METHODS & TOOLS

Global knowledge source for software development professionals **ISSN 1023-4918**

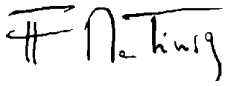
Summer 2003 (Volume 11 - number 2)

The Art of Compromise

The articles of Lisa Crispin and Hans Sassenburg are focused on two different technical domains, but they have something in common: they deal with the need make choices and to compromise. This discipline is rarely taught in software engineering courses. In our May text issue, we cited Bertrand Meyer who said: "It pays to be dogmatic here. It's hard to be a little bit object-oriented. Little violations beget huge disasters". Many training courses and books follow the same pattern. They present ideas as if our capabilities to choose and change things were unlimited, as if change was immediate without transition, as if they were no other option than to be or not to be. object oriented or something else.

However in the "real life" (at least mine as a participant to software development projects... ;-]), the choices that you can make are often limited by constraints. You deal with people, organizations, procedures, deadlines or legacy systems that sometimes you cannot change, or not in the short time-frame of your current project. Therefore, we are doomed to do "the best we can do in the current situation" instead of aiming for perfection, the natural incline of every software engineer...;-). Well, even if reality is not so bright, we are reluctant to recognize that we have compromised. How many times can we admit: this is not the best solution, but it will help us meet our deadline? This could be heard in some project meetings, but it will rarely presented to the "outside world".

The problem with the compromise is not that you have deviate from the "Truth", but rather that you often need time to evaluate the available options. However, time is a resource that is often scarce in the software development world. If you are the project manager of a tight-deadline project and you are asked to do a proposal for a new project, where will you find the time to study alternatives? How will a customer react if you propose three different answers to the same request for proposal? Does he really want to have the choice? Will he be ready to make trade-off decisions during the project? The answers to these questions are not obvious... just like finding a good compromise.



Inside

XP Testing Without XP: Taking Advantage of Agile Testing Practices

Lisa Crispin, lisa.crispin@att.net
http://lisa.crispin.home.att.net

In the rocket-fast late 90s, I struggled with using traditional software process and testing practices on e-commerce applications. Then I read Kent Beck's *Extreme Programming Explained* and had an amazing 'aha' moment. Ron Jeffries sums it up best:

Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation. (www.xprogramming.com)

Applying more discipline to traditional waterfall process had not helped my team. A new kind of discipline based on values such as communication, simplicity and feedback might be the answer! I immediately took an opportunity to join an XP team as a tester, and enjoyed working on XP teams for the next 18 months. Indeed, XP *did* allow us to produce high-quality e-commerce applications on time and on budget! Our customers were happy! We were happy! We struggled some with how I, as tester, could best contribute to the team, as the XP literature at the time didn't say much on that subject. With the whole team focused on quality and testing, we came up with some great solutions. I was so excited about XP testing that I co-authored a book on it (*Testing Extreme Programming*, with Tip House).

How XP Testing is Different

I found that XP testing was different in many ways from 'traditional' testing. The biggest difference is that on an XP project, the entire development team takes responsibility for quality. This means the whole team is responsible for all testing tasks, including acceptance test automation. When testers and programmers work together, the approaches to test automation can be pretty creative!

As Ron Jeffries says, XP isn't about 'roles', it's about a tight integration of skills and behaviors. Testing is an integrated activity on an XP team. The development team needs continual feedback, with the customer expressing their needs in terms of tests, and programmers expressing design and code in terms of tests. On an XP team, the tester will play both the customer and programmer 'roles'. She'll focus on acceptance testing and work to transfer her testing and quality assurance skills to the rest of the team.

XP Tester Activities

Here are some activities testers perform on XP teams.

- Negotiate quality with the customer (it's not YOUR standard of quality, it's what the customer desires and is willing to pay for!)
- Clarify stories, flush out hidden assumptions
- Enable accurate estimates for both programming and testing tasks
- Make sure the acceptance tests verify the quality specified by the customer
- Help the team automate tests
- Help the team produce testable code
- Form an integral part of the continuous feedback loop that keeps the team on track.

The Nature of XP Testing

The biggest difference between XP projects and most 'traditional' software development projects is the concept of test-driven development. With XP, every chunk of code is covered by unit tests, which must all pass all the time. The absence of unit-level and regression bugs means that testers actually get to focus on their job: making sure the code does what the customer wanted. The acceptance tests define the level of quality the customer has specified (and paid for!)

Testers who are new to XP should keep in mind the XP values: communication, simplicity, feedback and courage. Courage may be the most important. As a tester, the idea of writing, automating and executing tests in speedy two or three week iterations, without the benefit of traditional requirements documents, can be daunting.

Testers need courage to let the customers make mistakes and learn from them. They need courage to determine the minimum testing that will prove the successful completion of a story. They need courage to ask their teammates to pair for test automation. They need courage to remind the team that we are all responsible for quality and testing. To bolster this courage, testers on XP teams should remind themselves that an XP tester is never alone – your team is always there to help you!

XP Testing Without an XP Team

After my rewarding, if challenging, experience as a tester on XP teams, the bad economic times caught up with me and I had to take a job back on the "dark side". Oh, my new company was great and they were interested in XP, but they were a tiny shop with a huge job to do and it felt to me like barely controlled chaos. My initial attempts to apply XP practices, such as having daily XP-style stand-ups with my test team, were disasters. I just had to hunker down, hire people that I thought would be able and willing to try agile test practices, and wait for my opportunity.

Things eventually slowed down enough for my team to have time to spike test automation solutions. Many of our applications are written in Tcl. Tcl is a powerful scripting language which can lend itself to test automation, so we taught ourselves Tcl. We needed a way to load bulk data for volume testing, and one of the programmers helped out with a Tcl script; we took over maintenance for it and started expanding its functionality. Some of our test automation could be handled with fairly simple shell scripts. Other automation requirements demanded more robust tools. As we started down our own path of exploring test automation, the programmers started writing automated unit tests.

I learned the hard way that being an advocate for XP or any kind of agile development doesn't introduce change. Instead, you and your team need to identify what hurts and look at what XP or agile approaches could help heal that pain. By adopting XP-style planning and writing acceptance testing up front, my new development team has started exploring what agile practices might help them.

Let's look at what agile testing practices might help you, no matter what style of software development process you are using.

Values

Earlier in this article I referred to the values by which XP teams develop software. These were defined by Kent Beck in "Extreme Programming Explained":

- Communication
- Simplicity
- Feedback
- Courage

Some people have added their own values to this list – the one I liked best is 'enjoyment'. Extreme Programming is the only 'official' agile method I have actually practiced, but from what I have learned of agile software development in general, you can always turn to these values for guidance.

Communication is usually your first job. Whether you're new to a position or you just want to try something new, you're going to have to talk to your coworkers and your customers. Most importantly, you need to listen. As you proceed in your agile journey, remember the XP mantra: "Do the simplest thing that could possibly work". As a tester, feedback is your core function. You'll need plenty of courage to try agile practices in a less-than-agile environment.

Let's explore a typical project life cycle and see how XP practices can help you with testing, whether or not your development team is taking advantage of them.

