```
public class ToDoItem implements Locatable {
 ...
 @javax.jdo.annotations.Persistent
 private Location location;
 @MemberOrder(name="Detail", sequence = "10")
 @Optional
 public Location getLocation() {
  return location;
 }
 public void setLocation(Location location) {
  this.location = location;
 }
}
```

Listing 18: Implement Locatable to support being rendered on a map

Other extensions to support graphs, calendars and so on are just as easily supported.

If you want to go further, then you can consider some of the other viewers that Isis provides. One in particular – the RestfulObjects viewer – automatically provides a complete RESTful API of your domain objects, exposing both their state *and* their behaviour/business rules. Using this you are free to build any bespoke user interface that you wish.

Do remember though that there's a huge maintenance cost with bespoke UIs; as you've seen, Isis can generate a reasonably good user interface without introducing that maintenance cost. To decide whether a bespoke UI is justified, we can distinguish between users that are "problem solvers" and those that are "process followers".

Problem solvers are users with a (reasonably) deep understanding of the domain, and need a system that allows them to work in whichever way makes sense. The domain objects enforce business logic (don't allow themselves to be put into an invalid state), but the system doesn't prescribe how it is used. Many enterprise apps fall into this category. Generic UIs provided by Isis support this category of users very well.

Process followers, meanwhile, are users that follow a more narrowly defined process. This may be because they are inexperienced with the domain, or it may be that the process is high volume, simple, and needs optimization. These are the situations when a customized user interface should be built. In this case the RestfulObjects viewer is a great way to provide the backend plumbing, while you can choose your favourite UI technology for the front-end.

The dangers of custom UIs

While there are good and valid reasons for writing a custom UI to your app you should try to defer that step to as late as possible. Jumping too prematurely to a custom UI means that insights into the domain model can get lost. We should *not* be wasting time tweaking fonts; there are more important discussions to be had.

But the more obvious danger of a custom UI is that we start writing business logic in the presentation layer. What starts as a quick check for a non-negative number soon mushrooms into a huge chunk of logic that should instead reside within a domain object model.

In contrast, Isis' generic presentation and application layers act as a kind-of firewall, ensuring that domain logic does not leak out of the domain. With Isis it's impossible to misplace domain logic because there's only one place to write your code: in the domain objects themselves!

Moving into Production

At some point you'll want to take your application into production. Isis apps are typically built using Maven, resulting in a WAR file; in principle there's not a lot to do.

You will, of course, want to fully test your application. We're out of space in this article to cover this, however you should know that Isis has a full integration testing framework (build on JUnit), allowing end-to-end tests from the UI to the domain objects through to the database. It even supports testing of annotations (such as `@RegEx`) and supporting methods (such as `disableNotYetCompleted()`). The application generated by the quickstart archetype has a full set of tests so you can check these out for yourself.

Another area to consider, we've discussed already: replacing naïve implementations of your repositories (such as the `ToDoItems` class) with objectstore-specific ones (`ToDoItemsJdo`) that will scale.

The final aspect to consider is security. Isis integrates with Apache Shiro™[7], so authentication can be handled using Shiro's own pluggable architecture. Authorization, too, can be handled by Shiro; the only thing you need know is Isis' format for permissions:

```
packageName:ClassName:memberName:r,w
```

Here `memberName` is the property, collection or action name, `r` indicates that the member is visible, and `w` indicates that the member is usable (editable or invokable). Shiro allows "*" to be used as a wildcard at any level, so the permissions for a given role can be very compact.

Closing Thoughts

This article should have given you a good idea of the Isis' programming conventions and how they translate into a fully working webapp, but we've really only scratched the surface of the benefits that Isis brings. I can guarantee that you'll be amazed at how productive you will find yourself once you start to use Isis in earnest.

And even though we tout Isis' killer feature as being its ability to create a UI directly from your domain object model, that isn't really what Isis is about at all. You'll quickly discover when you work with Isis that things like names matter very much in Isis; and names are very much the heart of the *ubiquitous language* concept. Being able to quickly and cheaply rename classes and class members significantly enhances understanding. This is why we call Isis a framework for domain-driven design.

It's easy to get started with Apache Isis. The quickstart archetype will generate the same application that we've discussed in this article, and you'll find plenty of help on the mailing lists [8]. I hope to see you there.

References

[1] http://isis.apache.org

[2] http://wicket.apache.org

[3] http://isis.apache.org/getting-started/quickstart-archetype.html

[4] http://datanucleus.org

[5] https://developers.google.com/appengine/docs/java/overview

[6] https://github.com/danhaywood/isis-wicket-gmap3

[7] http://shiro.apache.org

[8] http://isis.apache.org/support.html

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