Microsoft

Acceptance Test Engineering Guide

Volume I

How to Decide if Software is Ready for You and Your Customers

Grigori Melnik, Gerard Meszaros, Jon Bach

Foreword by Kent Beck

patterns & practices

BETA 2 RELEASE

COMMUNITY PREVIEW LICENSE

Table of Contents

Preface: Acceptance Test Engineering Guide

Why We Wrote This Guide

Accepting software is one of the most important decisions for software to take flight and to start delivering value. Yet, there exists little guidance about what this decision should be based on and how to make it effectively.

Over many years in industry and applied research, it was our observation that people often misunderstood the challenges of accepting software, or worse, did not even have a clear notion of who is in charge of acceptance. Many appear not to be aware of the arsenal of practices and tools available to them to address those challenges. This led to disappointments, failed opportunities, broken business relationships, and litigations. There exists a rich body of knowledge on software testing (with landmark works by Boris Bezier, Bertrand Meyer, Gerald Weinberg, Cem Kaner, James Bach, Brett Pettichord, Lee Copeland, Mark Fewster, Dorothy Graham, Rex Black, Brian Marick, James Whittaker to name a few), but software acceptance in general, and acceptance testing, in particular, as a process, received no serious, well-reasoned or comprehensive treatment. We set out to fill this gap with two key objectives: devise mental models for thinking about software acceptance and offer actionable guidance on achieving software acceptance.

Who Should Read This Guide

This guide is intended for anyone who is involved in the process of deciding the degree to which a software-intensive product meets the acceptance criteria of those who commissioned its construction. Specifically, this guide will help you if any of the following apply to you:

- You are involved in deciding whether to accept the software as it has been built. This is the *acceptance decision*.
- You are involved in deciding on what acceptance criteria shall be used to make the acceptance decision.
- You are involved in collecting data that the person making the acceptance decision requires to make that decision. This is *acceptance testing*.
- You are involved in deciding whether the product is ready to be seen by the people involved in the acceptance decision or acceptance testing. This is the *readiness decision*.

You are involved in collecting data that the person making the readiness decision requires to make that decision. This is *readiness assessment*.

You are involved in defining the expectations against which the readiness assessment or acceptance testing activities will be conducted. This is a combination of *requirements gathering, test planning* and *test design*.

You are involved in managing any of the preceding activities.

This guide describes the practices used by people in the preceding roles, regardless what your job title is. If any of them describes your role, you should find something of interest in this guide. Appendix – Reader Personas describes our target audience in more detail. These personas are used to:

remind the authors and reviewers of our primary target audience.

give you, the reader a figure to empathize or associate yourself with. This should help you pick the persona that most closely resembles your own job responsibilities.

Guide Structure

This guide is structured as a three volume series:

Volume I provides an overview of the acceptance process and how acceptance testing and other key practices fit into the process. This volume is intended to be read from beginning to end. It is subdivided into three main parts:

Part I – Thinking about Acceptance explains six mental models that are useful when thinking about the acceptance process.

Part II – Perspectives on Acceptance describes the acceptance process from the perspectives of key stakeholders in two different kinds of organizations: the Information Technology Department in a business and the Product Development Company. Most readers involved in the acceptance process should find some commonality with at least one of the roles describes.

Part III – Accepting Software introduces the practices that are necessary for planning the acceptance process, for performing acceptance testing and for improving the acceptance process.

Volume II is a collection of what we call *thumbnails* that describe the practices introduced in volume I in more detail. A thumbnail is a short overview of a practice that explains what it is, when you may want to use it, the risks that it mitigates, and an overview of how to perform the practice. Thumbnails also include a list of references to papers, books, and other resources that provide more complete descriptions of the practices in question. The main purpose of a thumbnail is to describe a topic well enough to provide an overview, serve as a mental reminder for someone who has used the practice on how to do it, and give someone

unfamiliar with the practice enough information about the practice and its applicability to determine if they want to learn more about it. Some of these topics and practices have entire books written about them that describe the concepts in greater detail and depth than this guide could possibly do. Volume II is intended to be used as a reference; most readers will not read it from beginning to end.

Volume III. This volume is a collection of sample artifacts generated by applying different practices in a fictional but realistic situation for the fictional company Global Bank. These artifacts are embedded in a series of case studies of what the Global Bank team may have produced while building the application. The case studies provide some context to the individual artifacts. They also provide cross-references to the practices described in Volume II. The artifacts are intended to be used as way to learn more about how to perform a practice; they can also be used as templates for your own artifacts.

How to Read This Guide

The way you approach this guide will depend on what role you have and what you want to learn about the process of accepting software. Depending on what you want to do, you will want to apply different strategies.

Get an Overview of Acceptance Practices and Processes

Start by reading Volume I if you want to do any or all of the following:

Learn general information about acceptance testing.

Find acceptance testing practices.

Create a project plan.

Justify a project plan.

Justify an approach used for acceptance testing.

Validate that you are on track with your acceptance testing strategy or approach.

Get your project un-stuck.

Determine where there may be gaps in your acceptance testing approach or strategy.

After reading Volume I, you may want to skim particular practices of interest in Volume II and the corresponding samples in Volume III.

Decide Which Acceptance Practices to Use on Your Project

Start by reading Volume I to get an overview of possible practices, and then refer to the thumbnails in Volume II for specific practices you are considering. Each thumbnail includes a section titled "When to Use It," which includes advice about when the practice should be used, and a section titled "Limitations," which provides guidelines about when the practice should not be applied.

Learn How to Perform a Specific Acceptance Practice

Start by finding a thumbnail in Volume II if you want to do any of the following:

- Learn a specific acceptance testing practice or strategy.
- Teach a specific acceptance testing practice or strategy to someone else.
- Review a specific acceptance testing practice.
- Find more information and related resources to consult about a particular practice.

After you locate the thumbnail for the specific practice you want to learn about, read it and any related samples in Volume III. If you need more detailed information about the practice, see the "References" section in the thumbnail.

Get a Template for a Specific Artifact

Start by finding an example in Volume III if you want to do any of the following:

- Find a template for a specific artifact.
- Learn how to fill in a specific artifact.

Find the example you want in Volume III, remove the sample information, and populate it appropriately. If you need to review the practice that generated the example, the example lists all the appropriate thumbnails to refer to in Volume II.

Plan the Execution of the Practices on Your Project

Start by reading Volume I to get an overview of how the practices fit together and support each other. In particular, the sections on the Acceptance Process, the Decision-Making Model, and the Doneness Model may be of particular interest. After that, review the specific thumbnails in Volume II, paying particular attention to the subsection, "Test Life Cycle Applicability" in the section, "When to Use It." In Volume III, each sample artifact is accompanied by a notation that indicates at what point in the hypothetical project the artifact was produced. Note that some artifacts appear at several points in the project timeline because they evolve over time. If you find your acceptance process inefficient, the Streamlining Acceptance Process chapter of Volume I may help.

Find Tools for Acceptance Testing

If you are looking for tools to perform acceptance testing, you may use the guide to explore the available space. Although some of the case studies illustrate using specific tools, remember that the primary focus of this guide is on describing practices.

The choice of tools used while producing this guide should not be interpreted as an endorsement of any tool, nor should it be interpreted as an indication that any tool used is the best one for the job. By the time you read this guide, the tools illustrated in the samples in the guide may have been supplanted by newer and better ones.

Introduction

This guide is about accepting software. Accepting software involves acceptance testing, but it is much more than that. The concept of acceptance testing means different things to different people. In simple terms, acceptance testing is the set of activities we perform to gather the information we need to make the decision “Is this software ready for me (or my customer)?” This decision is usually composed of several decisions, each with supporting activities. Therefore, to define acceptance testing, it may be useful to understand the process by which the decision(s) are made. This process may involve several organizational entities, each with one or more decision-makers. The software is typically passed between the organizational entities for them to decide whether the software is ready to go through the next step . This process is introduced in more detail in the section "Acceptance Process Model" and followed up with a more detailed description of the decision-making process in the section called "Decision Making Model."

Software Acceptance and Acceptance Testing

Acceptance refers to the act of determining whether a piece of software or a system meets the product owners’ expectations. It includes both the final decision to accept the software and any activities, including acceptance testing, required to collect the data on which the acceptance decision is based. Both the acceptance testing and the acceptance decision can be relegated to a separate acceptance phase of the project or they can be done throughout the project, which is known as Incremental Acceptance Testing.

Mental Models for Acceptance Testing

While writing this guide, we struggled to determine a suitable definition of acceptance testing that would make sense to a broad range of readers. It seemed like there were many different vocabularies in use by different communities such as consumer product companies, information technology departments of large businesses, and data processing divisions of telecommunication service providers, to name just a few. To assist us in describing “acceptance”, we came up with several mental models of various aspects of acceptance testing. We tested the models against numerous examples from our collective project experiences at Microsoft patterns & practices, telecommunication product companies, IT departments and beyond. Then we tested the models with the people on the Advisory Board for the project. This was an iterative process. We also tested these through the public review process by releasing early drafts of this guide to the community and soliciting feedback.

It is important to note that our early models failed their acceptance tests! That was a great lesson about the need to get feedback incrementally, a practice we advocate for acceptance testing. Based on feedback from our advisors we refactored the models and came up with additional models to fill the gaps. The key breakthrough was when we came up with the Decision-Making Model, which ties together

most of the concepts around accepting a system. It builds on the Acceptance Process Model which describes the key steps and activities as the system-under-test moves from requirements, through development and into testing and finally production; it also describes how the decision to accept the system is made. The Decision-Making model describes who makes the decisions and who provides the data for those decisions.

The decisions are not made in a vacuum; there are a number of inputs. These include the project context, the nature of the system being built and the process being used to build it. The latter is important because it affects how we define “done”.

Figure1 illustrates the rel*a*tionships between the k*e*y models.

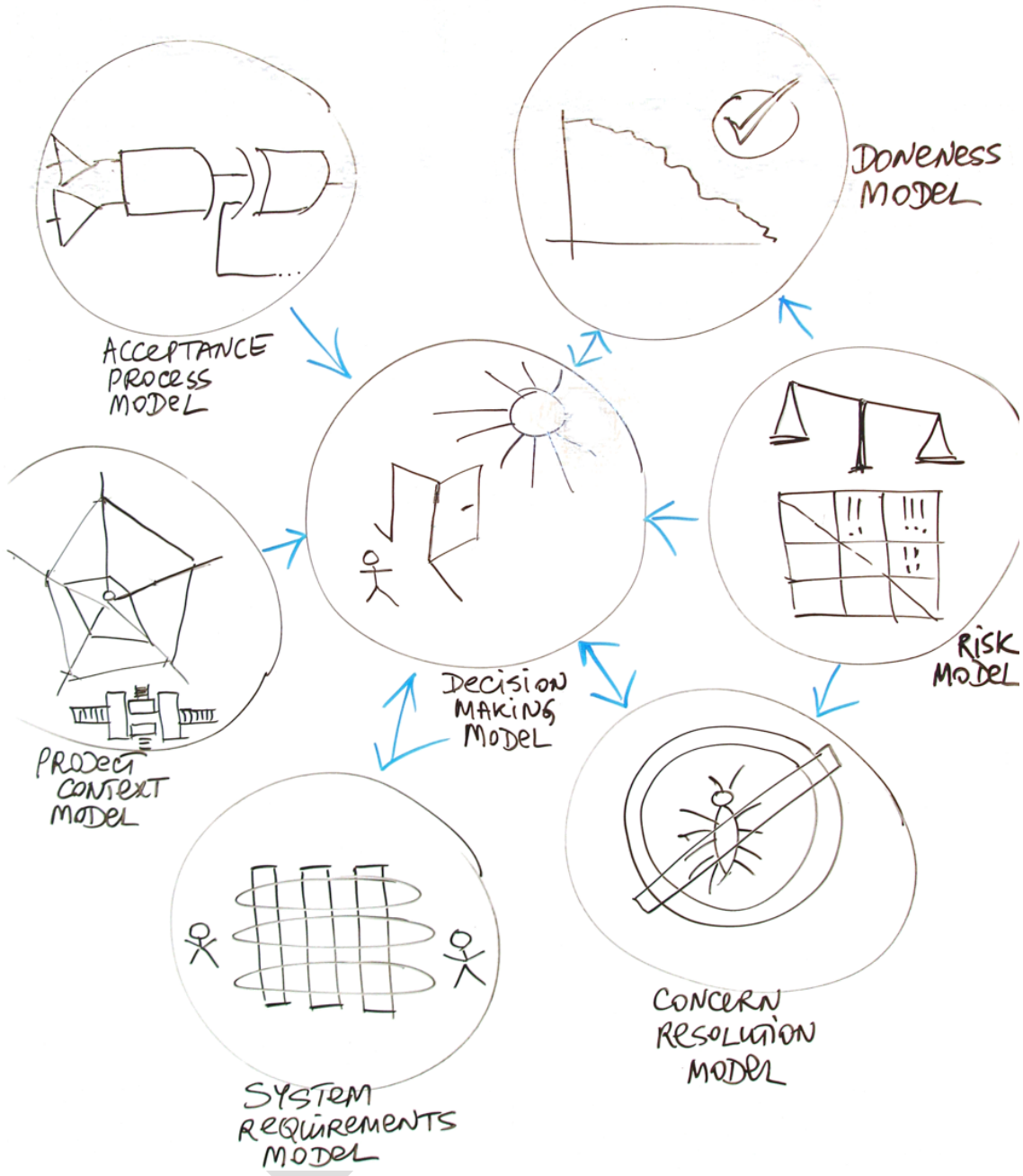


Figure 1

The Key Mental Models of Acceptance

These models are the focus of Part I – Thinking about Acceptance but here's a short introduction to each model to get us started:

The Acceptance Process Model. This model defines the overall stages of software development and the "gates" that must be passed through on the journey from construction to software-in-use.

Decision-Making Model. This model describes how to decide whether software can go through a gate to the next stage and who makes the decision. It also defines the supporting roles that may help the decision maker gather the information needed to make the decision.

Project Context Model. This model describes the business and project factors that influence the decision, including timeframes and deadlines, resource availability, budget, and anything contributing to project risks.

System Model. This model describes the attributes of the software-intensive system that may factor into the decision. This includes both functional and parafunctional attributes. The system model may arise out of requirements gathering activities or a more formal product design process. Techniques for capturing functional requirements include simple prose, use case models, protocol specifications and feature lists. The parafunctional (aka non-functional) requirements are typically captured in documents or checklists.

Risk Model. This model introduces the concepts of events, likelihood/probability, and consequence/impact. It helps everyone involved understand what could go wrong and, thereby, prioritize the acceptance criteria and the kinds of information to gather to help make the acceptance decision. It also describes several different risk mitigation strategies, including the following:

Do something earlier to buy reaction time.

Do additional activities to reduce likelihood of something occurring.

Doneness Model. This model describes different dimensions of "doneness" that need to be considered in a release decision and how they are affected by the process model.

The chapters in Part III – Accepting Software introduce other models that build on this core model:

Test Lifecycle Model. This describes the stages that an individual test case goes through how to gather information for making readiness and acceptance decisions.

Concern Resolution Model. This describes how to handle any concerns that are raised during the acceptance testing process.

Development Processes

The software development process has a significant impact on how acceptance testing is performed. Throughout the rest of this volume and the others to follow we found ourselves saying "On waterfall projects…" and "On agile projects …" but many people have their own definitions of what these terms mean. We wanted to make sure all readers understood what we meant by these terms. We feel that these names refer to points on a process continuum with other labelled points possible. This section describes the process continuum with two distinct process stereotypes on the opposite ends of the scale.

Waterfall/Tayloristic Processes

Classic Waterfall

The waterfall approach (also known as the Tayloristic, document-driven or plan-driven approach) involves organizing the project into a series of distinct phases. Each phase contains a specific type of work (such as requirements analysis) and has specific entry and exit criteria. Ideally, the phases do not overlap. The entry and exit criteria synchronize the activities delivering the functionality to cause them to occur at pretty much the same time. Figure 2 illustrates the major phases of a waterfall project.

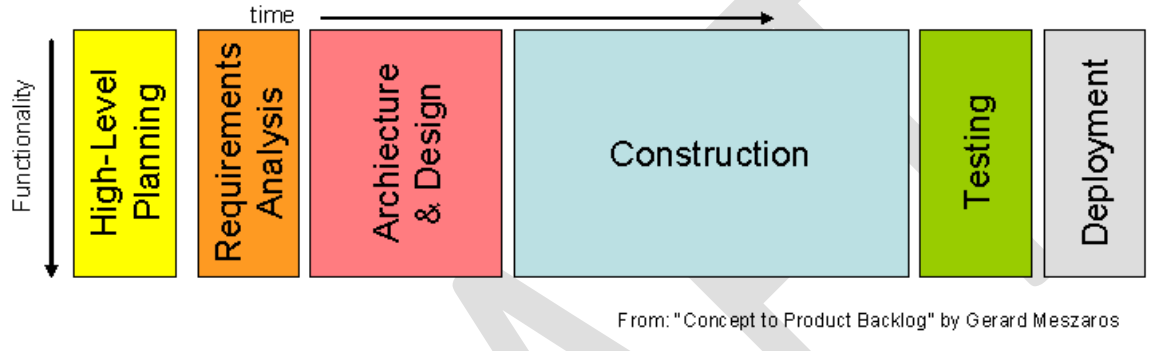


Figure 2

A Classical Waterfall Project

Each phase is broken out into a work breakdown structure appropriate to the type of work involved. For example, within the requirements phase, the work may be divided between analysts by requirement topic, but during the construction phase, work may be divided among the developers by module. The handoffs between phases are usually in the form of documents, except that the handoff from construction to testing also involves the code base. Readiness assessment is done by the supplier organization after all the construction is completed; acceptance testing is performed by the product owner after the software is deemed to be ready.

Incremental Waterfall

It is commonly accepted that the longer a project goes before delivering software, the higher the probability of failure. One way to combat this is to break the project into increments of functional . These increments can be tested and in some cases even deployed. Figure 3a illustrates a waterfall project with two increments of independent functionality each of which is tested and deployed (Incremental Waterfall, a.k.a Checkpointed Waterfall).

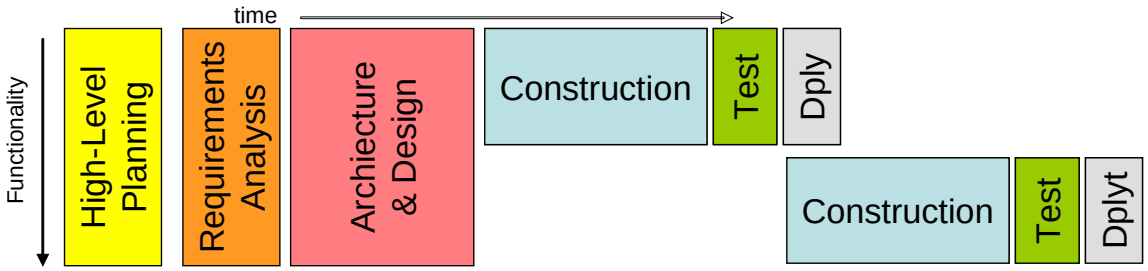


Figure 3a

Multi-release waterfall with independent functionality

In this approach, the planning phase, requirements analysis phase, and design phase are performed once early in the project while the construction phase, test phase, and deployment phase are repeated several times. The work within each phase is decomposed the same way as for single-release projects. If the functionality built in the second release overlaps the functionality in the first release, the testing and deployment must encompass the entire functionality. Figure 3b illustrates multiple releases with overlapping functionality. Note how the test activity must span the functionality of both releases.