Project Planning

Most projects begin with some kind of requirements gathering. The business experts may write business requirements. This is often followed by the development team producing functional specifications and/or technical specifications. If you're working on a project which is producing documents like this, try to get in on the planning meetings. I know, the last thing you need in your life is more meetings. Requirements gathering meetings can be a great opportunity to put your agile skills to work. You'll get a head start in understanding the project. By alternately looking at the requirements from the point of view of the end users, business experts and programmers, you can ask questions that flush out hidden assumptions.

Testing the Documentation

Whether or not you attend the meetings, you can add value to the project by 'testing' the documentation. This isn't an XP concept, but it is linked to flushing out hidden assumptions during XP planning games. One way to test documents is to read through the document, writing a list of assertions. For example, maybe you're reading requirements for an order entry system and have come up with these assertions:

- "The system will capture the customer service agent's name".
- "The new system will be 50% faster than the old system".
- "The billing address will be required."
- "The zip code will automatically populate the city name".

Go over your assertions with the document's author and clarify any ambiguous or contradictory assertions. Will the system capture the agent's name by means of the agent logging into the system? What exactly in the new system will be 50% faster – the response time on each page? The time for placing an order? Do we have a baseline for the old system? If not, how do we quantify this requirement so we know it's met? What about cases where multiple cities share the same zip code? Are there any optional fields in the address (such as the second address line)? Clearing up these issues now will save time for the programmers later, just as it does during an XP planning game.

If you've got a big project where the team sees a problem with finishing on time, see if there is a way you can suggest an XP approach to the planning. If the requirements can be grouped into story-like chunks, estimated by the programmers, and prioritized by the stakeholders, the team can take an iterative approach which will help ensure that some percentage of the necessary features are 100% ready by the deadline, even if less critical features are not yet implemented. If you're new to agile processes or new to this team, it might not be the right time to take such a bold step, but keep it in mind.

Acceptance Tests

Write the acceptance tests (meaning any tests needed beyond unit and integration tests: system, functional, end-to-end, performance, security, whatever) during the planning phases of the project, rather than after development is complete. More importantly, if you're trying an agile approach, write *executable* acceptance tests.

Executable acceptance tests serve two purposes. First of all, they're a mechanism to involve your 'customer' or business expert in defining the tests which will verify that the level of quality

and functionality he has specified is met. Secondly, they are a part of your automated acceptance test framework. If you can, avoid having to re-work your acceptance test cases into input to test automation tools.

There are quite a few test tool frameworks which allow this. The concept of data-driven testing isn't new to XP, and there are plenty of tools which let you define tests in an Excel spreadsheet and then use these as input to your automated tests. XP-style test frameworks abound, as well. Here are a couple of sample JUnit tests:

assertTrue(login("bob","bobspassword"))
assertFalse(login("bob",""))

The first test proves that logging in with a valid userid and password works correctly. The second proves that logging in without a valid password fails. If you have fairly savvy business users, you can train them to read this syntax. If not, it's not hard for your programmers to write a tool to suck data out of an Excel test case and input it to JUnit.

But wait a minute, you aren't on an XP team where your programmers are totally on board to help you with acceptance testing. That might rule out Fit (http://fit.c2.com) and FitNesse (http://www.fitnesse.org) as well. Fit allows you to define acceptance tests as HTML tables, which are input to fixtures written in Java, Ruby or some other language that executes the tests. FitNesse goes one better by letting you define the Fit tests as a Wiki page, a sort of modifiable HTML page that's even easier to handle than HTML. It will also let you define tests in an Excel spreadsheet and paste them onto the Wiki page. Here's a sample FIT test:

fit.ActionFixture		
Start	Page	
Enter	Location	http://www.searchme.com
Enter	Userid	bob
Enter	Password	bobspassword
Check	Title	Welcome to SearchMe

If you really can't get any programmer support to write the simple fixtures that drive these 'behind the GUI' tests, check out Canoo WebTest, This allows you to define test cases in XML format. Again, users might not be totally comfortable with XML format, so you might have to do the spreadsheet-to-test tool conversion.
(Canoo url: http://webtest.canoo.com/webtest/manual/WebTestHome.html.)

It doesn't matter what tool you're going to use. Defining acceptance test cases up front helps your programmers write the correct code. It gives you a huge jump on your testing too.

A word about test plans. Nobody ever reads them, so try not to write them. They're cumbersome to write and impossible to maintain. Just write test cases. If your organization requires you to

deliver a test plan, try to include information that will be useful to you, such as a release plan, and refer to the test cases as much as possible.

Technical Design

Depending on your technical skills, technical design meetings may be painful for you. Try to attend them anyway, if you have the opportunity. If you can even ask one question or make one comment that leads to a more testable design, you're way ahead of the game.

XP has the wonderful benefit of making everyone on the development team conscious of testability. If you have to automate and perform acceptance tests, and it's hard to do because your GUI is thick or hooked up to the back end in a way that you can't test much behind the GUI, your next application is bound to be designed to be more testable. Unfortunately, your traditional development project won't give you this golden opportunity. You can still raise these issues yourself (tactfully, of course) at design meetings. You can also continue to help facilitate communication between the business stakeholders and the programmers. You understand the application pretty well by now, and you probably are more familiar with the technical issues than the folks on the business side. Remember that *communication* value!

Executing Tests

Hopefully, you had the opportunity to write executable tests. If so, when code is delivered to you for testing, you've got a major jump on the test automation.

You're not on an XP team, but maybe there are still ways you can benefit from the advantages of 'pairing'. Collaborate with a fellow tester to spike some test automation solutions. Engage a programmer in brainstorming automation solutions.

The refactoring practice, changing code for the sake of maintainability or efficiency, is important for test automators. Take a little time every day to make little changes to your automated test scripts that will help you maintain them and run them effectively in the long term. You never really get the chance for that big modification, so take tiny steps as you go.

Start Early

Can the developers give you little bits and pieces to test? If you were able to convince them to take an XP-style approach to planning, this part is easy. They'll develop testable chunks in short iterations and you can race along. If not, see what you can get delivered to you early. Maybe a programmer has a solution in mind but isn't sure if it will scale well. Help by doing a performance test.

Involve the business experts as much as possible in testing. Keep simplicity in mind: What's the simplest way to execute these tests? I hate to have to say it, but automation isn't always the answer. You may have a one-time test, or you may need intelligent exploratory testing by a subject matter expert. Just remember to keep it as simple as possible! In most software projects, no matter what the methodology, you only have time to test the minimum.

Let Worry Be Your Guide

Risk assessments will help you and your stakeholders from worrying. Take a deep breath and determine what the *minimum* amount of testing will prove to the customer that the system behaves as specified. Lots of people have a negative view of the word 'minimum', but if it weren't 'good enough', it wouldn't be the minimum. Have courage: in most software projects, if you pull off the minimum testing, you're golden.

XP builds risk assessment into the system with the planning game. The programmers always estimate all the stories and state their potential velocity, and the business experts or customers choose the ones most important to them that will fit into each iteration. With a more traditional project, look to a traditional risk assessment to help you. Identify risk areas, assessing the probability and impact of each. Work to mitigate the top one or two risks, then tackle the next one or two. Here's a sample risk assessment:

Item	*Impact*	*Probability*	*Risk*
Search produces no results	4	3	12
System capacity exceeded	5	2	10
Data out of date	4	4	16

Numbers like these often get management's attention when nothing else will.

Retrospectives

Retrospectives are so important, some gurus such as Martin Fowler consider them to be one of the XP practices. Quality assurance is all about learning from history. Retrospectives let your team continually improve how you do your jobs.

Identifying Areas of Pain

There are different approaches to retrospectives (I much prefer this positive term to the oft-used 'post-mortem'). Generally, your team looks at what went well, what didn't go so well, what the team should start doing, what the team should keep doing, and what the team should stop doing. It's sometimes hard to find time for retrospective meetings, but they're a giant opportunity for the agile tester. I've found they are an ideal place to demonstrate how XP or other agile practices helped testing on the project. The goal of retrospectives is to identify areas of pain and form an action plan to mitigate the top two or three. This is a great opportunity to look to agile practices for solutions.

In my case, over the course of many months, our test team built up credibility by doing a super job of manual testing. Then, we had a chance to show what value we could add with test automation. We did our own internal retrospectives, and demonstrated improvement by focusing on our problem areas. I also took every opportunity to expose the technical management and the programmers to agile techniques, by encouraging attendance at XP Denver meetings, distributing links and magazine articles, and taking advantage of an XP/agile consultant who offered a couple hours of his time to demonstrate test-driven development.

As a result, the leaders of the development team are curious about what agile practices might help them. Retrospectives are an ideal time to talk about these. Are requirements documents always inadequate, or take too long to produce? Writing acceptance tests up front, if you haven't been able to do this up to now, is the ideal solution! This isn't an article on how to introduce agile practices, but as a tester who benefits from XP testing practices, you can help your team improve quality by seeing what else it can borrow from the XP and agile world.

Celebrating Success

My favorite aspect of XP is how it celebrates success. When programmers or testers complete a task, they ring a bell or go play a round of Foosball (maybe nobody plays Foosball anymore in this economy!) I like to bring treats such as brownies to celebrate a release or just making it through a hard week. We're often too busy going on to the next project(s) to notice, but it's a nice lift for our team to go out to lunch or a little happy hour after a retrospective meeting to toast our accomplishments. Like all creatures, we do better with positive reinforcement than with negative punishment.

Perhaps the most important aspect of XP, which should be applied across all software development processes, is the one-team approach. Testers aren't a separate, after-the-fact entity. They're an integral part of the development team. As my buddy Mike Clark says, there is no "QA" in "team". Do whatever you can to help your whole team work towards delivering high-quality software on time.

Testing provides continuous feedback to help your whole team do their best work. Take baby steps if you have to – any progress is hugely rewarding! Above all, remember to follow Kent Beck's advice, and embrace change!

Modeling the Real World for Load Testing Web Sites

Steven Splaine, Steve@Splaine.net
www.splaine.net

Introduction

If you want to get an accurate idea of how your Web site is going to perform in the real world, it pays to create a load profile that closely models conditions your site will experience. This article addresses nine elements that can affect Web load.

Requesting your Web site's home page 100 times per minute is not going to give you a very accurate idea of how your Web site is actually going to perform in the real world. This article seeks to provide an overview of some of the load parameters that you should consider when designing the test load to be used for performance testing a Web site. The more accurate the *load profile* can be made, the closer the performance tests will be to modeling the real world conditions that your Web site will ultimately have to survive. This, in turn, will lead to more reliable test results.

1. User Activities

Many transactions occur more frequently than others, and should therefore comprise a larger proportion of the test data and scripts used for performance testing.

In order to simulate the real workload that a Web site is likely to experience, it's important that the test data and scripts are a realistic representation of the types of transactions that the Web site can be expected to handle. If your organization typically only sells one product for every fifty visitors, it would be unrealistic to weigh two test scripts evenly (one mimicking a browser customer, the other an actual buyer). A better approach would be to use a 50:1 weight (fifty browsers for every buyer).

2. Think Times

The time that it takes for a client (or virtual client) to respond to a Web site has a significant impact on the number of clients that the site can support. People, like computers, have a wide range of different processing levels and abilities. Different users require various amounts of time to think about the information that they have received. Some users race from one Web page to another, barely pausing long enough to comprehend what they've seen; others need some time to contemplate what they've read before moving on to the next page. The length of this pause is called *think time*. Think time is generally considered to be the length of time from when a client receives the last data packet for the Web page currently being viewed to the moment that the client requests a new page.

In theory, a Web site that can support a maximum of 100 "10-second aggressive" clients should be able to support 300 "30-second casual" clients because both types of clients result in 600 transactions per minute. Unfortunately, this theory only holds true for the most rudimentary of Web sites. Web sites that are more interactive require resources for both active and dormant clients, meaning that the 300 casual clients are likely to require more resources than the 100 aggressive clients.

When recording or writing test scripts for performance testing, you should consider how much time each of the various types of clients might spend thinking about each page. From this data, you can create a reasonable distribution of timings and, possibly, a randomizing algorithm. Web logs can be a good source for estimating think times for Web pages that have already been hosted in production.

3. Site Abandonment

Basic test scripts/tools assume that a client will wait until a Web page has been completely downloaded before requesting the subsequent page. Unfortunately in real life, some users may be able to select their next page before the previous page has been completely downloaded (e.g., they may know where they want to go as soon as the Navigation bar appears).

Alternatively, some users will surf off to another Web site if they have to wait too long (i.e., terminate the test script before reaching the end of the script). The percentage of users that abandoned a Web site will vary from site to site depending on how vital the content is to the user (e.g., a user may only be able to get their bank balance from their bank's Web site, but could get a stock quote from numerous competitors). One thing is certain; the longer a user has to wait, the more likely it is that they will abandon the Web site. Therefore, in order to model the real world, test scripts should include a randomized event that will terminate a test script execution for a particular client if the client is forced to wait too long for the page to download. The longer the delay, the more likely that the client will be terminated.

4. Usage Patterns

Unlike typical mainframe or client-server applications, Web sites often experience large swings in usage depending on the type of visitors that come to the site. U.S. retail customers, for example, typically use a Web site in the evenings (7:00 p.m. EST to 11:00 p.m. PST). Business customers typically use a Web site during regular working hours (9:00 a.m. EST to 4:00 p.m. PST). The functionality of a Web site can also have a significant impact on usage patterns. U.S. stock quotes, for example, are typically requested during market trading hours (9:30 a.m. EST to 4:00 p.m. EST).

When attempting to model the real world, you should conduct some research to determine peak usage ramp-up and ramp-down, peak usage duration, and whether any other load profile parameters vary by time of day, the day of the week, or another time increment. Once researched, schedule tests that will run over the real Internet at appropriate times of the day/week.

5. Client Platforms

Different client-side products (e.g., Browsers and O/Ss) will cause slightly different HTTP traffic to be sent to the Web server. More important, if the Web site has been designed to serve up different content based on the client-side software being used (a technique commonly refereed to as browser sniffing), then the Web site will have to perform different operations with correspondingly different workloads.