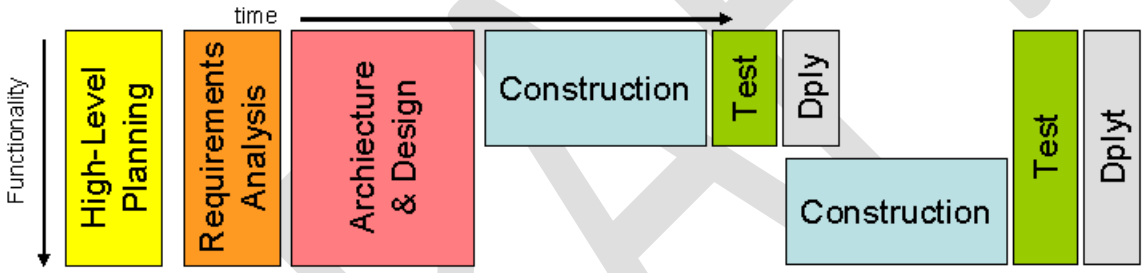


Figure 3b

Multi-release waterfall with overlapping functionality

Agile Processes

Most agile methods use an iterative and incremental approach to development. After an initial planning period, the project duration is broken into development iterations that deliver increments of working software. Figure 4 illustrates a project with two iterations; most projects would have many more iterations than this.

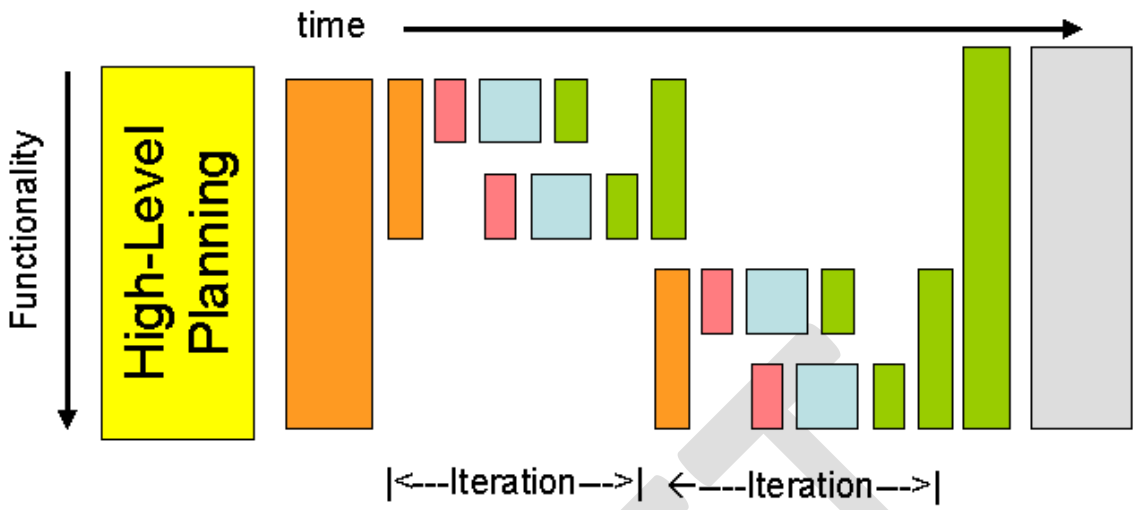


Figure 4

Iterative & Incremental Development

Figure 4 illustrates two iterations, each of which starts with an iteration planning session and ends with some acceptance testing. In each iteration the work is broken down into features or user stories, each of which independently goes through the entire software development life cycle. The product owner, "onsite customer" or customer proxy, who is readily accessible to the development team, is responsible for describing the details of the requirements to the developers. It is also the product owner's responsibility to define the acceptance tests for each feature or user story. They provide these tests to the developers as a more detailed version of the requirements description in a process known as "Acceptance Test Driven Development" or "Storytest-Driven Development." This allows the developers to execute the acceptance tests during the development cycle.

When all the tests pass for a particular feature or user story, the developers turn over the functionality to the product owner (or proxy) for immediate "incremental acceptance testing." If the product owner finds bug, the developer fixes it as soon as possible, preferably in the same iteration. Only when the product owner accepts the feature as built is the developer allowed to start working on another feature.

Multi-Release Agile Projects

Most agile methods advocate "deliver early, deliver often." In theory, the result of any development iteration could be determined, after the fact, to be sufficient to be put into production. This would lead directly to the deployment activities. In practice, most agile projects plan on more than one release to production and the iterations are then planned to deliver the necessary functionality. Figure 6 - Multi-Release Agile Project illustrates an agile project with two releases.

Figure 5

Multi-Release Agile Project.

Note how there is a testing cycle for the second release which includes regression testing of the functionality delivered in the first release.

Kanban-based Agile Process

Some agile methodologies dispense with iterations in favour of allowing a fixed number of features in progress at any time. This is designed to emphasize the concept of a continuous flow of working code for the on-site product owner (or proxy) to accept. From an acceptance testing perspective, these Kanban-based methods still do incremental acceptance testing at the feature level and formal/final acceptance testing before each release, but there is no logical point at which to trigger the interim acceptance testing that would have been done at iteration's end in iteration-based agile methods. Figure 6 – Kanban-based Agile Project illustrates this style of development. Note the lack of iterations.

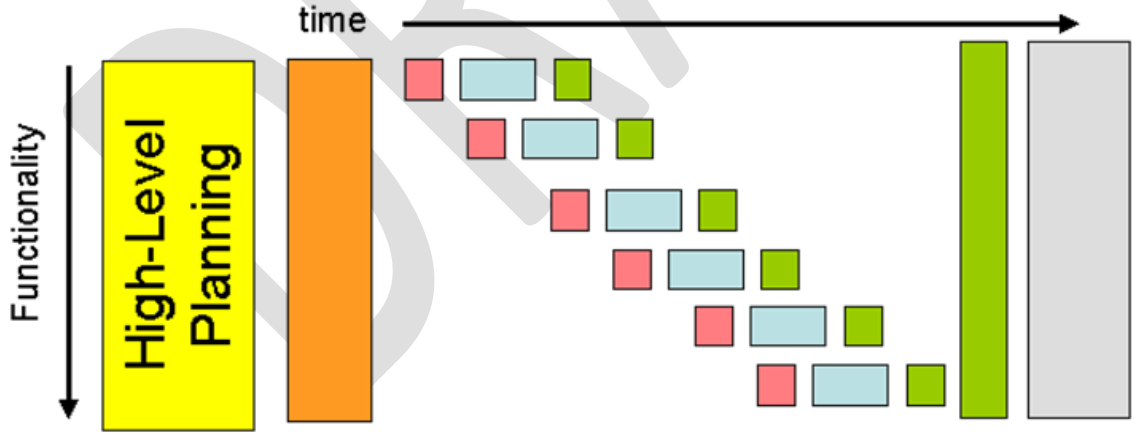


Figure 6

Kanban-based Agile Process

It is important to note here that there are never more than three features in progress at any one time. In other words, there are only three development "slots," and a slot becomes available for another feature only after it has finished its incremental acceptance testing. This is similar to how Kanban are used to control the inventory in lean factory production lines. In theory, the product owner can decide

at any time that there is enough functionality to warrant deploying the product following a short period of regression testing.

Process as a Continuum

Although "agile" and "waterfall" are two named styles of projects, they really are just named stereotypes consisting of certain combinations of characteristics. It is easy to imagine the decision on each of these characteristics as being the setting of a process "slider control". For example, the "Number of Releases slider" might have stops at 1, 2, 3, and so on. The Iteration slider could have values of 1, 2, 3, and so on, which indicate whether there are intermediate checkpoints or values of -1, -2, -3 indicating the number of development slots available in a Kanban-based system. Another dimension might be Integration Frequency, with settings of Big Bang, Major Milestone, Quarterly, Monthly, Biweekly, Weekly, and Daily.

The following table summarizes the positions of these sliders for what is considered to be a stereotypical project of each kind. These positions are not definitive or complete, but they challenge you to create your own sliders and settings for your context.

	Type of Process			
Project Attributes	Pure Waterfall	Incremental Waterfall	Agile (Iteration)	Agile (Kanban)
Number of releases	1	1	2 or more	2 or more
Number of iterations	1	2–6	4 or more	1
Iteration length	Not applicable	Many months	1-4 weeks	Not applicable
Maximum number of features in progress	No maximum	No maximum	1 iteration's worth	Less than the number of team members
Integration frequency	Big Bang	Quarterly	Daily or hourly	Daily or hourly
Requirement-to-test duration	Months or years	Months	Days	Days
Test timing	Separate phase	Separate phase	Mostly incremental	Mostly incremental
Release criteria	Scope-based	Scope-based	Time-boxed	Time-boxed
Average Requirement task effort	Person months	Person months	Person days	Person days
Average development task effort	Person days or weeks	Person days or weeks	Person hours	Person hours
Work style	Tayloristic	Tayloristic	Collaborative	Collaborative
Skills	Highly specialized	Highly specialized	Generalists	Generalists
Determining progress	Earned value	Earned value	True value	True value

	Type of Process			
Project Attributes	Pure Waterfall	Incremental Waterfall	Agile (Iteration)	Agile (Kanban)
	calculated based on WBS	calculated based on WBS	delivered in working code	delivered in working code
Work remaining	Estimate duration of remaining tasks	Estimate duration of remaining tasks	Estimated time for remaining features	Estimated time for remaining features

A Cautionary Tale

The following is the story a project where the person in charge of defining and accepting a product fails to rise to the challenge. It describes what goes wrong and why. It also provides an alternate outcome of how the project could have worked out if proper acceptance practices had been used.

Bob Kelly is a mid-level manager in the marketing department at Acme Products. He is in charge of the Mid Sized Markets product group and is the product manager of its core product, the XCelRator. He's just finished a bunch of market research and has come up with a great idea for a new module for the product that he believes will double the revenue of the product. The Product Development division has a team that is just winding down its current project and Bob is keen to get the team started on building the new module. He calls a meeting and runs through the PowerPoint slide deck he's been using when talking with focus groups and potential clients. He concludes the presentation by laying out the key delivery dates which must be met to allow the company to showcase the product at the annual trade show. He asks the project manager to draw up the project plan and get the team working on the new module.

Dev Manager meets with his team to collect some information and then he defines the project plan. Based on the delivery date he defines intermediate milestones for Requirements Complete, Design Complete, Code Complete and Test Complete.

Team starts writing requirements documents. At Req'ts Complete, dev manager declares requirements complete on time. The dev team knows full well that some areas of the requirements are still too vague to implement. They will have to get clarification on the details as they do the design. Hopefully the product manager will be more available to answer questions about the requirements as his lack of availability is at least partially to blame for the requirements still being vague.

At Design Complete, ditto. Dev team knows it hasn't designed some parts of the functionality except at a very cursory level. They will have to fill in the details as they code. Meanwhile, the product manager thinks everything is proceeding according to plan and that he'll have a great product to demo at the trade show. He starts thinking about how he'll spend the bonus he's sure to get for completing this project on time. He calls his architect and asks her to start drawing up plans for the extension to his house, complete with indoor/outdoor swimming pool.

At code complete the team requests an extra month to finish coding parts of the functionality they haven't had time to do yet. The Test Manager asks what parts of the software are complete so that testers can start testing and the team responds "None, we are each working on a different part of the system and it will all be finished at about the same time."

To ensure the product is ready for the trade show, the product manager asks the test manager to make up the schedule by compressing the duration of the test phase. He reduces the number of planned test cycles to reduce the elapsed time based on assurances that the code will be in great shape due to the extra time the dev team is being given. The product manager still believes the product can be ready for the trade show but nonetheless he asks the architect to scale back the extension to his house by removing the enclosure for the pool. “I’ll still be able to use it most of the year and I can always add the enclosure with my release 2 bonus.”

As the new Code Complete deadline approaches, the team asks for another month. Product owner reluctantly gives them 2 weeks. “Two weeks later will make it tight for the trade show but we can use the beta so it won’t affect my bonus too badly.” he hopes.

The dev team finally delivers the code two months late. The test team starts testing but finds significant problems that prevent completion of the first pass of the test cases. Testing halts after less than a week. Dev team takes a month to fix all the major show-stopper bugs before handing the code back to test. Test team makes it through the full test cycle this time but finds several hundred defects. Development starts fixing the bugs as they are found.

Test team finally accepts a new build with the Sev 1 and 2 bug fixes. Almost immediately they find several new Sev 1 regression bugs. Development disagrees that some of the bugs are even valid requirements. “The requirements never said anything about …” and “Why would anyone ever do that?” they exclaim. The product manager has to be brought in to settle the dispute. He agrees that some of the bugs aren’t real requirements but most are valid scenarios that he never considered but which need to be supported. Test tells the PO that there is no way in … that the original schedule can be met.

After 4 test&fix cycles that took 50% longer than the original schedule (let alone “making up the schedule”), most of the Sev 1 and 2 bugs appear to have been fixed. There is still several hundred Sev 3 and 4 bugs and Test has stopped bothering to log the Sev 5’s (poorly worded messages, field labels or button names, etc.). Test says the product is not ready to ship and will require at least 2 more test&fix cycles before they will agree to release it.

The product is now several months late and won’t be ready for the trade show even in alpha release. “We can still show the box and do some very controlled demo’s.” the product manager assures his boss. Bob is seeing his bonus diminish with each week of reduced sales in this fiscal year. He decides to ship the product, overruling the Test department. He revises the sales forecasts based on the late launch caused by the “underperformance of the development and test teams.”

The manager of the operations department hears about this too late and comes storming into Bob’s office. “I hear you are planning to release the new module with 300 known Severity 3 defects. Do you have any idea what this will do to our support costs?” A counterproductive argument ensues because there is no turning back at this point; the announcements have been made and to change plans now would be hugely embarrassing to the company. “I sure hope this works out OK.” thinks Bob to himself; at this point he doesn’t feel like he’s in charge of his own destiny.

Bob's marketing machine has been in high gear for quite a while and has generated quite a bit of pent up demand, especially since some users were hoping to be using the product months ago. Users try using the new product in droves. They run into all manner of problems and call the Help line which is overwhelmed by the call volumes. Many users get busy signals; the lucky ones wait listening to recorded announcements for long periods of time. The help desk has to bring on extra staff who need to be trained very hastily and therefore cannot provide very good service to the customers. Some customers give up trying to use the product because of long waits or poor support.

Many of the user problems are related to usability issues that were not detected during the rushed testing because it was so focused on the Sev 1 & 2 bugs (and the usability stuff was usually rate 3 or below, many of which weren't even logged due to the focus on the 1's and 2's.) At peak usage times the system slows to a crawl; the development team is called in to figure out why it is so slow distracting them from working on some of the improvements identified during testing. Users have trouble importing data from prior versions of the software or from competitors' products.

A large percentage of the users abandon the product after the free trial is over; the conversion rate is less than half of the projected rate. Revenues are running at 40% of the revised projections and less than 20% of the original projections. Support costs are 50% over original projections due to the hiring binge in the user support centre.

The capital cost is 35% over the original budget and has eaten into the planned budget for a second module with additional must-have functionality. Instead, the 2nd release has to focus on improving the quality. The new module will have to wait until 2nd half of next year. The product manager revises the revenue projections yet again and calls the contractor to cancel the addition to his house. Shortly after sending in his monthly status report his phone rings. It is the VP, his boss, "requesting" he come to his office immediately...

What Went Wrong

1. Overcommitted functionality at product manager's insistence. (Sometimes the dev team will over commit out of optimism but in this case the product manager did it to them.)
2. Waterfall process is inherently opaque from a progress perspective. The first milestone that isn't easy to fudge is Code Complete. The first realistic assessment of progress is well into the test cycle.
3. Product manager wasn't available to clarify vague and missing requirements. Testers were not involved soon enough to identify the test scenarios that would have highlighted the missing requirements. But no one could prove the requirements were incomplete so RC was declared on time.
4. Dev team couldn't prove design wasn't done (because it is a matter of opinion as to how detailed the design needs to be) so Design Complete was declared on time.

5. Dev team cut corners to make the new (late) Code Complete deadline. The code was written but much of it wasn't properly unit tested. They knew this and would have told anyone who asked but no one wanted to hear the answer.
6. The quality was awful when delivered to Test. So it had to be redone (we never have time to do it right but we always make time to do it over!)
7. Test was asked to recover the schedule (typical!) but testing took longer because the poor quality code required more test&fix cycles to get it into shape.
8. No clear definition of "done" so decision is made on the fly when emotions get in the way of clear thinking. The product manager let his attachement to his commitments override any sensibility about quality.
9. The operations acceptance criteria were never solicited and by the time they were known it was too late to address them.
10. Waterfall process hid true progress (velocity) until it was too late to recover. There was no way to scale back functionality by the time it became undeniable that it won't all be done on time. There was no way to reduce scope to fit the timeline because the waterfall-style approach to the project plan (RC, DC, CC, TC milestones) caused all features to be at roughly the same stage of development. Therefore cutting any scope would result in a large waste of effort and very little savings of elapsed time.
11. Development was rushed in fixing the Sev 1 problems so that testing could get through at least one full test cycle. This caused them to make mistakes and introduce regression bugs. It took several test&fix cycles just to fix the original Sev 1&2's and the resulting regression bugs. In the meantime the Sev 3's piled up and up and up. This resulted in several more test&fix cycles to fix and even then more than half were still outstanding.
12. Lack of planning for the usage phase of the project resulted in a poor customer support experience which exacerbated the poor product quality.

How it Could Have Gone

The product manager comes to Dev team with requirements that are representative of the clients' expectations..

Dev team estimates requirements as being 50% over teams capability based on demonstrated development velocity. The product manager is not happy about this. Dev Team proposes incremental development & acceptance testing. Instead of 4 waterfall milestones they suggest 4 incremental (internal) releases of functionality where each increment can be tested properly.

The product manager selects first increment of functionality to develop. He defines the user model consisting of user personas and tasks. Dev team whips up some sketches or paper prototypes and helps product manager run some Wizard of Oz tests that reveal some usability issues. The product manager

adjusts the requirements and dev team adjusts the UI design. The product manager works with dev team and the testers to define the acceptance tests. The dev team automates the tests so they can be run on demand. They also break down the features into user stories that each take just a few days to design and test.

Team designs software and writes code using test-driven development. All code is properly unit-tested as it is written. All previously defined automated tests are rerun several times a day to make sure no regression bugs are introduced as the new feature is implemented. As each feature is finished, as defined by the feature-level "done-done" checklist , the developer demos to product manager and tester who can point out any obviously missing functionality that needs to be fixed before the software is considered "ready for acceptance testing".

As part of incremental acceptance testing they do identify a few new usage scenarios that were not part of the requirements and provide these to the product manager as suggestions for potential inclusion in the subsequent increments. The product manager adjusts the content of the next increment by including a few of the more critical items and removing an equivalent amount of functionality. He also contacts the operations manager to validate some of the operational usage scenarios identified by the dev team and testers. The operations manager suggests a few additional operational user stories which the product manager adds to the feature backlog for the next increment.

At the end of first increment of functionality (which took 5 iterations to develop, one more than expected) dev team runs the para-functional tests to verify the software performs up to expectations even with 110% of the rated numbers of users. The first test results indicate it can only handle roughly 50% of the expected users and gets even slower as the database starts to accumulate records. They add some work items to the schedule for the second increment to address these issues and warn testing about the limitations to avoid their wasting time stumbling onto them. Testing finds only a few minor bugs during execution of the functional test scripts (the major stuff was all caught during incremental acceptance testing.) They move on to doing some exploratory testing using soap operas and scenarios as their test session charters. These tests identify several potential scenarios the product manager never thought of; he adds them to the feature backlog.

The product manager arranges to do some usability testing of the first increment with some friendly users based on some of the usage scenarios identified in the user model. The results of the testing identify several enhancements that would improve the user satisfaction. The product manager adds these to the things to do in the next increment of functionality.