Some browsers allow users to change certain client-side network settings (threads, version of HTTP, and buffer sizes) that affect the way the browser communicates and thus the corresponding workload that a browser puts on a Web server. While few users ever change their default settings, because different browsers/versions have different defaults, a more accurate test load would vary these values.

6. Client Preferences

Most browsers also allow users to change client-side preferences; but again, few users actually change their default settings. However, different products/versions of a browser may have different default settings. For example, a browser with cookies disabled will reduce the amount of network traffic due to the cookies not being sent back and forth between the Web site and the browser; however, it might increase the resource requirements of the application server as it struggles to maintain a session with the user without the convenience of the cookie.

If encryption is going to be used to send and receive secure Web pages, the strength (or key size) of the encryption used to transfer the data will be dependent upon a negotiation that takes place between the Web server and the browser. Stronger encryptions utilize more network bandwidth and increase the processing requirements of the CPUs that perform the encrypting and deciphering (typically the Web server). Therefore, users with low settings (e.g., 40-bit keys) will put less of a strain upon a Web server than users with high settings (e.g. 128-bit keys).

By indicating that they do not want graphics or applets downloaded, Web site visitors will not only speed up the delivery of a Web page that contains these files, but will also consume a smaller portion of the Web site's bandwidth and fewer Web server connections. If present in significant numbers, these clients can have a noticeable effect upon the performance of the Web site.

7. Client Internet Access Speeds

The transmission speed or bandwidth that your Web application will use can have a significant impact on the overall design, implementation, and testing of your Web site. In the early days of the Web (circa mid-1990s), 14.4 Kbps was the most common (e.g., standard) communications speed available. Hence, 14.4 Kbps became the lowest common denominator for Internet access. When 28.8 Kbps modems were introduced, however, they offered a significant performance improvement over 14.4 Kbps modems and quickly surpassed 14.4 Kbps modems in popularity. When 56.6 Kbps modems were introduced, the performance improvement wasn't as significant. Consequently, 28.8 Kbps are still in use and unlike the 14.4 Kbps (which has nearly vanished) still comprise a significant (although decreasing) proportion of the Internet population. Many companies therefore use the 28.8 Kbps transmission speed when specifying the performance requirements of their Web site.

8. Background Noise

Unless the production servers and network are going to be dedicated to supporting the Web site, you should ensure that the servers and network in the system test environment are loaded with appropriate background tasks. When designing a load to test the performance of a Web site or application, consider what additional activities need to be added to a test environment to accurately reflect the performance degradation caused by "background noise." Background noise is created by other applications running that will also be running on the production servers once the application under test moves into production, and other network traffic that will consume network bandwidth and possibly increase the collision rate of the data packets being transmitted over the LAN and/or WAN.

9. User Geographic Locations

Due to network topologies, response times for Web sites vary around the country and around the world. Internet response times can vary from city to city depending on the time of day, the geographic distance between the client and the host Web site, and the local network capacities at the client-side. Remote performance testing can become particularly important if mirror sites are to be strategically positioned in order to improve response times for distant locations.

But how can you effectively test the performance of your Web site from locations that are thousands of miles away? Possible solutions include

- using the services of a third-party company that specializes in testing a Web site from different locations around the world
- utilizing the different physical locations (branches) that your organization may already possess, coordinating the execution of your test plan with coworkers at the offices
- using a modem to dial ISP telephone numbers in different cities and factoring out the additional time for the cross-country modem connection
- buy an “around the world” airplane ticket for one or more of the Web site’s testers

Getting the Right Mix

When developing the load profile that will be used for performance testing, try to take into account as many of the previously mentioned parameters as possible. While a single parameter may only affect the test results by a few percent, the accumulation of several parameters may add up and have a significant impact of the test results.

Credit

This article is drawn from *The Web Testing Handbook* by Steve Splaine & Stefan Jaskiel (amazon.com) and from SQE’s *Web Performance & Security Testing* training course (sqe.com). The article is also available for download at stickyminds.com.

When can the software be released?

Hans Sassenburg, hsassenburg@se-cure.ch
SE-CURE AG, www.se-cure.ch

Introduction

Software suppliers create a software products and different releases of those products. Initial releases of new products are developed to penetrate into new or existing markets, subsequent releases are developed to add additional features or improve reliability. Despite the variety and increasing amount of software products, most suppliers still struggle to determine the right moment to release a product as well as the exact release contents. Problems during product development are the difficulty to determine to which degree product features have been implemented and what are the additional costs and schedule required to implement the remaining functionality. But even when the product is released, it is not always clear what the behavior of the product will be under different circumstances and what the impact of this will be on sales and post-release maintenance costs. Being not able to predict the behavior of software products can be dangerous as the impact of software on society is increasing rapidly. A frequently overheard misconception about reliability is that this can be related to an acceptable level of failure, like "failure rate is one in a thousand". This might be true for certain products used under certain circumstances. But how do you explain this to that particular patient in a hospital being x-ray diagnosed, when the system fails?

In this article some ideas and new approaches are presented which may help to answer the following questions:

- What are the most important strategic goals for a supplier and his projects and what is their influence on the release decision?
- Which techniques can be applied to software development to minimize, prioritize and quantify product features?
- How can quantified product features be deployed and measured during software development?
- Which tradeoffs can be made between costs, time to market and reliability?
- How can the reliability of a software product be predicted, even in the case of incomplete information?
- Which economic decision factors can be distinguished to decide whether or not a software product can be released?

Strategic Supplier Goals

Grady describes three possible strategic goals of software suppliers [GRA 1992]:

- Maximize customer satisfaction. This is accomplished mainly by offering products, which will both satisfy and delight customers. Other factors are important as well, for instance the price of the product and the required level service and maintenance efforts and costs from a customer's point of view. Maximizing customer satisfaction means for a project that essential *product features* must be identified and implemented.

- Minimize engineering effort and schedule. Improving productivity is important, as it will help to decrease development costs (from which a customer may also benefit). Shortening

development times will help to deliver products faster, which can be a highly competitive advantage in today's marketplace (and can offer another benefit to a customer). Minimizing engineering effort and schedule means for a project that work must be performed efficiently to reduce costs and that *time to market* must be minimized.

- Minimize defects. Minimizing defects during development will limit the amount of rework. This will also have a positive impact on minimizing engineering effort and schedule. Further, post-release costs will probably decrease as the product contains fewer defects. The customer will also benefit from reliable products and thus minimizing defects will have a positive contribution to customer satisfaction. Minimizing defects means for a project that product must be developed with a high *reliability*, which can be accomplished by applying appraisal methods like reviews and inspections.

Since the early nineties many supplier have initiated process improvement programs to improve these strategic capabilities. There is however little evidence, that conformance to process standards guarantees good products, which meet customers' demands [KIT 1996]. However, this criticism may be unfair as popular process models (CMM) and standards (ISO 9001, ISO 15504) also insist that process improvement enhances product quality [KIT 1996].

When developing products not all goals are equally important for a project. It depends on the lifetime of a product, which goal has highest importance. It can be seen from Figure 1 that the priorities of the goals shift during the evolvement of a product.

Market description	Introduction	Early adopters	Mainstream	Late Majority	End of Life
Buyer profile	Technology enthusiasts	Visionaries	Pragmatists	Conservatives	Skeptics
Importance	1. Time to market 2. Product features 3. Reliability	1. Time to market 2. Product features 3. Reliability	1. Reliability 2. Time to market 3. Product features	1. Reliability 2. Product features 3. Time to market	1. Reliability 2. Product features 3. Time to market

Figure 1: Project priorities as a function of a product's lifecycle ([ROT 2002], [MOO 1995]).

This information is important, as it will help a project to set the right priorities. As such, it supports in global terms the decision-making process when to release a software product.

Product Features

When discussing product features often distinction is made between functional and non-functional requirements. The non-functional requirements are often referred to as the quality requirements. In the widely used ISO 9126 standard [ISO 9126] quality requirements are mapped onto quality characteristics (like 'maintainability') and then onto quality sub-characteristics (like 'changeability'). The term quality introduced here can be very misleading and is often interpreted differently depending on the perspective taken. Figure 2 lists five different views on quality.

Trancendental view	**User view**	**Manufacturing view**	**Product view**	**Value-based view**
quality can be recognized but not measured (gut feel)	quality is fitness for purpose (usefulness)	quality is the conformance to stated requirements	quality is the presence or absence of product characteristics	quality is what the customer is willing to pay for

Figure 2: Views on quality [GAR 1984].

When talking about "the quality requirements of a product" like in the ISO 9126 standard, one defines quality from both a user view (set of six quality characteristics) and a product view (decomposition of quality characteristics into quality sub-characteristics). Other models like the McCall Model [McC 1977] use other definitions like "quality factor" instead of "quality characteristic". From here on however, quality will be defined from a value-based view: "a quality product is a product that provides performance at an acceptable price or conformance at an acceptable cost". This definition complies with the definition for quality in the ISO 8402 standard: "The totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs" [ISO 1994] with the addition of the acceptability of cost. Further, the term non-functional requirements will be used to denote those requirements that define product properties and put the constraints upon the functional requirements. They determine the behavior of a product.

Defining and managing product features is one of the most challenging tasks for software projects. Unclear and unstable requirements are two of the main failure factors of software projects [STA 1995]. Several models can be used to capture, organize and prioritize (functional and non-functional) requirements. A relatively new method is Software QFD (quality function deployment). QFD is a method developed in Japan [AKA 1990] and is mainly used in

manufacturing environments. It provides a systematic way to identify the essential customer requirements ("fitness for purpose") and bridges the gap to the customer, which has been left behind by other requirements engineering methods used today. The QFD method prevents software suppliers from "over-specifying" product requirements and in this way implementing the wrong (unwanted) functionality. When transferring classic QFD to software development, two essential differences have to be taken into consideration [HER 1999]:

- The production process in software industry is (compared to manufacturing) a mere duplication process and the implementation process can hardly be influenced by adjustable process parameters. Therefore, QFD applied in a software development context has to focus on the early phases of development.
- Software as a product is identified by its behavior and not by its physical characteristics. This means that one has to take into account both the functional and non-functional requirements. The latter ones determine the behavior by putting constraints on the functional requirements.

The basic instruments that facilitate the QFD process are the 'houses of quality'. These houses are used to make the relationships clear between "What" is to be produced (derived from customer needs) and "How" it is to be produced. In Figure 3 an example is given for the "top" house of quality. The customer requirements ("WHATS") are prioritized and a rating of how well competitors meet the requirements is included. Note that it may be considered to add other "WHATS" to the customer requirements, like the requirements of a maintenance department within the supplier's organization. This holds for the additional requirements of all people having certain interests regarding the software: the stakeholders. The relationships between "WHATS" and "HOWS" can be expressed in terms of strong, medium, weak or none on an ordinal scale (to be converted to values on a numerical scale for further calculations). Further, the type of association or tradeoffs between "HOWS" can be indicated (for example: strong positive, positive, negative, strong negative). Finally, target values for the "HOWS" can be specified and the priorities can be calculated from the relationships with the "WHATS".

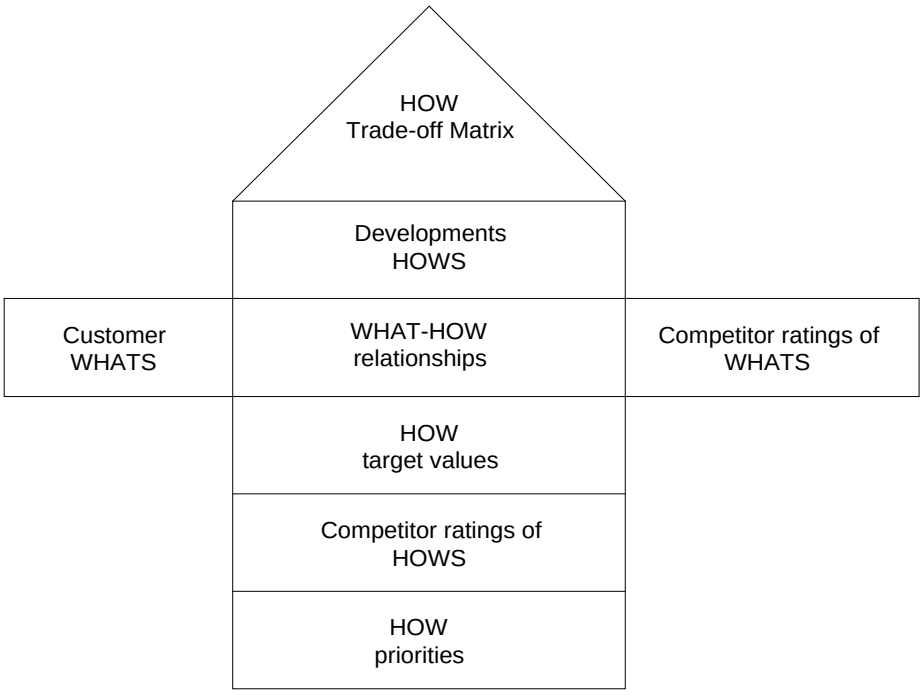


Figure 3: House of Quality structure (QFD).

Different models for Software QFD have been developed ([ZUL 1990], [OHM 1993], [HER 1999]). All authors claim that positive experiences have been gained. The technique of Software QFD may have several advantages above model- or standard-oriented approaches during the requirements engineering process. By bridging the gap to the customer the most important customer needs (WHYS) are derived and translated into software requirements. One could then build a house of quality with the functional requirements as the WHATS and the non-functional requirements as the HOWS. By assigning target values to the HOWS quantified release criteria become available. Note here that in the end release criteria deal with both the functional requirements (presence of functions) and the non-functional requirements (behavior of the functions). As such, the technique holds certain promises. A critique note here is however that dealing with the quantification of non-functional requirements is not an easy task (how to quantify maintainability or reusability for instance in such a way, that it can be easily measured afterwards?). But, this statement is true for any requirements engineering method used.