The product manager calculates that demonstrated development velocity is 25% less than original estimates. Based on this he adjusts his expectations for the functionality to be delivered in the release by removing some of the less critical features and "thinning" some of the critical features by removing some nice-to-have glitz. "Better to deliver good quality, on time than to try to cram in extra functionality and risk everything" he thinks to himself.

The development team delivers the 2nd increment of functionality with similar results as the first. The work they did to improve performance results in the performance tests passing with flying colors with acceptable response times at 120% of rated capacity and no degradation as the database fills up with

transactional data. They add some tests for penetration testing and schedule a security review with the security department. The product manager makes some adjustments to the functionality planned for the 3rd increment of functionality. He includes some functionality to address operational requirements such as migrating data from earlier versions of the software and importing data from competitor's products; he wants to make it real easy for users to adopt his product. He's happy with how the project is progressing and confident that they will be able to deliver excellent quality and on schedule.

In the third increment the development team delivers 20% more functionality than originally planned. The product manager had to scramble to provide the acceptance tests for the extra features brought forward from the fourth increment. Based on this, the product manager is able to plan for some extra functionality for the fourth increment. He considers reviving some of the functionality that he'd cut after the first increment but decides it really wasn't that useful; a much better use of the dev teams efforts would be some of the usability enhancements suggested by the last round of usability testing. He also adds functionality to make it easy to upgrade to the next (yet unplanned) release without having to take down the server. That will help reduce the operational costs of the software. "Yes, this is going to a great product!" he says to himself.

As the development team is working on Increment 4, the product manager discusses the Acceptance Test Phase of the project. "We had originally planned 3 full test&fix cycles each of 2 weeks duration with a week for fixes in between for a total of 8 weeks of testing." recounts the Test Manager. "But based on the results of testing Increments 1, 2 and 3 I'm predicting that we'll only need one full test cycle of 2 weeks plus a 1 week mini-cycle of regression testing for any fixes that need to be done (and I'm not expecting many.) The automated regression testing the dev team is doing as part of readiness assessment has been preventing the introduction of many regression bugs and the automated story tests we've been co-authoring with you and the dev team has prevented misunderstandings of the requirements. This is the most confident I've ever felt about a product at this point in the project!"

Summary

There are good ways to effectively define, build and accept products and there are ways that may result in disappointment. The rest of this guide is dedicated to helping you understand which is which.

Part I - Thinking About Acceptance

A key part of understanding any system, whether we are building it, testing it, accepting or using it is to build mental models about how we expect it to work. Systems that are easy to use make it easy for users to develop simple mental models or at least familiar mental models that allow them to reuse knowledge about how other systems work. This part of the book focuses on describing six relatively simple models that help us reason about the process of accepting software.

Chapter 1 – The Acceptance Process introduces the overall acceptance process. It describes the decisions that must be made and how acceptance relates to the overall software development life cycle (SDLC) , the Stage-Gate process, and the classic V-model of testing.

Chapter 2 – The Decision-Making Model describes the process for making the decisions introduced in Chapter 1. It also provides guidance on who (which roles) should make the decisions and who should collect the information on which the decisions are based.

Chapter 3 – The Project Context Model describes the various project factors that might influence the acceptance process.

Chapter 4 – The System Requirements Model describes various ways requirements can be partitioned and described and how that relates to the acceptance process.

Chapter 5 – The Risk Model describes a simple way of thinking about potentially undesirable events and how they might influence the acceptance process.

Chapter 6 – The Doneness Model describes ways to describe to what degree the product is finished. It gives us ways to ask the question “what does done look like?” as a way to define our acceptance criteria for individual features and for entire releases of products. It also introduces the concept of incremental acceptance.

Chapter 1 The Acceptance Process

This chapter defines the process by which software is deemed acceptable by the customer. It introduces the three major constituencies who must make decisions at key points in the process and how these decisions relate to, and influence, each other. We start by drilling into the Accept phase of the software development lifecycle to examine the key decisions and how release candidates flow through the process. Then we examine how more complex scenarios such as multiple releases, complex organizations and complex products influence the process. Sidebars examine how the decision process relates to Stage-Gate processes and Quality Gates. We finish this chapter with techniques for streamlining the acceptance process. Subsequent chapters describe the roles and responsibilities of various parties involved in making the decisions and how this process varies between waterfall and agile projects.

1.1 Acceptance as Part of the Product Development Lifecycle

Software-intensive systems go through a number of stages on their journey from concept, a half-formed idea in someone's mind, to providing value to its users. This process is illustrated in Figure 1 – The Product Development Lifecycle with the stages placed into "swim lanes" based on which party has primary responsibility for the stage.

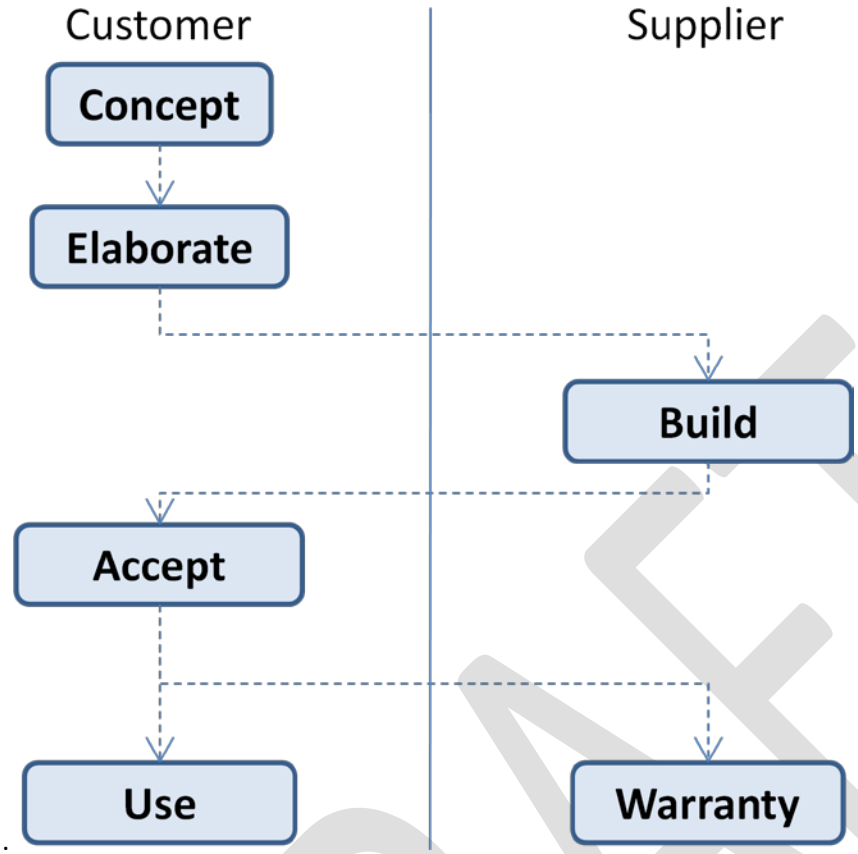


Figure 1

The Product Development Lifecycle

A common scenario is for a business person to have an idea for how to solve a business problem with software; this is the concept. They elaborate on the concept to produce a specification of the product that once built will deliver value. The supplier builds software to satisfy the specification even though it is well known that focussing on the specification instead of customer satisfaction often leads to the *wrong-product-built* phenomenon. The customer then assesses how well it satisfies the needs of the intended users and if satisfied, accepts the software. When the software is deemed acceptable, the software is made available to the users who then use the software to realize the intended value that solves the originally identified business problem. The supplier is usually obligated to provide support for a warranty period.

There are several different ways to describe the role of testing and acceptance in the development lifecycle of a software-intensive product. Some of the more commonly used models are:

- The phased (or Stage-Gate) model.
- The V-Model.

The first describes where testing fits within the overall lifecycle of software development while the latter focuses on describing the relationships between different kinds of testing.

1.1.1 The Stage-Gate Process™

Many projects adopt a Stage-Gate process [Ref] that treats each of these stages as mutually exclusive with gate decisions between each stage. Figure 1b – The Stage-Gate Process shows this process laid out as a Gantt chart with milestones representing the gate decisions between each stage.

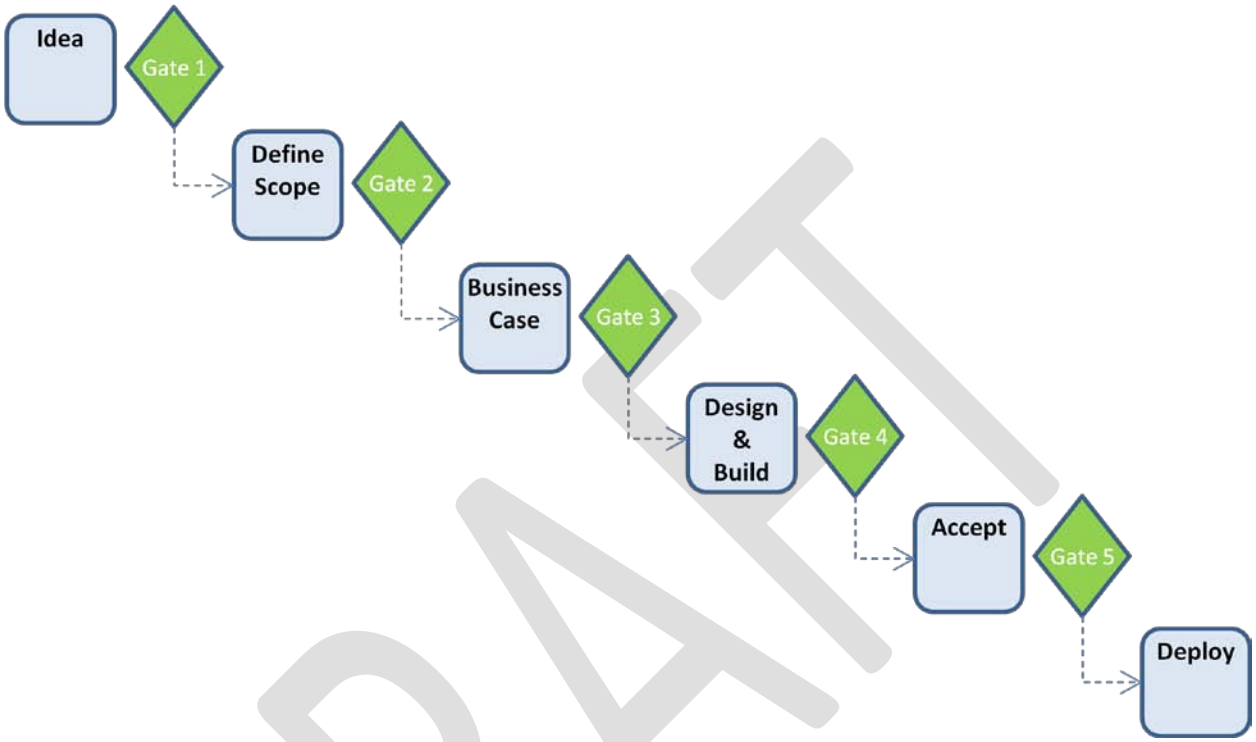


Figure 1b

The Stage-Gate Process

The stages depicted in Figure 1b represent stages of the project. A project can only be in on stage at a time but there can be many kinds of activities happening at the same time. The criteria for passing from one stage of the Stage-Gate process to the next need only be satisfied once; it is a classic waterfall with very little opportunity go back upstream.

The acceptance process that we describe in this book describes many activities and decisions that are made many times throughout the lifetime of a project. For example, the supplier may build many release candidates for testing but only a few are every accepted. Each release candidate is assessed for whether it satisfies functional and para-functional criteria. The transitions from the Design & Build stage to the Accept and Deploy stages are business decisions and involves many criteria that may have very little to do with whether the software meets the functional and para-functional requirements (See Chapter X – System Requirements Model). Examples include:

- A need to free up the development team for other work
- Change who pays for changes; before acceptance it is usually the supplier while after acceptance it is typically the customer.

- Change the funding source from capital to expense.
- Satisfy a milestone to provide an illusion of success despite failing to meet real objectives.

For example, the project could be in the Deploy stage (having passed through the gate after Accept phase --Gate 5 in Figure 1b--) even though the supplier is working on providing another release candidate for acceptance testing. Care has to be taken to avoid decoupling the acceptance process from the stage-gate process to such a degree that it renders the stage-gate process irrelevant. This guide is focussed on how to do acceptance well and tries to skirt around such issues which may be considered primarily political.

1.1.2 The V-Model of Software Development

A common way of describing the relationship of the various kinds of testing done on software-intensive systems is the V-Model [Ref] shown in Figure 1c – The V-Model of Software Development.

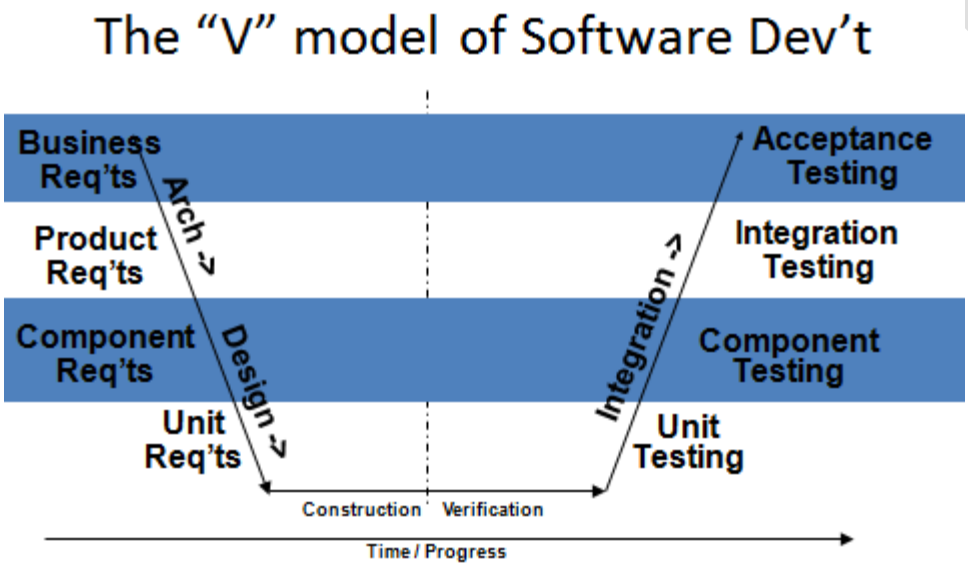


Figure 1c
The V-Model of Software Development

The system requirements are used as input into the design process that result in the overall architecture. This gets turned into component requirements which are then used as input into the detailed design of individual units of the software. As the units are implemented, we do unit testing. As the units are integrated into components we climb up the right side of the V to do component testing, then integration testing. When all these tests pass, we move up the final step to the top right side of the V to do acceptance testing.

The acceptance process describes what happens as we go up the right side of the V-Model. We make the decision to accept or reject the software based on how well it meets the requirements described at the top right of the V-model. When applied in a waterfall way, as part of a Stage-Gate™ style process,

we call this approach to acceptance testing the Acceptance Test Phase. Agile enthusiasts might refer to it test-last acceptance to contrast it with Acceptance Test-Driven Development and Incremental Acceptance Testing. These practices are described in section The Acceptance Process for Highly Incremental Development later in this chapter.

1.2 The Basic Acceptance Process

Each of the stages shown in Figure 1 involves many activities and decisions. While the Accept stage in Figure 1 looks like a single decision event that takes us directly from the Build stage to Use stage, it is actually a process involving a sequence of assessment and decision-making activities as shown in Figure 2 – The Acceptance Process.

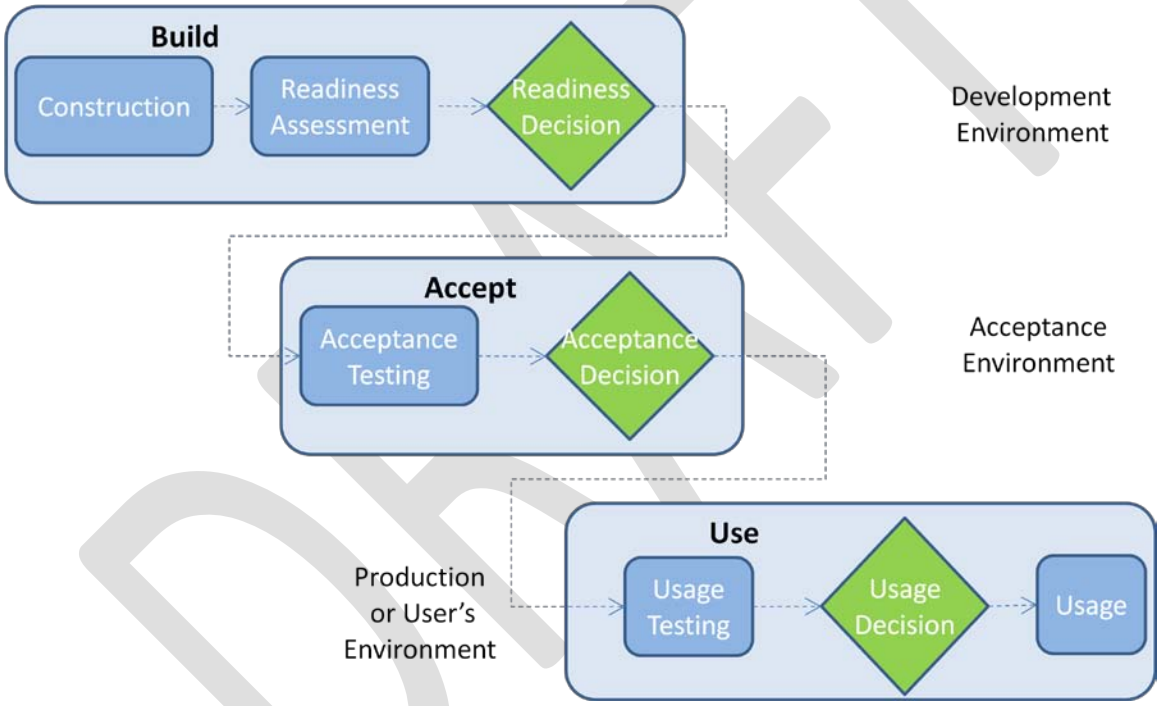


Figure 2
The Acceptance Process

In the ideal world this sequence of activities would be done exactly once and the assessment could be done entirely by the customer; in practice, this would be a recipe for disaster as untested software can contain large numbers of bugs per thousand lines of code. It usually takes many tries to build a product that the customer finds acceptable therefore the acceptance process is traversed, at least partially, many times. To minimize the number of times the customer is asked to accept the product, most suppliers will include some level of self-assessment of each release candidate before providing the software to the customer for making the acceptance decision. In this book we call the testing and other verification activities that are performed prior to the handoff to the customer *readiness assessment* and

the decision to hand off the software is called the *readiness decision*. The testing done by the customer or their proxy after the handoff from the supplier is called *acceptance testing* and the decision whether to accept the software in its current state is called the *acceptance decision*. The decision each user makes whether or not to actually use the product once it is available to them is called the *usage decision*.

Each decision can result in a positive or negative outcome. Only the positive outcomes are shown in Figure 2; the negative outcomes are shown in Figure 3 - Paths through the Acceptance Process.

The transitions from the Build stage to the Accept stage, and from the Accept stage to the Use stage typically require the movement of software (or software containing products) from one environment to another. For the purposes of this discussion, the specific steps involved in making the software available, though important for acceptance, are not considered a part of the acceptance process.

1.2.1 Assessing Release Candidates using the Acceptance Process

The version of the software that is put through the acceptance process is often referred to as a *release candidate*. It goes through the readiness decision and acceptance decision processes step by step and decision by decision until it meets one of the following requirements:

- It passes through all the decisions and is deemed ready for use by users.
- It is deemed insufficient in some way; at which point, it is sent back to an earlier phase.