After having defined the product features (and thus the quantified release criteria) they must be deployed during the architecture phase. Two steps may be considered. First, architecture alternatives are evaluated to determine to which extent each alternative meets the functional and non-functional product features. The quantified non-functional requirements play an important role here as they specify the behavior of the product. An overview of available approaches is given in [LOS 2002]. Most approaches (like for instance SQUID [BOE 1999]) make a distinction between the external non-functional requirements that determine the product's operational behavior, from which the internal non-functional requirements are derived. The goal is to monitor and control these internal non-functional requirements through project measures to achieve the external non-functional requirements. Second, ideally the product features can be deployed to the identified components in the selected architecture. Product features (functional and non-functional) are decomposed into smaller parts and assigned to individual components. In this way, it is clear in the implementation phase what the requirements of each module are and how each module can be tested to see if the requirements have been met.

The two steps described here to move from requirements to architecture are difficult to be carried out when put into practice. The fundamental problem is that individual requirements represent fragments of behavior, while an architecture that satisfies a set of requirements represents integrated behavior [DRO 2003]. This task is further complicated by defects in the original requirements and changes to them. Further, quality models like for instance ISO 9126 [ISO 1991] and the McCall model [McC 1977] share some common problems [KIT 1996]:

- They lack a rationale for determining the hierarchy (characteristics and sub-characteristics in ISO 9126, factors and criteria in the McCall model), making it impossible to use the model as a reference to define all non-functional requirements.
- There is no description of how the lowest level metrics (indicators in ISO 9126) can be used to evaluate non-functional requirements at a higher level. Dromey suggests instead a bottom-up approach by defining and building in a consistent, harmonious, complete set of product properties and link these product properties to high-level non-functional requirements [DRO 1996].

So, when dealing with release criteria weak points in the software development process are the step from requirements to architecture (architecture evaluation and selection) and the step from architecture to implementation and test (deployment). Further research is conducted in this area and recently, attention is paid to integrate cost modeling in the architecture phase ([ASU 2000] and [ASU 2001]). Most approaches in the architecture phase offer a framework for software architects to discuss about technical tradeoffs. No attention is paid to reason about the economic costs and benefits of the various design options.

Costs and time to market versus reliability

Developing software products is normally characterized by business pressure to minimize costs and time to market. It is often stated that delivering a higher quality product does not necessarily mean that development costs will increase. This is only partially true. For instance, striving for higher reliability through investing in appraisal techniques like reviews and inspections will be paid back up by a decrease in the repair costs of finding and fixing defects. There is however an optimal level (see Figure 4). Beyond this point a further increase in appraisal costs will not have a net positive effect (for sake of completeness, one should also take into account the effects on post-release costs):

$$\textit{delta (appraisal costs)} + \textit{delta (repair costs)} < 0$$

Amongst other factors, the need for higher reliability depends on the phase in the product's lifetime (see Figure 1).

Time to market is also influenced by the phase in the product's lifetime as well as other characteristics of a market like the level of competition. Figure 5 depicts some examples of profit models related to time to market. When for instance the entry of a new product is delayed in a market with heavy competition, the probability of a supplier capturing the "Early Adopters"-advantage will decrease.

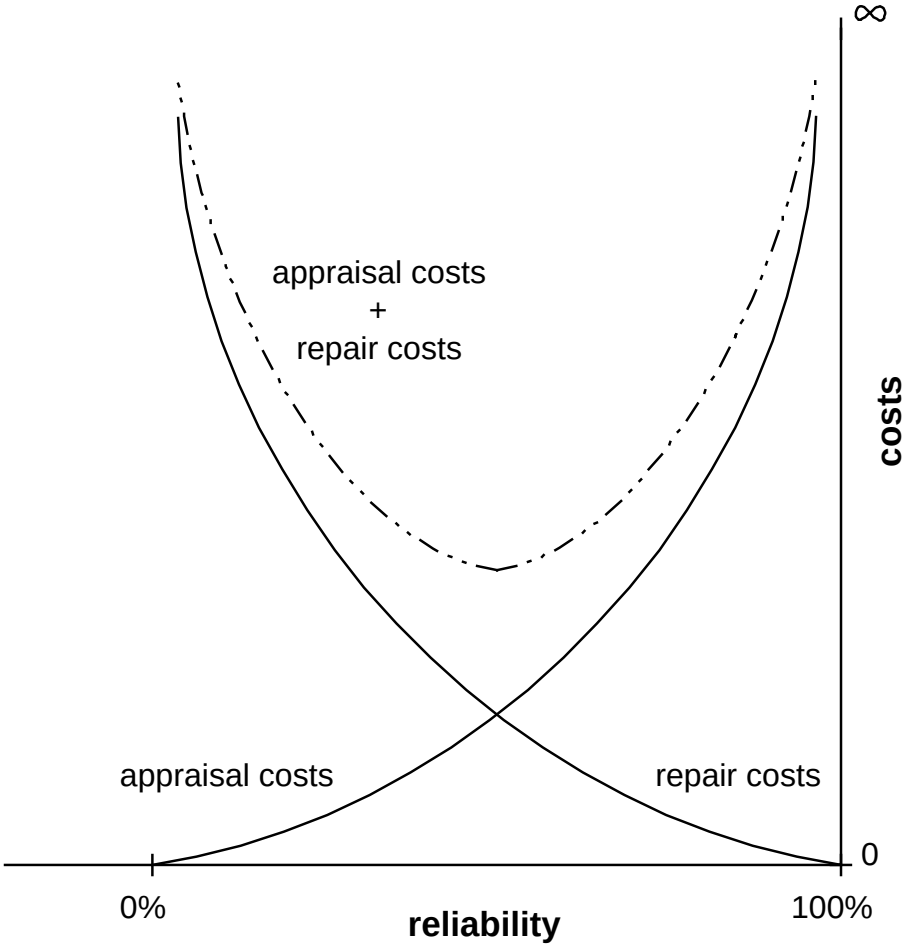


Figure 4: Appraisal costs versus repair costs.

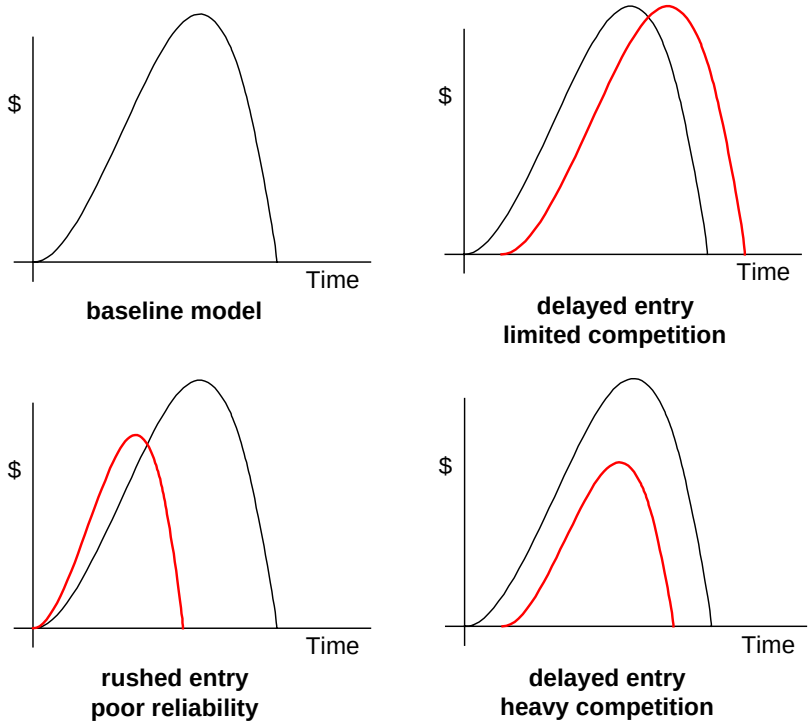


Figure 5: Examples of profit models.