While an individual usage decision does not in any way affect the acceptance decision that preceded it, a negative acceptance decision or readiness decision can cause the release candidate software to be "sent back" to an earlier point in the process.

Figure *3* - Paths through the Acceptance Process illustrates the possible paths through the acceptance process when the readiness or acceptance decisions cannot be made and require additional capabilities or information.

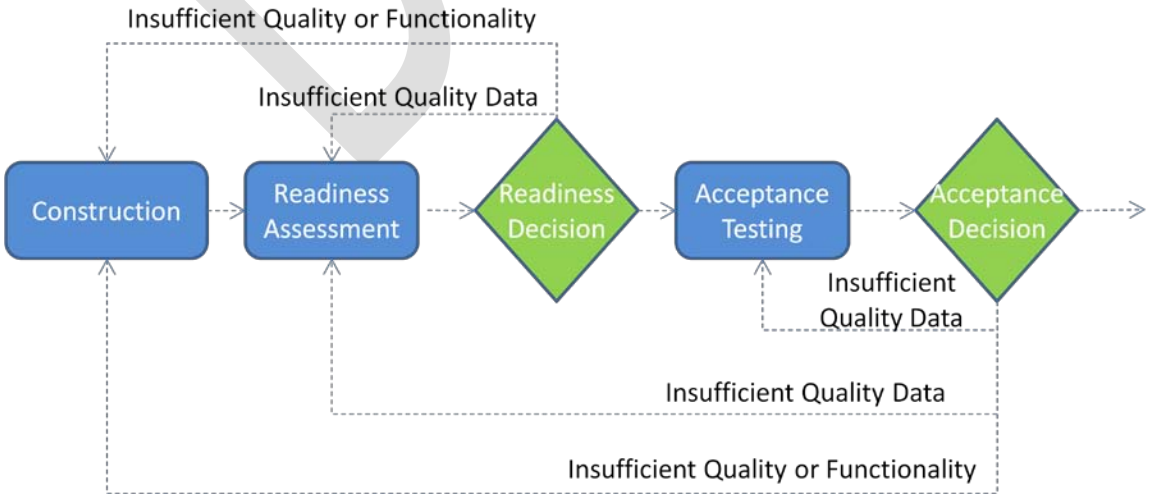


Figure 3

Paths through the Acceptance Process

From the readiness decision, a release candidate can be sent back to readiness assessment if there is insufficient quality data to make a well-informed readiness decision. If the data is sufficient to determine that critical functionality is missing or has sufficiently severe deficiencies, the release candidate can be sent back to development for remediation.

From the acceptance decision, a release candidate can be sent back to acceptance testing if there is insufficient acceptance testing quality data. If the acceptance decision also depends on data collected during readiness assessment and it is deemed to be insufficient, the release candidate can be sent back to readiness assessment. If critical functionality is missing or has sufficiently severe deficiencies, the release candidate can be sent back to development for remediation.

Normal practice is to log any bugs found in the bug management system so that the supplier can start the process of remediation but to continue testing until all the tests have been run or testing is blocked by a bug that prevents further testing from occurring. At this point testing stops until a new release candidate is received.

Bug triaging is the process of determining which bugs need to be fixed before building a new release candidate. After remediation of one or more bugs, a new release candidate is built and passed through the decision-making process. Each release candidate should be a distinctly named version of the software (and should be tagged as such in the source code management system.)

Note: Although the acceptance process outlined here appears to be very waterfall-ish with the software moving from one phase to another, this is not always the case. On agile projects, following the practice of *incremental acceptance*, each feature may traverse this process individually. Therefore, it is possible for one feature to be in the concept phase, with the development team working on another set of features in the development phase, and another set of features with the customer in the acceptance phase. Even on waterfall projects we may have more than one release candidate going through the process at the same time. For example, the Alpha release may be In Use, the Beta is in Acceptance Testing and developers may be working on additional functionality for the general release while also fixing bugs in the next release candidate for this release. That is four instances of the process running in parallel. A more complete discussion follows later in this chapter under the headings *The Acceptance Process for Alpha & Beta Releases*.

Why Separate Readiness Assessment from Acceptance Testing?

The process of verifying and validating the software-intensive system against it requirements and expected usage can be viewed as a single monolithic activity or as very fine grained steps. The primary reason for grouping these fine-grained steps into two major buckets is to separate the activities that should be carried out by the supplier before turning the system over to the customer from those that the customer does as part of deciding whether to accept the system. Readiness assessment is primarily about the professionalism of the supplier organization while acceptance testing is about validating that

the system as delivered will suit the purposes of the users and other stakeholders and confirming compliance to contractual agreements.

There may be other reasons to divide the activities into various categories. For completeness, we mention them only briefly because they are largely beyond the scope of this book:

- Deadline Pressure – Some managers might believe that having an earlier milestone at which time development should hand over software to readiness assessment might be viewed as an effective way to keep developers from slacking off. Others might consider these managers "pointy-haired bosses."
- Accounting – The initial construction of the software may be treated differently from an accounting perspective than fixing of bugs (and especially those that can be considered change requests).
- Early Validation – It is useful to have real users try using early versions of the product to validate that the product, once finished, will fill the niche for which it was targeted. This clearly isn't full-on acceptance of the product so it *could* be considered readiness assessment. Prior to doing this type of testing with users, the supplier would want to do their own due diligence to ensure the software was working well enough to allow the users to try it. Therefore, we would consider this a form of Alpha/Beta release or possibly incremental acceptance testing or even conditional acceptance.
- End User Training – Exposing the software to users before acceptance can be an effective way to start the training process. If the quality of the system is high enough, it can also be useful as a form of viral marketing. Both of these uses fall into the category of Alpha/Beta releases rather than readiness assessment.

Readiness Assessment or Acceptance Testing?

Given a particular testing activity, should it be considered part of readiness assessment or acceptance testing? There are several factors that play into this decision:

1. **Who's doing the assessing:** Acceptance is done by the customer or their proxy (someone acting directly on the customer's behalf) while readiness assessment can be done by anyone involved in the project. Where the supplier and customer are distinct entities, the roles should be fairly clear. Things get murkier when there is no real customer involved, common in product companies, or when there is a separate organization charged with testing. These scenarios will be covered in more detail in Chapter 2 - The Decision Making Model.
2. **Formality of the testing:** Acceptance is a more formal activity because the customer is involved. This implies more formal record keeping of any concerns that were identified. Readiness assessment can be much less formal; it need only be as formal as dictated by the project context. Suppliers that need to be able to pass formal audits will keep much more formal records. Agile projects tend to be very informal; during readiness assessment they just fix the bugs immediately rather than defer them for fixing later.

The lines can get somewhat blurred when there are more than two different stages of testing; these scenarios are described in Acceptance in Complex Organizations and Accepting Complex Products. In the end, it is less important to decide which label to apply to a particular testing activity than it is to ensure that the right testing activities get done and that each party knows for which activities they are responsible. It can be useful to list all the potential testing activities and for each one decide whether it is mandatory, optional, or not required, and to assign responsibility for it to a specific person or organization. A sample spreadsheet is provided <online at http://testingguidance.codeplex.com>.

1.2.2 The Acceptance Process for Multi-Release Products

Thus far, we've focused on the acceptance process as it applies to a single release candidate. How does the acceptance process get applied when a product will have multiple releases?

For the most part, long-lived multi-release systems can be thought of as simply a sequence of individual products, where each product is being individually assessed for readiness and acceptance. Each release goes through the entire decision making process. Figure 4 - The Acceptance Process with Multiple Releases illustrates an example of this process.

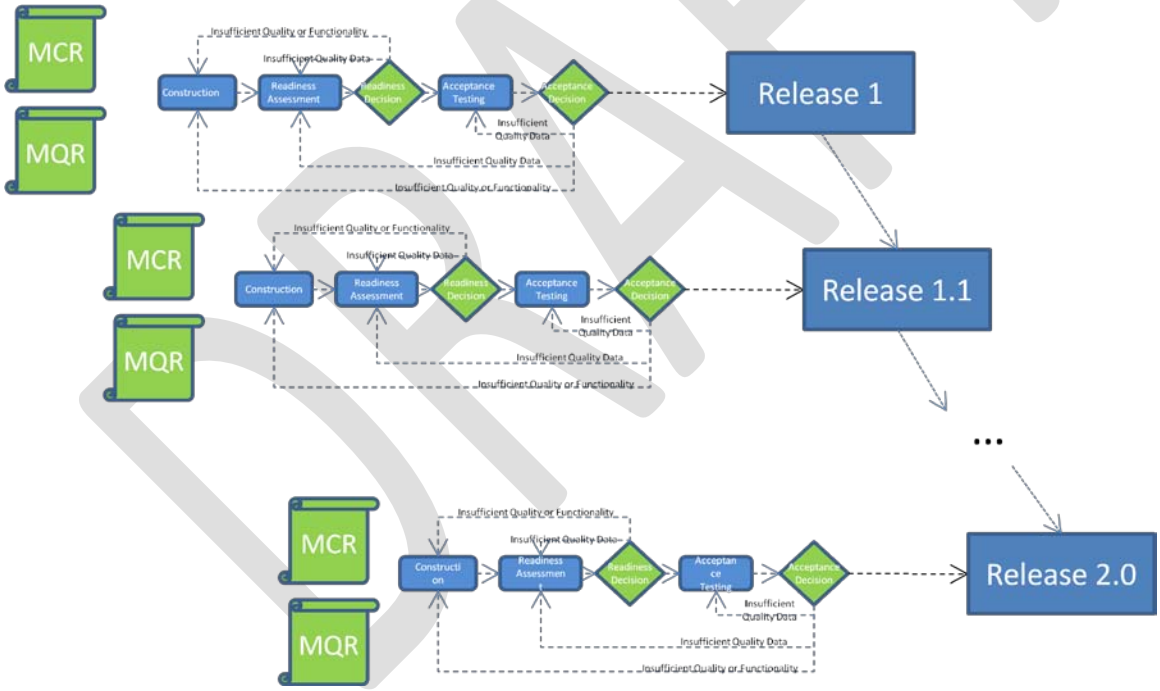


Figure 4

The Acceptance Process with Multiple Releases

The set of functionality required for each release, which we call the minimum credible release or MCR, is unique based on the goals for the release determined by the product owner. The set of quality criteria, which we can the minimum quality requirement or MQR is somewhat more consistent from release to release but it may evolve as the product matures and product context evolves. The specific criteria would be selected from the set of criteria in effect at the time of the project (which may vary from those

that were in effect for earlier releases.) An example of this is that the Sarbanes-Oxley Act (SOX) was enacted in 2002, so all subsequent releases required compliance with this act as a readiness and/or acceptance criteria. Subsequent releases may also have backward compatibility requirements that did not exist for earlier releases.

1.2.3 The Acceptance Process for Alpha & Beta Releases

Alpha and beta releases are ways to use end users as accept*ability* (not accept*ance*) testers to gather more data about the product as it might be used "in the real world." The end users may be internal to the organization, or external but friendlier (meaning more bug-tolerant) or having a stake in the outcome (wanting to see the product earlier for their own benefit.) In some organizations internal alpha testing is called “dogfooding” so named because when someone produces dog food, they should be prepared to feed it to their own dog.

The final outcome of alpha and beta testing is as likely to result in changes to the MCR and MQR of the product (in effect, new functionality or quality requirements) as it is to be a collection of bug reports that describe failure to meet the existing MCR or MQR.

Each alpha release and beta release can be considered a separate release with its own release decision and acceptance decision. That is, the development organization needs to deem it to be ready for an alpha (or beta) release and the product owner needs to accept it as being ready for alpha release as in: (“I accept this alpha release as having sufficient functionality and quality to warrant releasing to users to collect feedback …” This is illustrated in Figure 5 – The Acceptance Process with Alpha and Beta Releases.

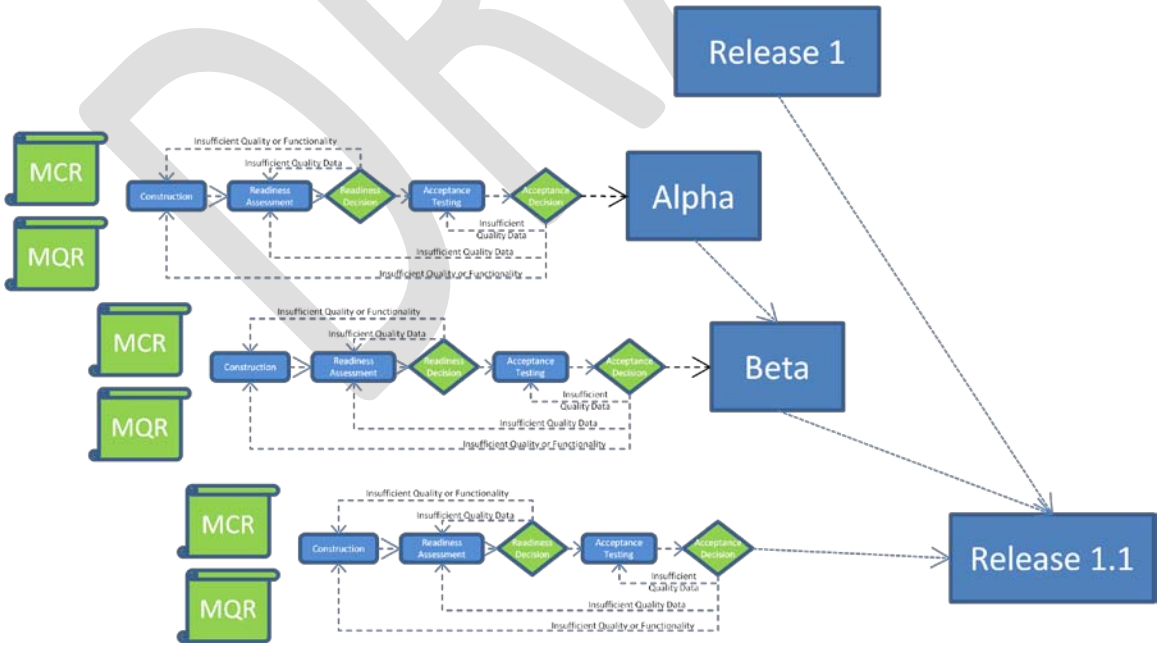


Figure 5

The Acceptance Process with Alpha and Beta Releases

Note that every release, whether alpha, beta or final has a readiness decision and an acceptance decision. The functionality (MCR) and quality (MQR) for an alpha release are typically lower than that needed for a beta release, which is lower than needed for a general release. For example, the MCR for the Alpha release may be a core subset of functionality; not all features need be present. The MQR may be "no severity 1 bugs" and "works for up to 10 users (versus the 1,000 required in production)." Typically, the MCR/MCR criteria for the Beta release will be based on the Alpha criteria with additional criteria based on improvements previously planned as well as feedback from the Alpha testing as shown in Figure 6 – Alpha or Beta Feedback.

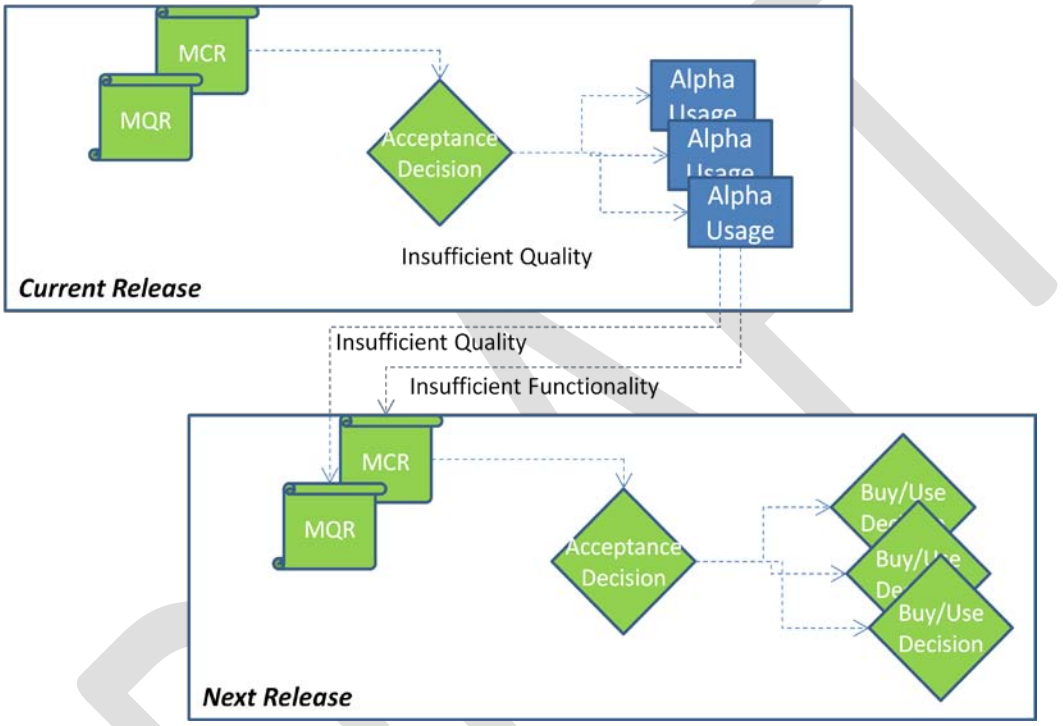


Figure 6
Alpha (or Beta) Feedback

Similarly, the MCR and MQR for the general release would be affected by feedback from users of the Beta release.

1.2.4 Soaking, Field Trials and Pilots

New products are often tested with pilot groups before being rolled out to larger groups of users. A “pilot” is typically the first production quality release with functionally picked to satisfy a particular target audience. Therefore, the MCR bar is lower but the MQR needs to be sufficient to satisfy the users. Unlike an Alpha or Beta release, this is real production software. A pilot user group will often receive several dot releases based on issues they find and report. The initial pilot may be followed by a larger pilot group, who may require additional features, or the product may go straight to general availability.

Each of the pilot releases would be subject to the acceptance process with both readiness and acceptance testing preceding deployment.

A similar strategy is known as field trial or “soaking” (think of the washing metaphor: just like clothes you let the software “soak” for a while, to make sure it comes out “clean”). The software is deployed to friendly customers/users for an extended period (more than just a test cycle) to see how it behaves in a real customer environment. As with a pilot, readiness assessment and acceptance testing would be done to ensure that the MCR and MQR are satisfied. The outcome in all cases is to gather user feedback that may cause the product owner to adjust the MCR and MQR expectations of subsequent releases, not to revisit the acceptance decision of the pilot or field trial release.

1.2.5 Software Maintenance

Any time software needs to be maintained (such as when small changes are made to the software and those changes are deployed), you are, in effect, creating a minor dot release of the software that needs to go through the entire decision-making cycle yet again. It is common to look for ways to reduce the cost of gathering the data to support the acceptance decision. Some ways of doing this increase the risk of possibly missing newly created bugs (also known as "regression bugs") by reducing the amount of testing (for example, risk-based test planning) while others simply reduce the effort to get similar test coverage (for example, automated regression testing).

Another unique aspect of software maintenance relates to the warranty period on a software release. Any changes that need to be made to the software should be made in the source code management (SCM) system. When building multiple releases, there may be ongoing development for the next release that should not, under any circumstances, be inserted into the production system along with the warranty bug fixes. This requires managing separate code streams or branches during the warranty period and ensuring that all warranty fixes are also applied to the new development code stream. For practical strategies for using source code management systems, see [SCM].

1.2.6 Conditional Acceptance/Readiness

Frequently, the acceptance decision maker accepts a product with conditions. Accepting a product with conditions is a short-hand way of saying,

"The product is not acceptable yet, but it is close to meeting our criteria for functionality (MCR) and quality (MQR). If you address the following concerns (and we find nothing new in the subsequent round of acceptance testing), we intend to accept the product in the next pass through the decision making process."

Conditional acceptance brings the process back to the construction/development phase of the acceptance process, but this time with a much better idea of exactly what must be done to make it through both the readiness decision and the acceptance decision on the next round.

1.2.7 The Acceptance Process for Highly Incremental Development

The acceptance process as described thus far looks very ”waterfall” in nature but it can also be used in a highly incremental fashion on agile projects. Each chunk of functionality (often called a “feature” or a “user story”) can be passed through the acceptance process independently as shown in Figure 7 - The Acceptance Process Applied to Incremental Development.

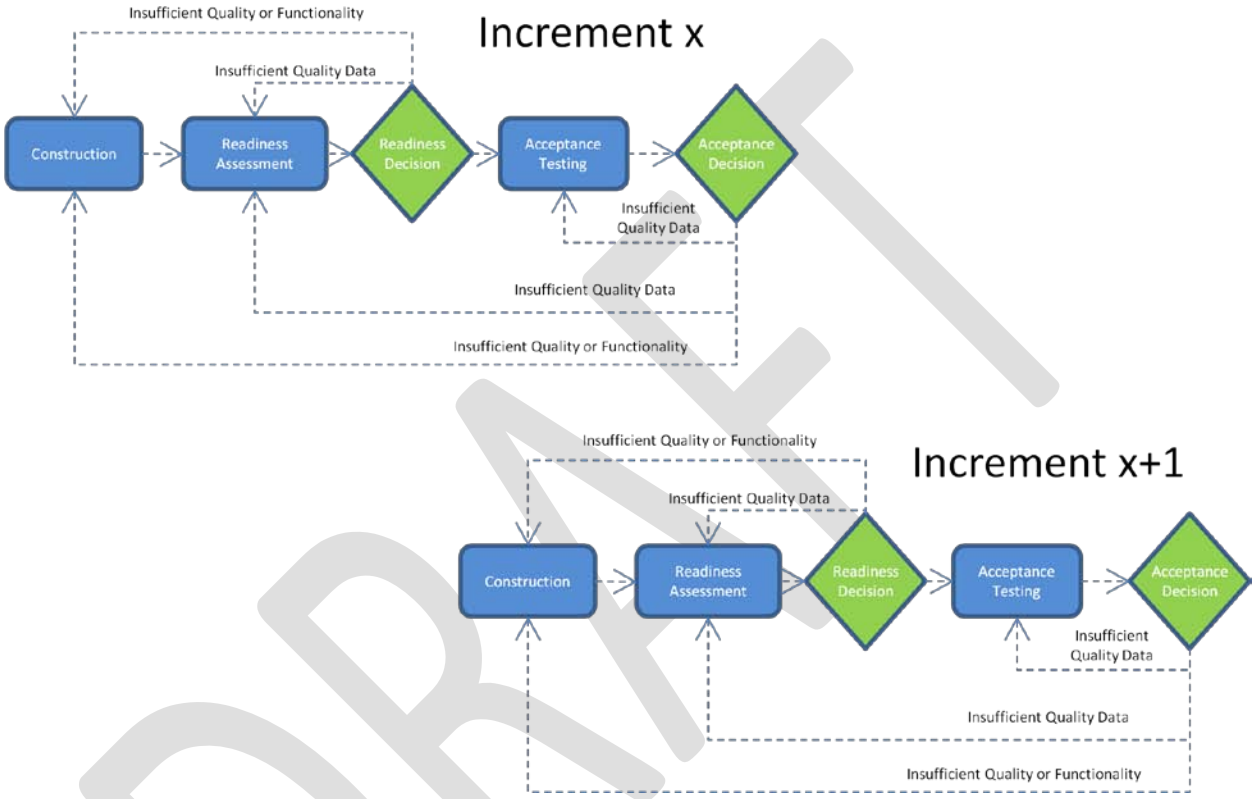


Figure 7

The Acceptance Process Applied to Incremental Development

The individual MCR for each feature or user story may be:

- *independent* of that of other features of user stories; or
- *incremental* when it may build on the MCR of prior features.

At some point, a product-level readiness and acceptance decision is made to ensure that everything works properly together. Refer to Accepting the Output of Feature Teams later in this chapter for a more complete discussion.

This allows the developers to pre-execute the acceptance tests as part of the development cycle. When all the tests pass for that feature or user story, they turn over the functionality to the Product Owner (or other customer proxy) for immediate "incremental acceptance testing." Therefore, acceptance testing

at the feature level starts immediately after the developer decides that all or most of the functionality is built and working correctly. There may also be a round of acceptance testing performed at the end of the iteration, as illustrated by the medium-sized testing bars in Figure 4 in the Introduction. The activities conducted for each feature within the iteration are illustrated in Figure 8

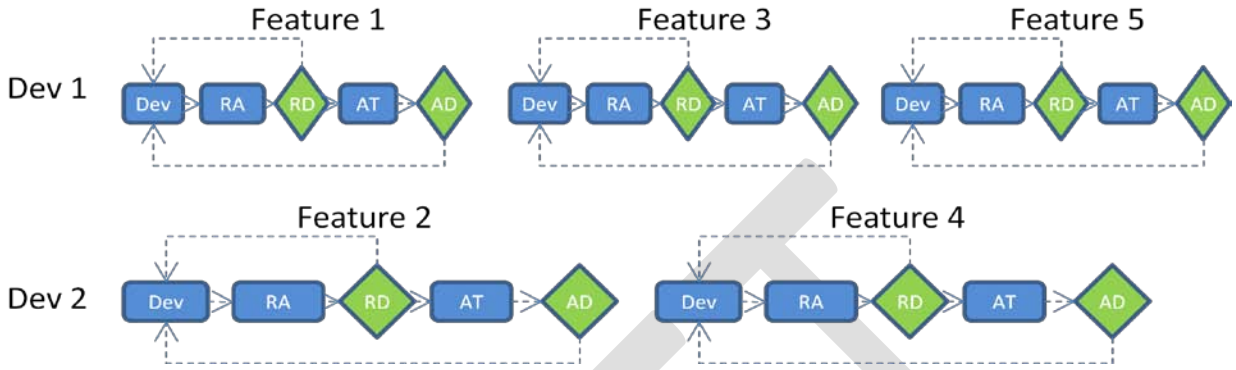


Figure 8

Agile Development with Incremental Acceptance Testing

Development is done one feature at a time by either a single developer or a small team of developers, testers and user experience people, depending on the size of the feature. In Figure 8 – Agile Development with Incremental Acceptance Testing, Dev 1 and Dev 2 could each be a single developer or a small team. When the development work is complete, the developer, possibly assisted by a tester or business analyst conducts readiness assessment on the feature by running all the known unit, component and business acceptance tests against the software. If they find any problems, they fix them before proceeding.

When they are satisfied that the software is working properly, they ask the customer to run their acceptance tests for that one feature. If the customer finds any problems, they show the problems to the team who then fixes the problems immediately and repeats the readiness assessment on the revised code before turning it over to the customer for further acceptance testing. When the customer is satisfied, they accept the feature and the developer (or team) starts working on their next feature. This practice, called *Incremental Acceptance Testing,* has two key benefits. First, any concern found by the customer during acceptance testing can be discussed with the developers while they still remember the details of how they implemented the functionality. Second, the defects or deficiencies can be addressed immediately before the developer moves on to the next feature instead of being stockpiled for a "bug-fixing phase." This is one of the key reasons co-located agile project teams frequently do not use a formal bug-tracking database; one sticky note per bug on a bugs board promotes high visibility with very low management overhead.

This process works best when the customer supplies the acceptance tests to the team before development is finished at the latest and ideally before development even starts. This practice is known as *Acceptance Test-Driven Development (ATDD)* or *Storytest-Driven Development (STDD).* The team will typically automate much of the testing to keep the cost of readiness assessment low. The sidebar “What it takes to do Incremental Acceptance” describes other success factors for doing highly incremental acceptance testing.

1.3 Elaborating on the Acceptance Process

The acceptance process described thus far has been kept deliberately simple. This section introduces some ways to vary the acceptance process to deal with more complex situations including complex organizations and products.

1.3.1 The Role of the Usage Decision

Thus far we have focused almost entirely on the readiness and acceptance decisions. What about the usage decision? It turns out that the usage decision doesn't directly impact the acceptance decision because it happens after the product is already accepted.

Separating the Acceptance Decision from the Usage Decision

When a software product is being selected or built specifically for a customer by another party, the customer may decide to accept the software as suitable before offering it to the users who will then make the usage decision individually as illustrated in Figure 9.

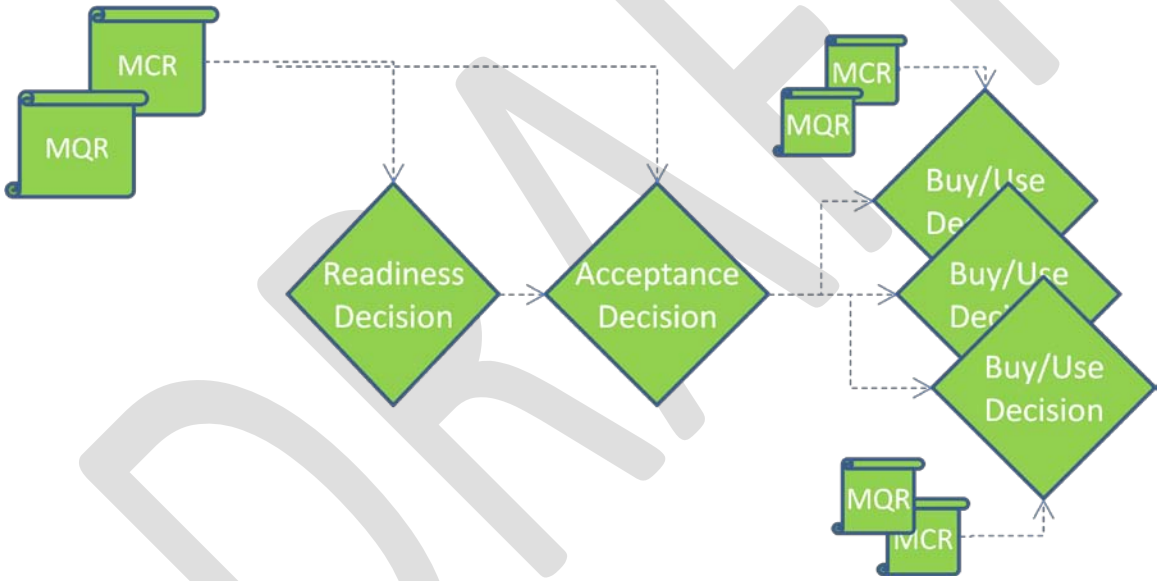


Figure 9
Acceptance and Usage Decisions and Criteria

Almost all products have many users and each is likely to have their own usage criteria which while it may have influenced the acceptance criteria used during the readiness and acceptance decisions, may in fact be quite different from it. In some cases, the use of the product is not optional, such when software is used to carry out a specific job such as a call center agent or airline reservations system. In these cases the usage decision is made on behalf of the users by their management; this decision need not follow immediately after the acceptance decision as there may be other prerequisites for usage such as user training.

While a user could, in theory, choose not to use the software, the consequence of losing their job would make this decision highly unlikely. The user's feedback, however, might influence subsequent releases of the software as we'll see in Feedback from the Usage Decisions.

Determining the Acceptance Criteria

The acceptance decision is made based on some criteria including both functionality and quality related elements. The minimum level of functionality is sometimes called the Minimum Credible Release (MCR) or Minimal Marketable Functionality (MMF). The minimum level of quality is sometimes called the Minimum Quality Requirement (MQR). These criteria should be determined well in advance of the acceptance decision to give the supplier of the software a reasonable chance at being able to satisfy them. Therefore, the decisions about the MCR and MQR come before the Acceptance Decision as shown in Figure 9 – Acceptance and Usage Decisions and Criteria.

Feedback from the Usage Decisions

Note that the usage decisions made by individual users after the acceptance decision do not directly affect the acceptance decision but it may, through sales data, post-sales support data or focus groups affect the definition of MCR or MQR for subsequent releases of the product. This is illustrated in Figure 10 - Usage Decision Feedback.

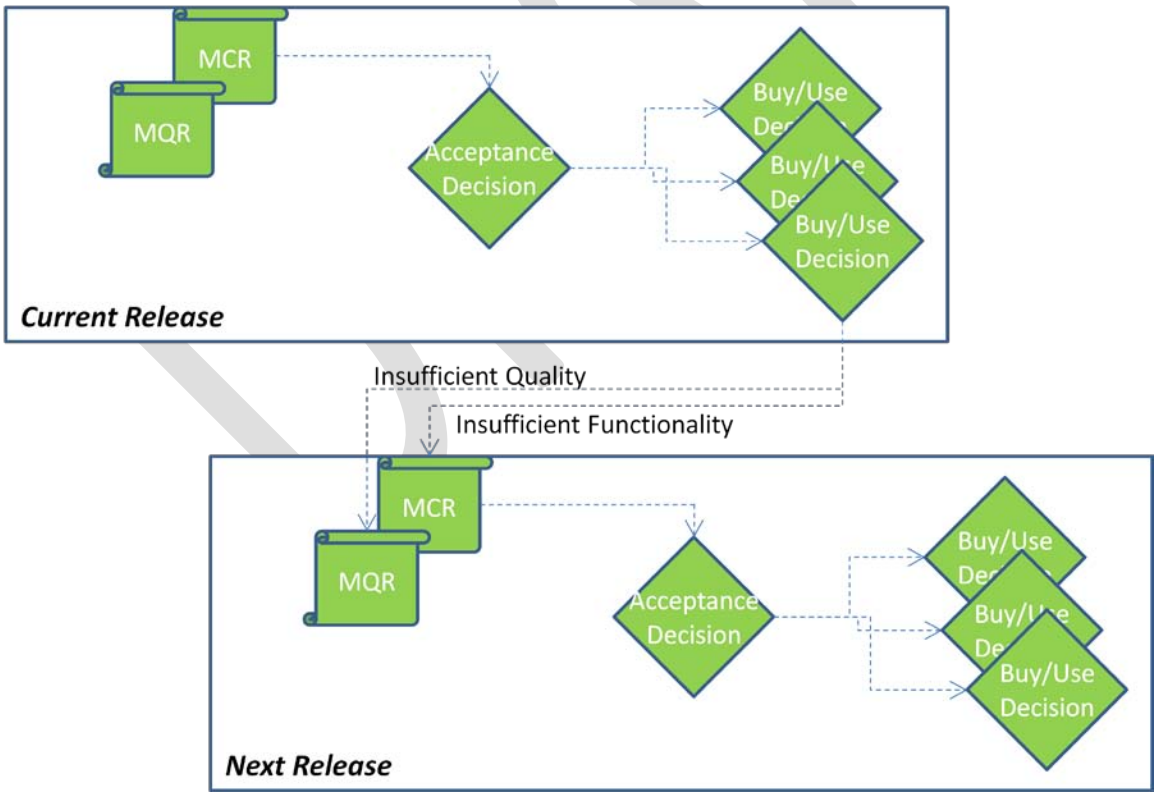


Figure 10

Usage Decision Feedback

While the MQR tends to be fairly constant from one release to the next, the customer or product manager may revise the quality criteria for a subsequent release based on the feedback received from users. For example, user testing of an alpha release might reveal that the response times specified in the MQR are not fast enough in practice and might cause the product owner to make them more stringent for the beta or final release.

The MCR for each release tends to be unique because it is the list of functionality that needs to be included to warrant having the release. Feedback from users can influence the MCR based on requests for missing functionality, suggestions for how existing functionality should be changed, and through bug reports.

Feedback from users of Alpha and Beta releases often results in changes to the MCR or MQR of the final release.

1.3.2 Acceptance in Complex Organizations

The simple acceptance process described previously will suffice in many organizations. But there are some organizations whose structure is very complex as are the products they develop. These organizations require a decision-making process that mirrors the complexity of the organization or product. The decisions may be made in parallel by several organizations each with veto power over the handoff or they may be made in series as a more complex product is assembled from components.

Parallel Acceptance Decisions

Many organizations have other departments who are responsible for operating and supporting the system once it is deployed. These departments need to have a say in the acceptance decision. This is reflected in Figure 11– Parallel Acceptance Decisions shows several acceptance decisions being made in parallel, each based on a different set of acceptance criteria.

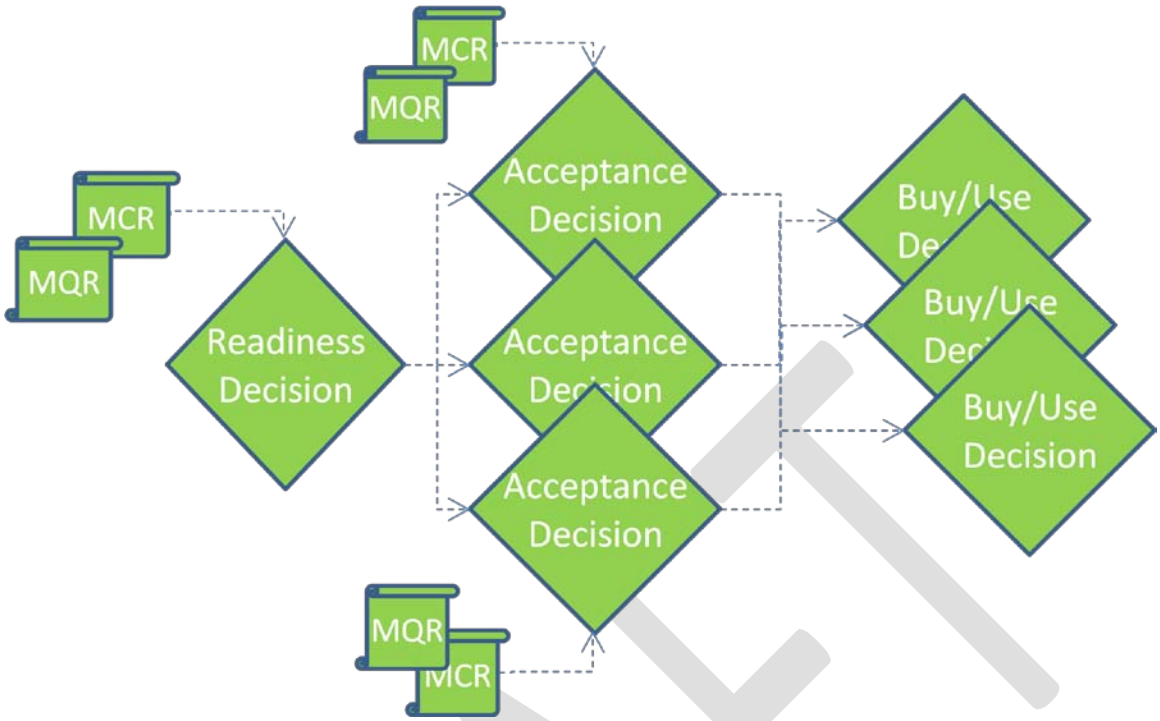


Figure 11
Parallel Acceptance Decisions

When there are several organizations involved in the acceptance decision each with their own area of specialty, each may insist in conducting its own activities independently of each other. Each makes their own decision regarding the release candidate and any one of them can effectively veto the others.

Examples of such stakeholders, and their area of interest , could include:

- Business Unit – Functionality is acceptable.
- Operations Department – The product can be managed and supported in production.
- Corporate Branding – Product complies with branding requirements.
- Corporate Security – Product has complied with security requirements.
- Usability Group – Product complies with usability requirements.