Reliability

Although reliability is in most models and standards defined as a non-functional requirement explicit attention is given to this requirement. Reason lies in the fact that reliability is considered important in nearly every software product and that the impact of not meeting reliability requirements may be dramatic on users and their environments. Software defect prediction models have been developed since the seventies. Fenton and Neil have studied the most widely accepted models [FEN 1999]. They identified severe problems such as:

- There is no distinction made in different notions of ‘defect’.
- Statistical methods are often flawed.
- Product size is wrongly assumed to be a causal factor for defects.
- Obvious causal factors are not taken into account.
- Black box models hide crucial assumptions.
- The models cannot handle uncertainty.

They conclude that as a result these models provide little support for determining the reliability of a software product. Their study also showed that the number of pre-release faults is not a good indicator of the number of post-release faults. The problem is that many software suppliers use the pre-release fault counts as a measure for the number of post-release faults, e.g. the reliability of the released product.

These research outcomes combined with a further investigation led to the conclusion by Fenton and others that Bayesian nets offer a model that takes into account the crucial concepts missing from classical approaches [FEN 1998].

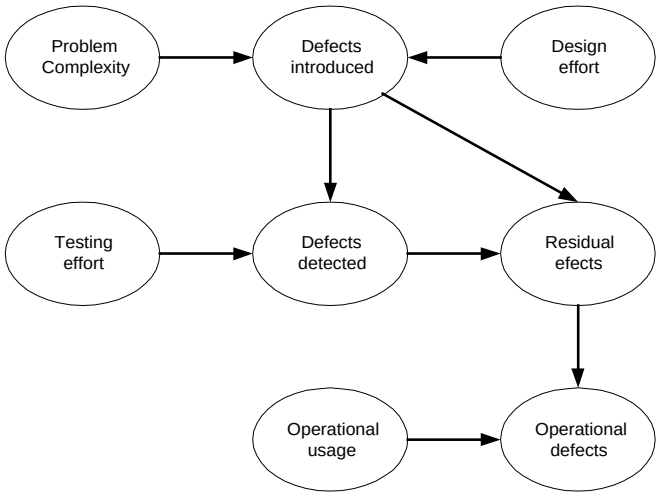


Figure 6: Example of Bayesian Net for defects [AGE 2002].

A Bayesian net is a graphical network (see Figure 6) together with an associated set of probability tables. The nodes in the net represent uncertain variables and the arcs in the net represent causal/relevance relationships between the variables. Classical prediction methods do not take these relationships into account, but focus on correlation between variables (for instance size and defects). The probability tables for each node provide the probabilities of each state of the variable of that node. For nodes without parents these are just the marginal probabilities while for nodes with parents these are conditional probabilities for each combination of parent state values [AGE 2002].

Once a Bayesian net has been set up evidence about variables (as soon as available) can be entered. All the probabilities will be updated accordingly, offering valuable information about variables we are interested in predicting. Suppose we are interested in a low number of operational defects (post-release defects) as a requirement. Having entered all available evidence (for instance “high” problem complexity), the entire network will be updated and the probability will be calculated for each possible value of the testing effort variable (for instance: “very low”, “low”, “medium”, “high”, “very high”).

Using this method has certainly advantages over the classical prediction models. A Bayesian net can take into account all products and process factors that influence a variable. It will at the same time not always be easy to define a Bayesian net quickly and exactly. Building a network for an entire product to be developed can become difficult and requires the support of tools to hide the complex calculations. However, it is a technique that facilitates the decision-making process of when to release a software product by analyzing all available evidence data and in this way making predictions of the reliability.

Conceptual Economic Model

The one and only appropriate measure a software supplier would place on the decision whether or not to release a product is the profit difference. Suppose a software supplier produces a product to be sold at a price *p*. Profits are revenues (price times quantity) minus costs (pre-release and post-release costs):

$$\textit{supplier profits} = \textit{revenues} - \textit{costs} = p \,.\, q - \textit{pre-release costs} - \textit{post-release costs}$$

(in this simplified model other costs like production costs and sales costs are not taken into

account)

In order to be able to predict profits the following sub-questions related to expected revenues and costs must be answered:

- Which product features have been implemented and tested?
- What is the current level of reliability compared to its target value?
- What are estimated sales figures (price, quantity, reputation) when the product is released now?
- What are estimated post-release costs when the product is released now?

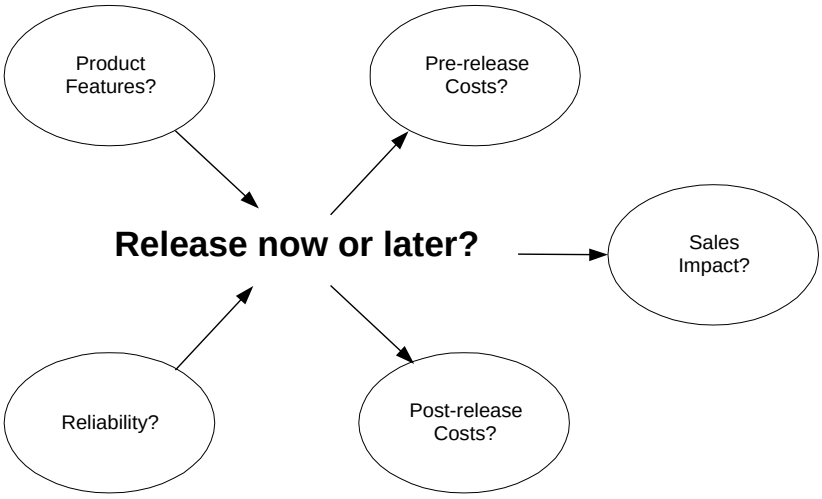


Figure 7: Decision factors for product releasing (supplier perspective).

As a second step the counterfactual situation must be considered. Delaying the time to market will have several consequences depending on the phase in the product's lifetime and the characteristics of a market as discussed before. Extending the development time will bring extra product features and higher reliability that may have a positive effect on price and quantity, but the impact may be negative as well in a highly competitive market.

$$\textit{supplier profits`} = p` \,.\, q` - \textit{pre-release costs`} - \textit{post-release costs`}$$

In order to be able to predict profits in the counterfactual situation the following sub-questions related to expected revenues and costs must be answered. Sub-questions to be answered are:

- What are the additional pre-release costs to improve and extend product features?
- What are the additional pre-release costs to improve the reliability?
- What will be the impact on sales (price, quantity, reputation) when market introduction is delayed?
- What will be the impact on post-release costs when the reliability is improved?

With this information the profit difference of the supplier can be calculated:

$$\textit{delta (supplier profits)} = \textit{supplier profits} - \textit{supplier profits`}$$

In practice, these questions remain unanswered in most cases. There is no analytical approach to model the supplier profits as a function of product features and reliability. A combination of the

following non-analytical methods are used to decide when a software product is "good enough" to release [RTI 2002]:

- A "sufficient" percentage of test cases run successfully.
- Statistics are gathered about what code has been exercised during the execution of a test suite.
- Defects are classified and numbers and trends are analyzed.
- Real users conduct beta testing and report problems that are analyzed.
- Developers analyze the number of reported problems in a certain period of time. When the number stabilizes or remains below a certain threshold, the software is considered "good enough".

There may however even be cases in which suppliers ship the product when there is no budget or schedule left for further testing and repair activities.

Answering the raised questions about product features, reliability, costs and sales impact is on the other hand not a task that can be easily accomplished. It requires a mature development process to define, deploy and evaluate the release criteria for product features and reliability. Further, all stakeholders within the supplier organization will have to work closely together with the development team. Product management must play a central role by defining and managing the entire lifecycle of each product. Marketing or sales must be involved to deliver inputs with respect to sales impact analyses and profit models. The group or department responsible for maintenance of released products must be involved to define maintenance criteria as part of release criteria and give input with respect to post-release costs.

A conceptual economic model should possibly also incorporate the profits of the end-user(s) of a product.

delta (economic welfare) = delta (supplier profits) + delta (end-user profits)

What are the profit differences for the end-user in case the product features are extended or the reliability is improved, e.g. when releasing the product is delayed? This will be difficult to determine in most cases as the end-user's profile heavily depends on product and market characteristics. In general, one could say:

end-user profits = revenues – pre-purchase costs - software expenditures –post-purchase costs

For a specific product in a specific market, the effect of extended product features or improved reliability on the end-user profits could be predicted.

Conclusions

Every software supplier is confronted with the question when to release a software product. Answering this question requests a supplier to define measurable release criteria. On one hand, these criteria must be oriented towards the product. What is the minimal set of product features to satisfy and delight the customers and what reliability level is still acceptable? On the other hand, the necessary data also comprise sales and cost information. What is the impact on sales and (pre-release and post-release) costs when the product is released now or later? Techniques like Quality Function Deployment applied in the context of software development may be of help to define, prioritize and quantify product features. However, deploying and measuring these quantified features during subsequent development phases is still an uncultivated area and

needs further research. Various models are being developed and tested to support the design of software architectures, but there is still limited insight how lower-level metrics can be used to assess quantified features at a higher product level. Predicting the reliability of software products is of increasing importance as the variety and amount of software products in our society increases exponentially. The consequences of making an ill-judged decision can be highly critical for the end-user and thus indirectly for the software supplier. Classical defect prediction models are considered naïve as they fail to include all the casual or explanatory variables. Further, they cannot handle uncertainty and incomplete information. Newer models like Bayesian nets sound promising as they eliminate the problems with classical models and do not introduce any additional metrics. They make use of available evidence data.

Building an economic model to predict profit differences will be of great help to support the decision-making process whether to release or not. It bridges the existing gap between project teams and other stakeholders within the supplier organization. By defining and managing quantified release criteria project teams are directed to focus continuously on those activities that contribute most to meeting those criteria. The data needed for calculations and predictions require a mature development process as well as close co-operation between the different stakeholders within the supplier organization. Incorporating the profit differences for the end-users of the product can extend this economic model. However, due to the variety of end-users' profiles only general statements can be made. Such an extended model must be tailored to the specific characteristics of a product, its market and its end-users.

References

[AGE 2002] *"Software Metrics and Risks"*, Technical Report, Agena Limited, 11 February.

[AKA 1990] *"Quality Function Deployment"*, Y. Akao, Productivity Press, Cambridge (Mass.).

[ASU 2000] *"An Architectural Approach to Software Cost Modeling"*, J. Asundi et al., Second International Workshop on Economics-driven Software Engineering Research, Limerick, (Ireland).

[ASU 2001] *"A Foundation for the Economic Analysis of Software Architectures"*, J. Asundi, R. Kazman, Third International Workshop on Economics-driven Software Engineering Research, Toronto (Canada).

[BOE 1999] *"A Method for Software Quality Planning, Control and Evaluation"*, J. Bøegh et al., IEEE Software, Vol. 16, No. 2, pp. 69-77.

[DRO 1996] *"Cornering the Chimera"*, R. G. Dromey, IEEE Software, Vol. 13, No. 1, pp. 33-43.

[DRO 2003] *"From Requirements to Design – Without Miracles"*, R.G. Dromey, ISCN, to appear in 2003.

[FEN 1998] *"Assessing Dependability of Safety Critical Systems using Diverse Evidence"*, N. Fenton et al, IEE Proceedings Software Engineering, 145(1), pp. 35-39.

[FEN 1999] *"A Critique of Software Defect Prediction Research"*, N. Fenton and M. Neil, IEEE Transactions on Software Engineering, Vol. 25, No. 5.

[GAR 1984] *"What Does 'Product Quality' Really Mean?"*, D. Garvin, Sloan Management Review, Fall 1984, pp. 25-45.

[GRA 1992] *"Practical Software Metrics for Project Management and Process Improvement"*, Robert B. Grady, Prentice Hall, pp. 22-24.

[HER 1999] *"Higher Customer Satisfaction with Prioritizing and Focused Software Quality Function Deployment"*, G. Herzwurm et al., Proceedings of the Sixth European Conference on Software Quality in Vienna (Austria).

[ISO 1991] *"ISO/IEC 9126: Information Technology - Software Product Evaluation - Quality characteristics and guidelines for their use"*, ISO.

[ISO 1994] *"ISO 8402 Quality Management and Quality Assurance – Vocabulary"*, 2nd edition.

[KIT 1996] *"Software Quality: The Elusive Target"*, B. Kitchenham, Shari L. Pfleeger, IEEE Software, Vol. 13, No. 1, pp. 12-21.

[LOS 2002] *"Standard quality model to design software architecture"*, F. Losavio, Journal of Object Technology, Vol. 1, No. 4, pp. 165-178.

[McC 1977] *"Factors in Software Quality"*, J.A. McCall et al., RADC TR-77-369, US Rome Air Development Center Reports.

[MOO 1995] *"Crossing the Chasm"*, G. Moore, Harperbusiness, New York.

[OHM 1993] *"Software quality deployment approach: framework design, methodology and example"*, Akira Ohmori, Software Quality Journal, No. 3, pp. 209-240.

[ROT 2002] *"What Does Success Look Like?"*, J. Rothman, http://www.jrothman.com/Papers.

[RTI 2002] *"The Economic Impacts of Inadequate Infrastructure for Software Testing"*, Planning Report 02-3, Prepared by RTI for National Institute of Standards and Technology, U.S. Department of Commerce.

[STA 1995] *"Study on project successes and failures"*, Standish Group Report.

[ZUL 1990] *"Software Quality [Function] Deployment. Applying QFD to software"*, Richard E. Zultner, Transactions from the Second Symposium on Quality Function Deployment, Novi, Michigan, pp. 132-149.

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; e-mail franco@martinig.ch ISSN 1023-4918
Free subscription: www.methodsandtools.com