Figure 12 – Parallel Acceptance by Customer and Operations illustrates the scenario where the customer is the party who has conceived the software as a solution to a business problem and the operations department is tasked with supporting the system once it is put into production.

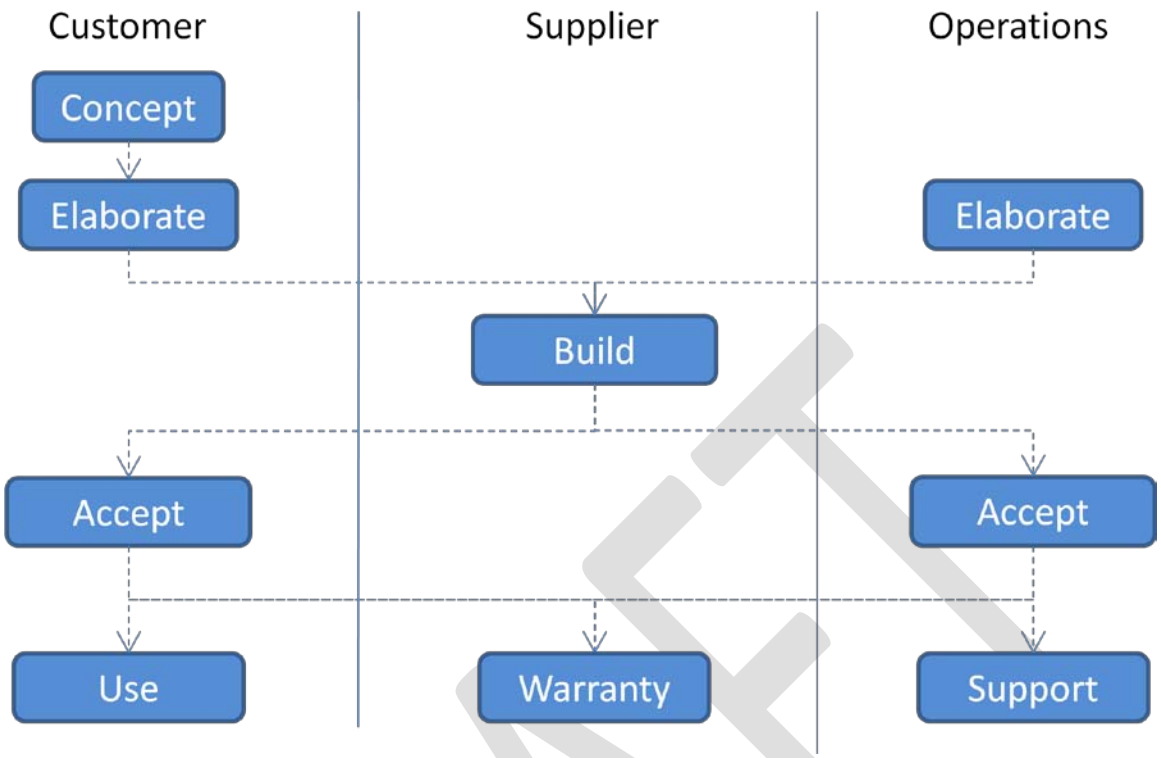


Figure 12

Parallel Acceptance by Customer and Operations

Parallel Readiness Decisions

In some cases several organizations do their own readiness assessment of the release candidate and make independent readiness decisions. The product will not be handed over to the product owner for the acceptance decision if either organization deems the product “not ready” until any additional work they identify has been completed and they deemed the software “ready”. This is illustrated in Figure 13 – Parallel Readiness Decisions.

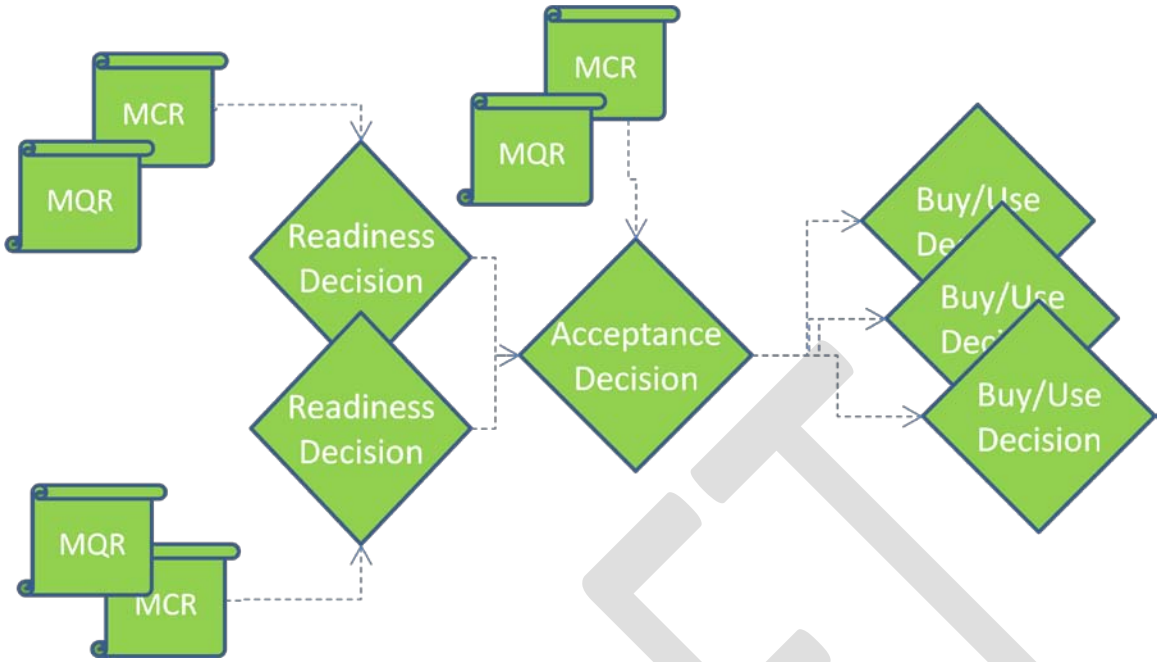


Figure 13

Parallel Readiness Decisions

Examples of potential stakeholders with parallel readiness decisions include:

- Development Team - Code is passing all unit tests and functional tests
- Security Team – Design has passed security review
- Data Architecture Team – Data architecture has been approved
- Legal Department – All embedded technologies have been suitably licensed.

Whether a decision is deemed to be part of readiness or acceptance is somewhat arbitrary. The choice is usually based on when the decision maker wants to get involved and which side of a larger organizational boundary the stakeholder sits. For example, in some organizations the security department may be considered a business function who must accept the release candidate while in others it may have a presence in or near the IT department or product development group and thereby choose to participate when the readiness decision is made.

1.3.3 Accepting Complex Products

Organizations that build large, complex products typically use a more complex version of the acceptance process that involves multi-stage acceptance. Large complex products typically require large numbers of people, too many to work in a single team. While the large complex products are typically composed of components, the teams that build them may be organized in several ways. Each component may be built and owned by a component team or teams may be organized around the functionality provided to the users. The latter approach is called “feature teams” and is widely used by the Developer Division of Microsoft. The manner in which these people are organized impacts the acceptance process.

Accepting the Output of Component Teams

When the product consists of components that are built and verified in parallel by separate component teams, each component should have its own readiness assessment prior to conducting readiness assessment on the integrated product. This is illustrated in Figure 14 - Component Teams with Serial Decisions.

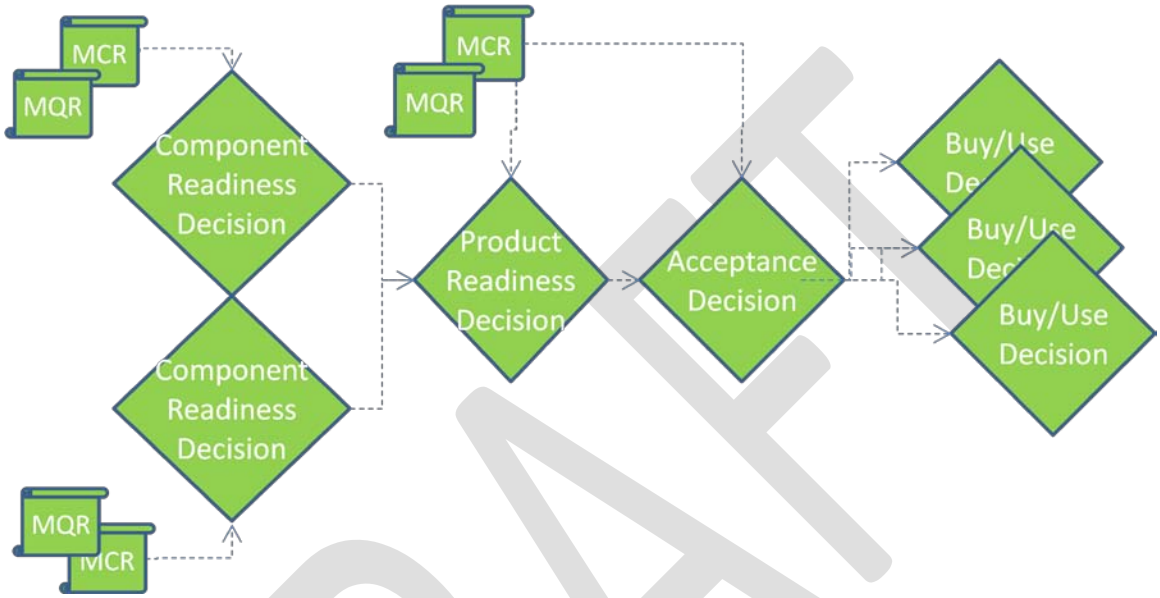


Figure 14

Component Teams with Serial Readiness Decisions

Each component must go through a component readiness decision before being integrated into the larger product for subsequent product-level readiness decision. The integrated product is then turned over to the customer for the acceptance decisions. The readiness assessment activities may go by various names including “component testing” or “system testing” for the components and “system testing” or “integration testing” for the entire product. Each component has its own MCR and MQR; hopefully, these are related to the MCR/MQR for the readiness and acceptance decisions made about the entire product.

This reveals several of the weakness of this approach to complex products. First, because the components cannot be directly understood and therefore accepted by the customer, some intermediary, usually an architect has to be responsible for translating customer requirements into component requirements. It is also possible for the component requirements to drift from what is actually required. Second, the product readiness cannot easily be assessed until all of the components have passed their respective readiness decisions and are integrated to form the product. This delays the product readiness assessment until after the “big bang” integration of the components, which is rather late in a project to discover that some of the component requirements were misunderstood. This

problem can be somewhat overcome through the use of continuous code integration (CI) practices combined with incremental integration [Crocker] and incremental acceptance testing.

Accepting the Output of Feature Teams

Another way of organizing the development of large complex products is to use feature teams. Each feature team is responsible for building and verifying a feature based on the customer's acceptance criteria. Building each feature may involve modifying several components but the team includes expertise on each component and ensures that the components are properly integrated as part of their feature readiness assessment. Therefore, the product readiness assessment and the product acceptance testing are both focused on feature integration rather than on component integration. This scenario is illustrated in Figure 15 – Feature Teams with Serial Acceptance Decisions.

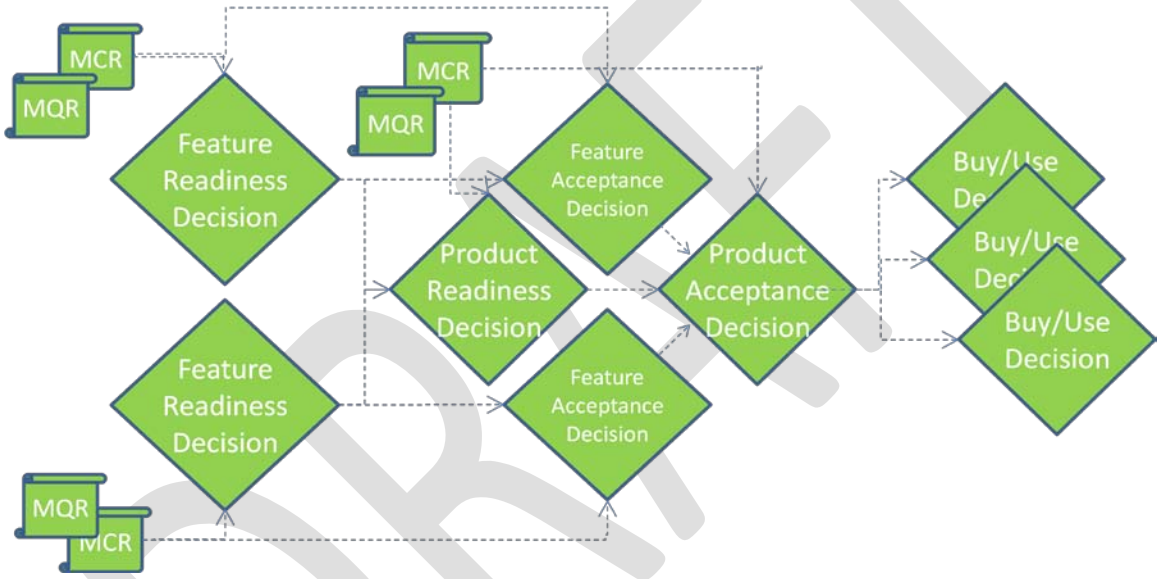


Figure 15

Feature Teams with Serial Acceptance Decisions

Each feature's readiness decision is made based on the MCR and MQR specific to that feature as supplied by the customer; no intermediary is required to translate the customer's requirements into component requirements. The feature readiness decision is a prerequisite for the corresponding feature's acceptance decision which should use roughly the same acceptance criteria. Meanwhile, the product is integrated and a product-level readiness decision is made based on the product's MCR and MQR. The product-level MCR should simply be an aggregate of the feature-level MCRs[1]. When the product-level readiness and all feature-level acceptance decisions are positive, the product-level acceptance decision can be made.

[1] This in turn forces the product owner to specify interactions between features as part of the MCR for each affected feature.

This style of organization and acceptance is preferable to component teams because it ensures each team is working with a real end customer and is responsible for delivering a whole product, not just an internal component. See the sidebar Feature Teams on Microsoft Office for an example of how this works in practice.

Accepting Customizable Products

Many products are built by one organization for sale to another customer organization. When the product needs to be customized for each customer there are two distinct acceptance phases for the two different products in question. The first product is the generic product built by the selling organization to the requirements specified by the product manager. The second product is the one built to the specification of the purchasing customer. The construction of the latter product may include either configuration or custom coding in addition to the generic customizable product.

The first acceptance decision is when the product manager accepts the customizable product as saleable to the customers. This acceptance process could be any of the variants described previously including Accepting Complex Products.

The second acceptance decision is when the purchaser of the customizable product accepts the product as customized to their own situation. There is a single acceptance decision for each version of the customizable product and one acceptance decision per customer as shown in Figure 16 – Accepting customizable products.

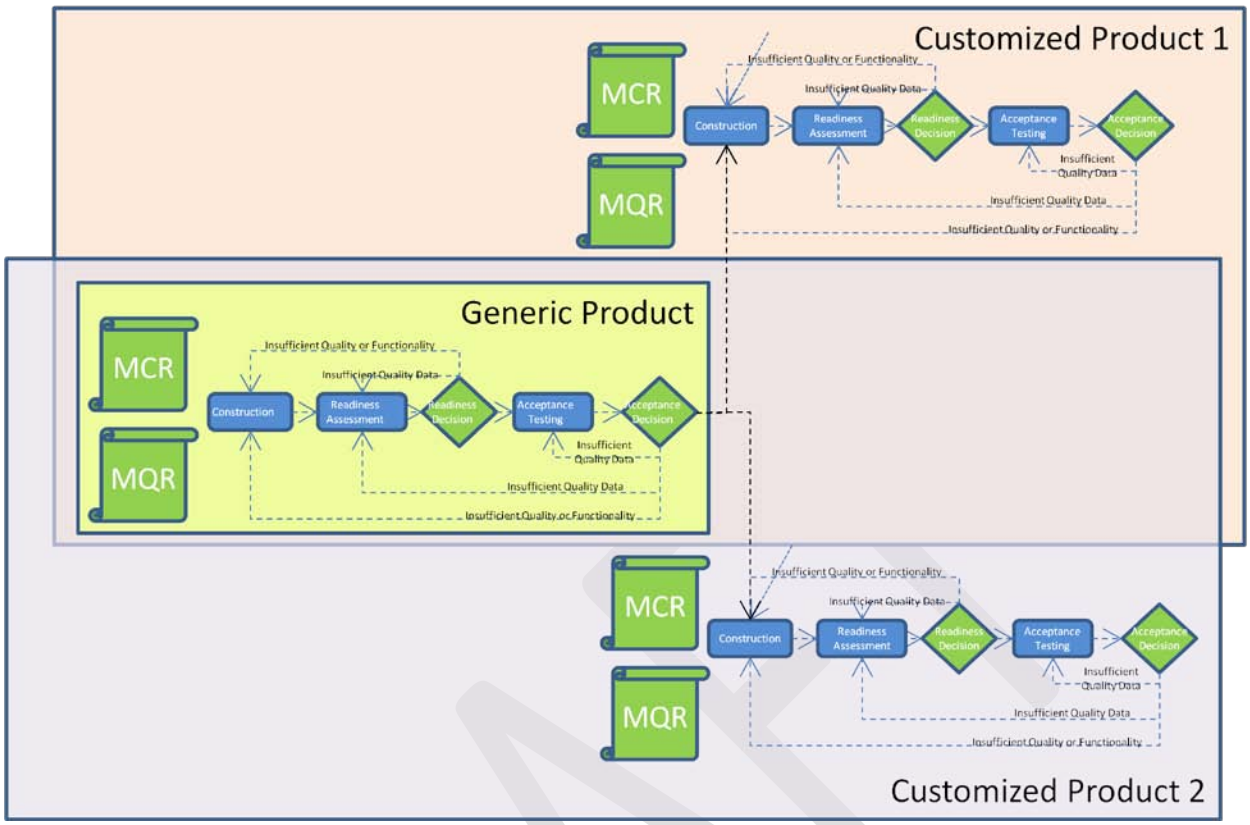


Figure 16
Accepting customizable products

Sidebar: Accepting Configurable ERP Products. Many software products are sold as generic implementations of a business process that can be configured to suit a business' specific needs. Two good examples of this are ERP systems such as SAP's ERP product and Microsoft's Dynamics. Both provide generic implementations of core business processes such as time-reporting and payroll, supply ordering and accounts payable, product order processing and accounts receivable. The out-of-the-box implementations of these processes can be modified though configuration data or extended by plugging in custom-written procedures. For a particular customer's installation of the ERP system, it goes through two completely distinct acceptance processes. First, the product manager at SAP or Microsoft defines the goals of the next release by synthesizing new feature ideas from the product definition team, bug reports from previous releases and new feature requests from new and existing customers. This gets turned into the functional (MCR) and parafunctional (MQR) requirements for the release. The ERP product is built, goes through readiness assessment by the development team, acceptance testing by the product management team and professional testers before being declared "ready to ship" to customers

Meanwhile, the customers are defining their own goals and objectives for the installation based on the expected feature set of the new release and their specific business context. They may do business process modelling and gap analysis between their business process and the "out-of-the-box" product to define their own MCR and MQR. When the product ships, they install it into their own development system and do their customization and/or configuration. If this is their first installation of the product then they are the various configuration parameters and new procedures. If this is an upgrade from a previous version of the ERP system then much of the work consists of reapplying the configuration from their previous system and doing regression testing of the functionality. In both cases, the acceptance decision is based on the MCR and MQR defined by the customer for their specific installation of the ERP system, not the MCR and MQR defined by the vendor for the generic product.

1.3.4 Exit vs. Entry Criteria

Many of the decisions in the acceptance process involve handoffs between parties. Each handoff involves two sets of criteria. The party making the handoff may have some exit criteria that must be satisfied before they are prepared to make the handoff. The party receiving the handoff may have some criteria to be satisfied before they are prepared to receive the handoff. These criteria can be assessed at separate exit and entry decisions as illustrated in Figure 17 – Separate Exit vs. Entry Decisions & Criteria.

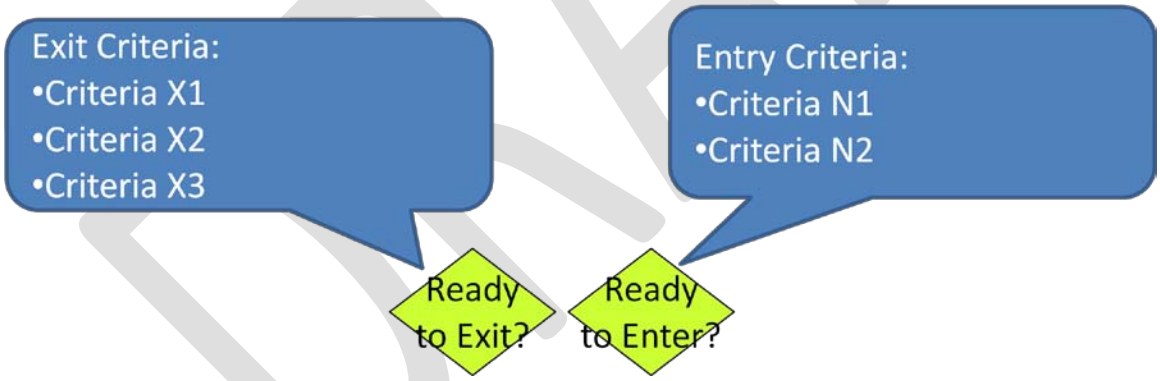


Figure 17
Separate Exit and Entry Decisions and Criteria

This effectively results in a situation where there is no "no man's land" between the Build and Accept phases as shown in Figure 18:

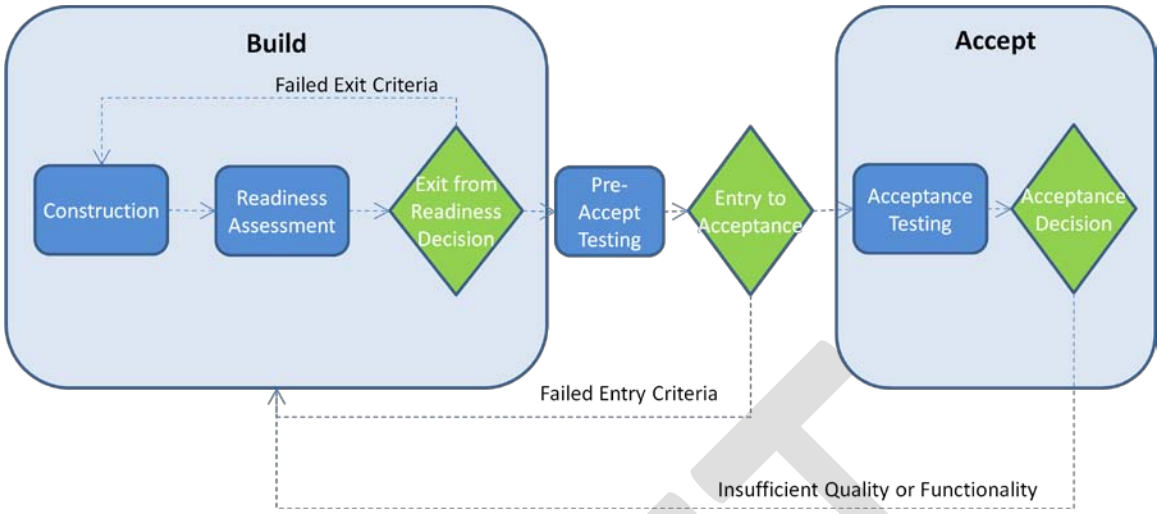


Figure 18

Acceptance Process with Separate Exit/Entry Criteria for Acceptance

When the supplier and product owner parties have a good working relationship they should be able to agree upon a single set of criteria that satisfies both their needs and have a single decision point based on the agreed upon set of criteria as illustrated in Figure 19 – Single Decision Point with Combined Exit and Entry Criteria

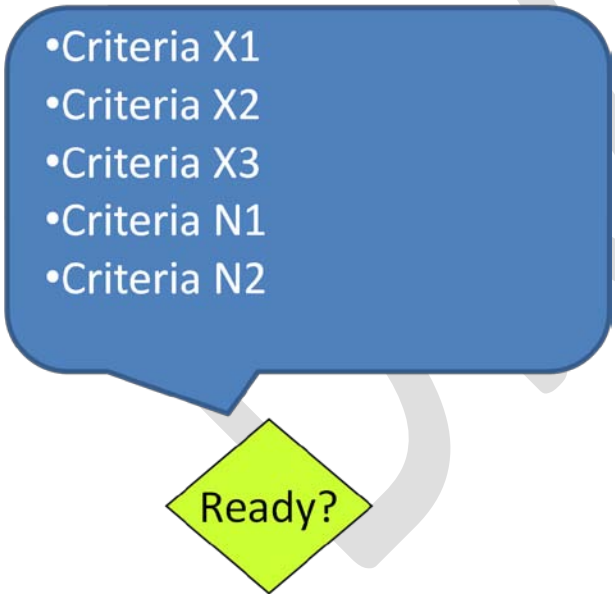


Figure 19

Single Decision with Combined Exit and Entry Criteria

The two parties would agree on which party does the assessment for each criterion and who makes the decision based on the assessment. Typically it is the party doing the handoff that will do the assessment and the receiving party may be interested in seeing the results.

The main entry criterion for the acceptance phase is that the supplier has deemed the software ready for acceptance testing. Secondary criteria may include whether the customer is sufficiently prepared to conduct the acceptance testing. The decision to enter the acceptance phase should take all the criteria into account.

For example, a supplier's Readiness Decision should be made based on a combination of:

- Entry criteria for acceptance testing provided by the customer including:
    - All features in MCR functional with no known severity 1 & 2 bugs open.
    - Stress tested at 110% of rated capacity for 48 hours with less than 1 failed transaction per 1000.
    - Regression test of previous release features executed with 100% pass rate.
- Exit criteria for construction provided by the supplier such as:
    - 90% code coverage by unit tests.
    - Data architecture reviewed and approved.
    - Static analysis has identified no deficiencies.
    - Security review completed with no high priority issues open.

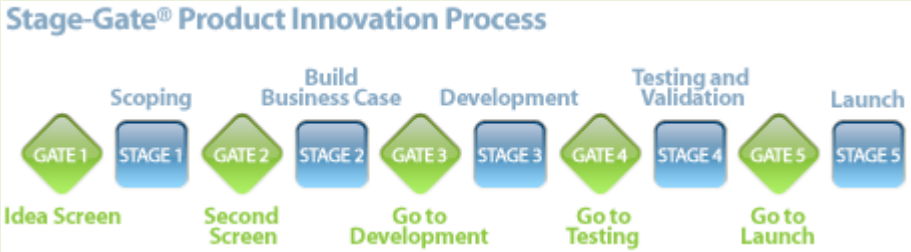
In practice it is highly preferable for the readiness criteria to include all the customer's acceptance criteria. This avoids finding bugs during the acceptance testing phase.

Sidebar: The Acceptance Process in Stage-Gate Processes

Many companies employ a gating process to manage their product development pipeline. Common examples include Stage-Gate™ and home-grown process. These all employ a series of project phases and decision points between them. The main goal is to manage the dispensing of resources (funding being the most common) to projects. The stages typically involve increasing amounts of funding with go/no-go decision points between them.

A typical process will have an initial idea phase with a gate to decide whether the idea is worth pursuing. At this point a fixed amount of money/time/resources are allocated to the product to develop the idea further and build the business case. The next gate determines whether the idea warrants advancing to full-scale development. If so, resources are assigned for developing the product; if not, the project is killed. The next gate typically involves deciding whether development of the product is complete. If so, the product enters the testing or validation stage; if not, it stays in development unless it has run out of time or money in which case it may need to go back to the previous gate to get more money & time. This would typically involve updating the business case. The next gate determines whether the product is ready to be sold or used. The exact number of gates varies from implementation to implementation but it typically ranges from 3 to 5.

This example from http://www.stage-gate.com is fairly typical:



This book breaks down Stage 4: Testing and Validation into the acceptance process. While it could be argued that Gate 4: Go To Testing assumes the supplier organization has in fact tested the software extensively, in most organizations the majority of testing occurs in Stage 4. This book proposes moving much of the testing activities back into Stage 3: Development as the readiness assessment activity and making the readiness decision part of Gate 4: Go to Testing. This leaves Stage 4 focussed on Customer or User acceptance testing and Gate 5: Go to Launch as the Acceptance decision. The variations of who makes which decision and when are described in Chapter 2 – The Decision Making Model.

The Unified Process describes 4 phases: Inception, Elaboration, Construction, Transition, but it does not preclude doing any particular kind of work in any of the phases although the emphasis does change. The milestones at the end of each phase could be treated as gates.

The agile literature provides a different spin on how innovative products should be developed. Rather than a waterfall process with different decisions made at each gate, a more iterative approach is used where each cycle through the gating process approves additional funds for the next set of functionality based on what was learned in previous rounds. [Highsmith]

Given the waterfall nature of a Stage-Gate process at the project or product level, how does the Acceptance Process we execute for each release candidate relate to it? This depends on the exact definitions of the Stage and the criteria for the gates between them, but, in general, once all the stakeholders agree that MCR and MQR criteria for a particular release have been met, the gate opens, the project passes through to the next stage, and the gate swings shut behind it. Figure S-1 – The Acceptance Process in a Stage-Gate Process illustrates this through an example.

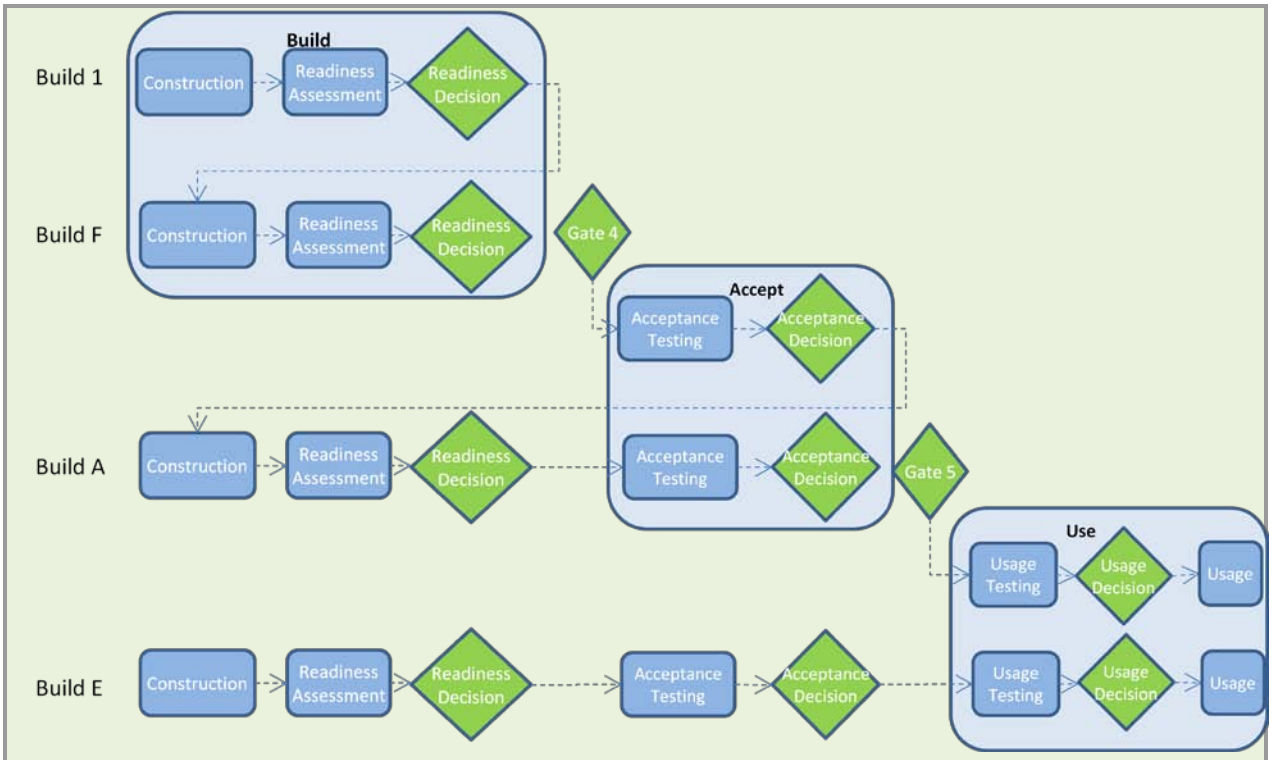


Figure S-1

The Acceptance Process in a Stage-Gate Process

Build 1 is the first release candidate that development asks their readiness assessors to evaluate. It has several severity 1 bugs that must be fixed and is missing some key features. Development fixes these bugs and adds the missing functionality. After several more tries, Build F finally makes it all the way through Readiness Assessment and is released to the customer for acceptance testing.

This satisfies the criteria for Gate 4 therefore the project is now in the Accept stage. The customer conducts acceptance testing only to discover some key issues. These are provided to Development who addresses them in Build A after several intermediate builds (not shown) failed readiness assessment. Note that the project is still in the Accept stage despite the fact that the ball was in Development's court to fix the bugs.) The customer finds no significant issues with this build and accepts the software as finished.

The project is now in the Warranty stage. After a few weeks of usage, a user experiences a severe crash that is traced back to a missed scenario. The customer asks development for a fix. Development comes up with a fix in Build E which passes readiness assessment and acceptance testing (mostly regression plus a new test case inspired by this bug) and Build E is put into production. The project stays in the Warranty Stage until the criteria to exit the warranty stage are met; this is usually the passage of an agreed-upon period of time.

Sidebar: Quality Gates and the Acceptance Process. (Microsoft Quality Gates examples as MCR/MQR criteria for readiness and acceptance decisions.)

1.4 Summary

This chapter introduces the concept of the acceptance process as a way to think about the activities related to making the decision to accept software. The acceptance decision is the culmination of a process that involves several organizations each with specific responsibilities. Each release candidate is passed through the decision points of the acceptance process on its way to the making the final acceptance decision. There are three major decision points between software construction by the supplier working with their customer and software usage by the end user. First, the supplier must decide whether the software-intensive product in its current form, known as the release candidate, is ready for acceptance testing. If it is, the customer must decide whether the release candidate meets their acceptance criteria before the product can be made available to the end users. Ultimately, each end user must decide for themselves whether they want to use the software. This usage decision has very little influence on the current acceptance decision although it may influence the acceptance criteria for future releases of the product.

While each user potentially makes their own usage decision, the readiness and acceptance decisions are normally each a single decision. Each decision point or "gate" should have well-defined criteria to guide the decision making. These criteria should be known in advance by both the supplier and the product owner. Complex organizations may have several readiness or acceptance decisions being made in parallel, each based on their own criteria (for example, operational acceptance); the release candidate's progression through the process may be vetoed by failing to satisfy the criteria of any of the parallel decisions. Complex products may require multiple supplier teams each with their own readiness, and potentially acceptance decisions prior to an overall readiness and acceptance decision for the integrated product; the release candidate can be sent back for further work from any of the decision points.

The acceptance described in this chapter applies to both waterfall and agile projects but in slightly different ways. The phased nature of waterfall projects means that all testing is done within a separate testing phase and the acceptance process describes what goes on within that phase. Agile projects traverse the entire development lifecycle for each feature or user story. Therefore, the acceptance process is executed at several levels of granularity with the finest grain execution being at the individual feature level and the largest being at the whole product (or feature integration) level.

1.5 What's Next?

In Chapter 2 – Decision Making Model we examine the roles and responsibilities of the various parties involved in making the decisions which make up the acceptance process. We also describe the roles played by the parties who provide the data on which the decisions are based.

1.6 References

[Crocker] Crocker, Ron “Large-Scale Agile Software Development” Addison Wesley Longman, Redwood City, CA 2004 ISBN:0321166450

[Highsmith] Highsmith, Jim “Agile Project Management: Creating Innovative Products” AWP 2004 ISBN-13: 978-0321219770

[SCM] Berczuk,Steve, Brad Appleton “Software Configuration Management Patterns: Effective Teamwork, Practical Integration “Addison Wesley (2003) ISBN: 0-201-74117-1

[Swim] Ambler, Scott “UML 2 Activity Diagramming Guidelines” http://www.agilemodeling.com/style/activityDiagram.htm

[LSD] Poppendieck, Mary & Tom “Lean Software Development” Addison Wesley (2003) ISBN: 0-32-115078-3

[TPS] Ohno, Taiichi “The Toyota Production System: Beyond Large-Scale Production”

Productivity Press Inc. (1995) ISBN: 0-915-2991-4-3

Chapter 2 Decision-Making Model

The previous chapter introduced the acceptance process and the three key decisions that need to be made as software is assessed for acceptability by whoever makes the acceptance decision: the Readiness Decision, the Acceptance Decision and the Usage Decision.

Figure 1 illustrates this simplified version of the decision making model.

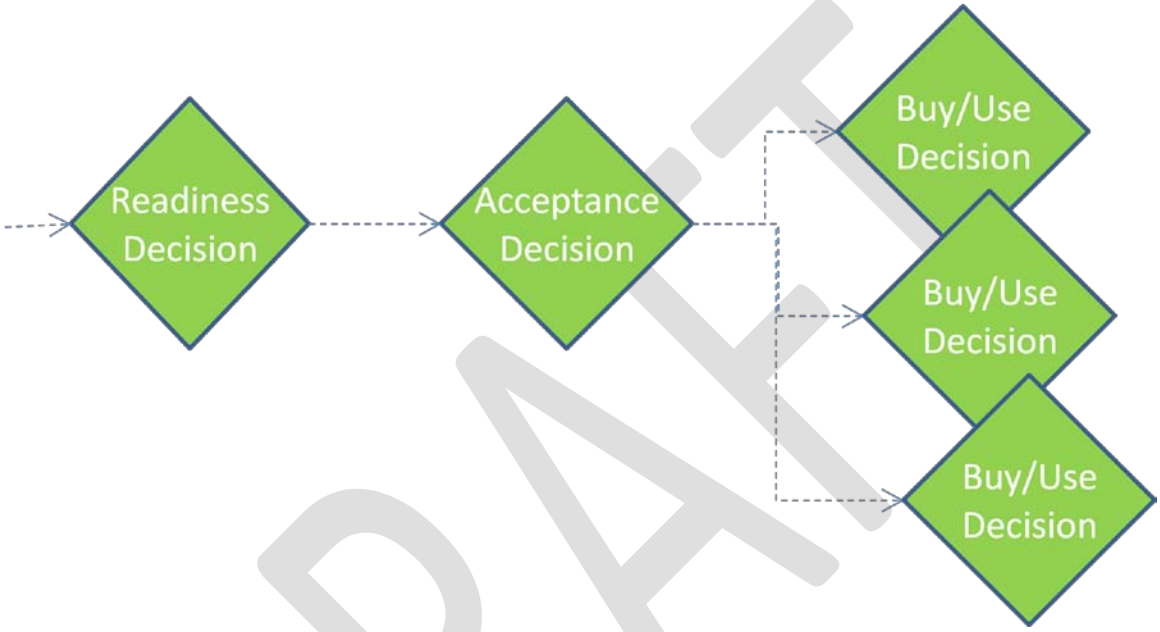


Figure 1
Simplified decision-making model

This section elaborates on *how* the first two decisions are made and *who* makes them in a variety of business models. The decisions are not made in a vacuum; they require information that must be made available through activities. Figure 2 illustrates this process for a single decision:

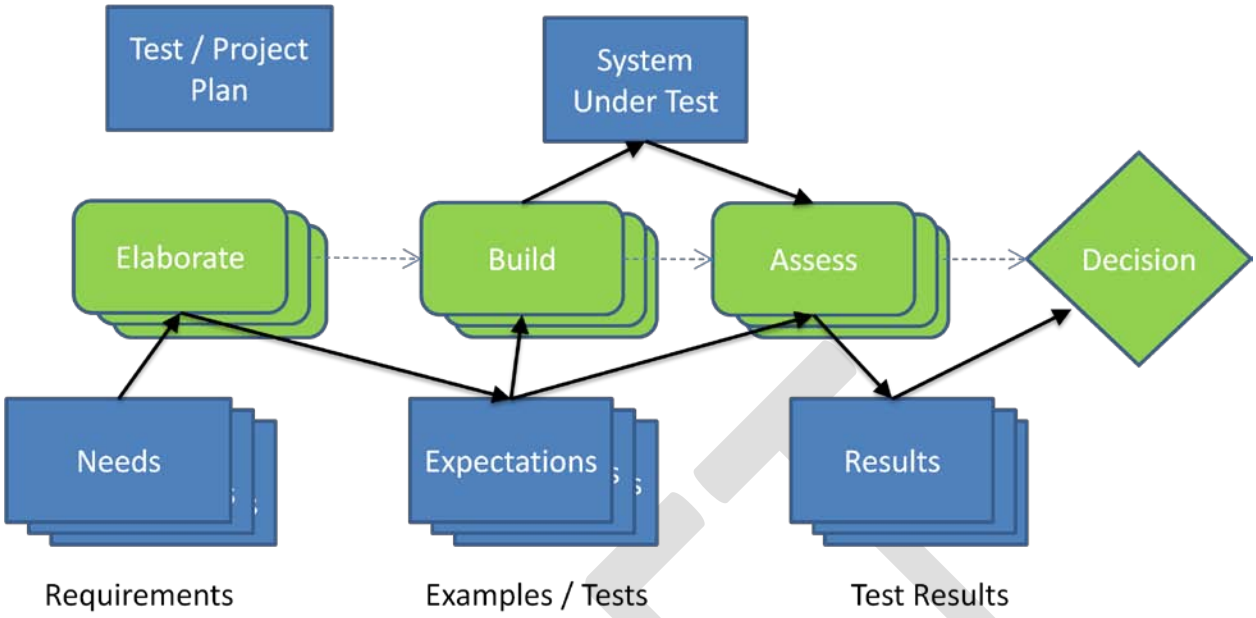


Figure 2

Decision-making model sample activities

The diamond on the right side of Figure 2 represents the decision to be made based on the test results (the decision can be either the readiness decision or the acceptance decision). The test results are based on the testing and assessment activities, which assesses the system-under-test against the expectations. The expectations of the system-under-test were defined based on the users' requirements. All of these activities are executed within the context of a test plan.

Many of the practices in Volume II describe how to do the assess activity, and other practices in Volume II describe ways to define the expectations based on the needs. That is one of the reasons this guide has a number of requirements-related practices—it is not about testing, it is about acceptance, and acceptance is based on expectations and requirements.

2.1 The Six Abstract Roles

The job titles of the decision makers vary greatly from business model to business model and across business domains and organizations, so this guide uses abstract role names to describe the roles within the decision making model. This guide also provides a list of common aliases. However, be aware that many of the names are highly overloaded and that your "customer" (to pick just one example) may be an entirely different role than the one mentioned as an alias here. To see how the abstract role names map to job titles within organizations in specific business models, see the sidebar "Decision-Making Model Stereotypes."

2.1.1 Readiness Decision-Maker

The readiness decision maker makes the final readiness decision based on input from others. When a single person performs this role, the job title might be something like Chief Engineer, Project Manager,

Development Manager, or VP of Engineering. This role could also be played by a committee, as is common in enterprise deployment with many stakeholders, or made by several people or committees in parallel as described in Complex Organizations in the previous chapter.

2.1.2 Development Team

The development team builds the software. Generally, this team may include user experience researchers and designers, graphic artists, requirements/business analysts, software architects, software developers, middleware and integration specialists, system administrators, DBAs, release/deployment specialists, testers and technical writers. In other words, this team includes anyone who is involved in any way in the actual construction, customization, or integration of the software.

2.1.3 Readiness Assessors

The readiness assessors assess the readiness of the software for acceptance testing. They provide information that is used to make the readiness decision. The job titles involved depends very much on the nature of the project and the organization, but it typically includes roles such as developers, testers, and documentation writers. In effect, a readiness assessor can be anyone who might be asked to provide an opinion on whether the software is ready. In some cases, this opinion is based on formal testing activities, but it might also be based on technical reviews or even qualitative inputs.

2.1.4 Acceptance Decision-Maker

The acceptance decision-maker is the person or committee who decides whether to accept the software. In a product company, a job title for this role might be Product Manager, but in an information technology (IT) environment, this role is typically filled by a customer, product owner, business lead, or business sponsor.

2.1.5 Acceptance Testers

Acceptance testers provide data on acceptability of the product. They perform activities to assess to what degree the product meets the expectations of the customer or end user. Acceptance testers may include two teams – one focusing on functional acceptance of the system, while another focusing on operational acceptance. They provide information to the acceptance decision maker. They may be dedicated testing staff, end users asked to do testing, or anywhere in between in skill set.

2.1.6 Users

Users make individual usage decisions. Each user decides whether to use the product as it is when it is shipped or deployed. Their feedback might be used to adjust the requirements for the next release or to do usability testing of the beta versions of the current release. People who are users may also be involved as acceptance testers or beta testers.

2.2 Making the Three Decisions

This section describes how the preceding six abstract roles are involved in making the three decisions.

2.2.1 Making the Readiness Decision

The readiness decision is made by the readiness decision maker(s). The readiness decision is an exit gate with a decision about whether to let the product be seen beyond the boundaries of the supplier organization. The decision is based on readiness assessment (which is based on the features included and the quality of those features) done by the readiness assessors. The decision can be made by a single person (such as a Chief Engineer) or by a committee (such as engineers, architects, or other project stakeholders). When it is not a single decision as described in Chapter 1 – The Acceptance Process, each decision may be made by a single person or a committee. From the point of each decision maker the software system is either ready or it is not ready. If it is not ready, there may be a list of concerns that need to be addressed before it will be considered ready. For more information, see How Will We Manage Concerns section in Chapter 16 – Planning for Acceptance.

There may have been a number of earlier decision-making checkpoints as part of the development process (such as "requirements complete," "design complete," or "code complete"). These are beyond the scope of this guide because they are neither directly part of the readiness decision nor are they easily tested.

2.2.2 Making the Acceptance Decision

The acceptance decision is made by the person (or persons) playing the Acceptance Decision Maker role. The decision is summarized by the question *"Should we accept the software and put it into use delivering value to our organization?"* There may be additional contractual consequences for making the acceptance decision, such as a commitment to pay the supplier, the start of a predefined warranty period, and so on. While in theory these should not be the primary considerations when making the decision, in practice they often are. The decision should be whether the software is " "complete" or "done" enough to be deployed or shipped. For more information about the definition of "done," see the section, "Doneness Model." For more information about the complete definition of the system attributes that may be considered when making the acceptance decision, see the section, "System Model."

The definition of "done" is influenced by several factors, including the following:

Minimum credible release (MCR) of functionality. What features or functions must the product support to be worth releasing? This is based on whatever criteria the product owner decides are important, derived from product plans, market surveys, competitive analysis, or economic analysis.

Minimum quality requirement (MQR) for the product. What is the level of quality that must be achieved before the product can be released? Quality has several dimensions. The presence or absence of bugs/defects in functionality is just one dimension of quality. Para-functional

requires, also known as quality attributes is another dimension. The MQR encompasses the latter while MCR encompasses the former.

Hard deadlines. By what date must a particular version of the product be released to have any value. These can include trade show dates, regulatory deadlines, or contractual obligations. Each deadline could be satisfied by a different version (or “build") of the product. For more information, see "Project Context Model."

The acceptance decision is made based on data acquired from a number of sources and activities. Acceptance testing generates much of the data needed to make the acceptance decision. This data includes the following:

Pass/fail results of all tests that were performed as part of the acceptance testing. This could verify both functional requirements and parafunctional requirements.

Feature completeness – As implied by the pass/fail results of functional tests

---

The acceptance decision may also use data gathered during readiness assessment. The most common example of this is data related to system performance (capacity, availability, etc.) which many customer organizations would not be capable of gathering themselves but which they are capable of understanding once it has been gathered.

The acceptance decision is all about maximizing value derived from the product by the customer and minimizing risk. Time has a direct value in that time spent collecting more data through testing has a direct cost (the cost of resources consumed in gathering the data) and an indirect cost (the deferral of benefit that can only be realized once the system is accepted). Risk has cost that could be calculated as the sum of the cost of all possible negative events multiplied by the probability of their occurrence. Though this kind of literal calculation is not frequently done our perceptions of risk are inherently based on an intuitive interpretation of the circumstances along these lines. The intent is for the acceptance decision maker to understand the trade offs and decide on whether we need more data, or we have done enough and we can make the decision. In doing this it is important to be aware of the two extreme negative possible outcomes, one in time, the other in risk. Examples of costs of risk might include the following:

Cost of patching software in the field

Cost of manual workarounds for bugs

Cost of maintaining specialized resources for software maintenance

Losing customers that need specific features that are missing

---

The cost of time can be non-linear in that if a deadline or need is missed, all benefits may be lost and punitive action taken. The risk the calculation is different for 'black swan' [BS] events of extremely low probability but extremely high impact such that again all value may be lost. These cannot be ignored in the acceptance of new software. In concept, when the cost of risk exceeds the cost of delay, more

testing should be performed. When the cost of more testing exceeds the risk-related cost that would be reduced (by reducing probability of one or more events occurring or by reducing the expected cost given the event does occur), you can decide to accept the product without further testing.

2.3 Making the Usage Decision

Each potential user of the system has to make a personal decision about whether to use the software. This decision is different from the acceptance decision in that it is made many times by different people or organizations. In fact, there may be several tiers of these decisions as companies decide whether to adopt a product (or a new version thereof) and departments or individuals decide whether to comply with the organizational decision. The important consideration from the perspective of this guide is that these decisions happen after the acceptance decision and do not directly influence the acceptance decision. They may indirectly influence it in one of the following two ways:

Proactively. Usage decisions may indirectly influence the acceptance decision for an upcoming release by future users communicating the individual acceptance criteria to the product owner in response to market research or surveys. This type of criteria may also be submitted to the product owner through unsolicited inputs, such as feature requests or bug reports.

Retroactively. Usage decisions may indirectly influence the acceptance decision by providing feedback on the released product indicating a lack of satisfaction in either functionality or quality. This may influence the acceptance decision criteria of a future release, but it rarely causes the acceptance decision already made to be revisited. The notable exception would be the discovery of "severity 1" bugs in critical functionality that might result in a recall of the release software.

2.4 Roles vs. Organizations

The roles described in this decision-making model may be played by people in several different organizations. The primary value of discussing organization here is in making it easier to map terminology from various organization models to better understand who plays which decision-making role. If the organizational model does not help in this endeavour, it can be ignored.

When the software is being built by a different organization than the one who commissioned its construction, the organization that commissioned the software is often referred to as the customer, and the organization that is building the software is the supplier. This is true whether the organizations in question are separate, unrelated companies or simply departments within a single company. For example, the IT department is typically a supplier of systems to the core business departments (such as Transportation or Manufacturing) and supporting departments (such as Human Resources or Finance.)

When acceptance testing is outsourced to a third-party test organization, it is often referred to as the (third-party) test lab. The test lab is a supplier of services as distinguished from the supplier of the software.

An organization that buys and deploys shrink-wrapped software can also be referred to as a customer, and the organization they buy it from may be referred to as the vendor or supplier. The fact that the vendor contracts the work to an outsourcer (another vendor of which they are the customer) illustrates the problem with using the term "customer" to describe the acceptance decision maker (the product owner) as advocated in extreme programming - which customer is it referring to?

Figure X illustrates this problem. The Purchaser (in the role of Customer) buys shrink-wrap from S/W Vendor (in role of Supplier). S/W Vendor (in the role of Customer) outsources development to S/W Developer (the Supplier). S/W Developer (as Customer) outsources readiness assessment to Test Lab (as Supplier). So we have three separate customer-supplier relationships in this scenario making it hard to tell which party we are talking about when we refer to the “customer”.

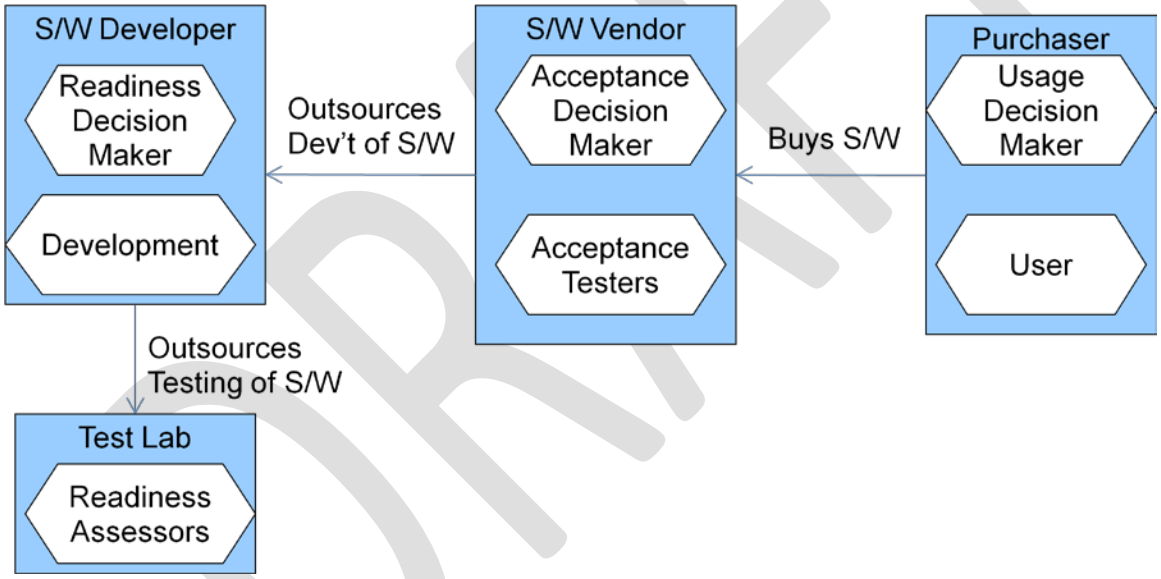


Figure x
Multiple Customers and Suppliers

2.4.1 Who Plays Which Roles?

Thus far the discussions have centered on the abstract roles involved in the decision making process. But who actually plays the roles.

Who Plays the Readiness Decision Making Role?

The Readiness Decision Maker role is typically performed by someone in the development organization. They are the person who ultimately decides whether the software is ready to be shown to the acceptance testers. Typical job titles include Development lead, Project Manager, Director of

Development, Chief Engineer. On agile projects where this decision is made separately for each feature, the feature-level Readiness Decision Maker is often the developer who signs up and implements the story/feature.

Who Plays the Acceptance Decision Making Role?

The Acceptance Decision Maker role is typically performed by someone in the business side of the organization when development is done internally or by someone in the customer organization when development is outsourced. They're the person who ultimately decides whether the software is ready to be shown to the acceptance testers. Typical job titles for internally developed software include Business Lead, Product Manager, or some kind of business title. On agile projects where this decision is made separately for each feature, the feature-level Acceptance Decision Maker is often the specific business subject matter expert who provided the detailed description of the functionality to be built.

Who Does What Testing?

Readiness assessment is most commonly done (or at least it should be done) by the developers of the software. But are they the only ones doing readiness assessment? Acceptance testing should be done by real users of the software but what if the real users are anonymous? In Figure X – Testing Role Players, each cell describes who does which testing in a common project context. The top left triangle in the cell indicates who is doing readiness assessment and the bottom right triangle indicates who is involved in acceptance testing.

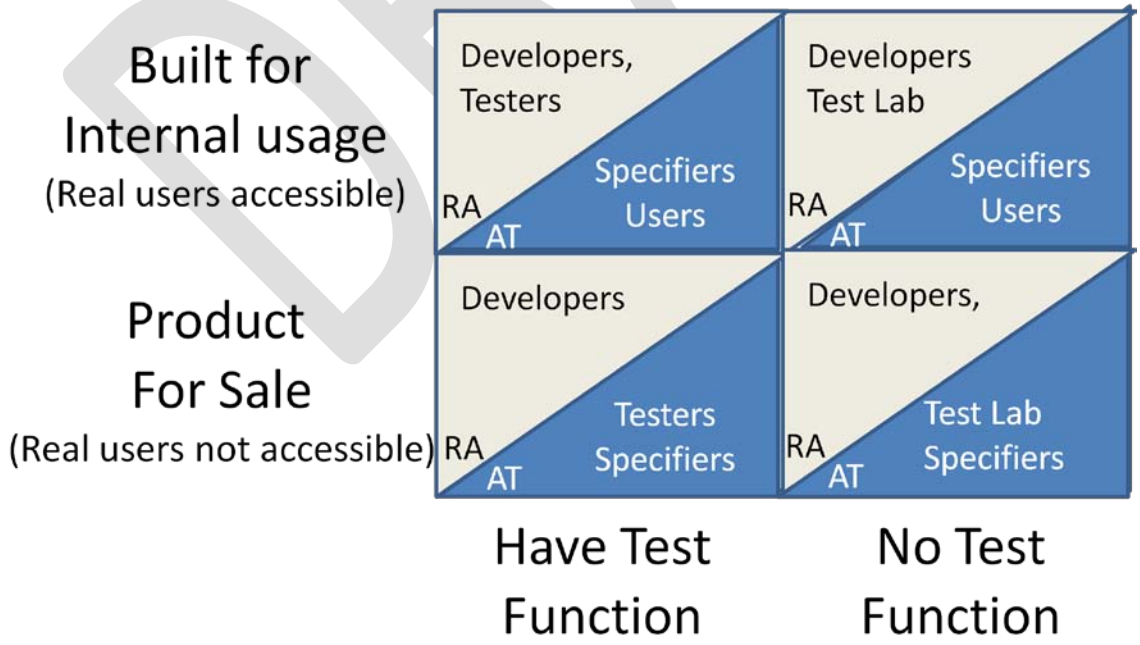


Figure X – Testing Role Players

When software is being built by an organization for its own use, the users should be available for participating in the acceptance testing. If there is no testing function, the business typically expects development to test the software thoroughly before handing it over. If there is a testing function, they would typically participate in a 2nd phase of readiness assessment, often called "system testing" and "integration testing" before handing the software over to the users for "user acceptance testing" (UAT) or "business acceptance testing" (BAT).

When software is being built for sale to anonymous customers, real users don't take part in acceptance testing. Instead, the testing organization does acceptance testing on their behalf acting as a proxy (or surrogate) user. If there is no testing organization, the testing could be done by the product owner's team of product specifiers or the acceptance testing could be outsourced to a 3rd party test lab.

2.5 Summary

The Readiness Decision is made by someone or some committee from the supplier playing the role of Readiness Decision Maker based on information gathered by Readiness Assessors. Readiness Assessors typically include developers, architects and sometimes testers. A key motivation of the Readiness Decision Maker is to protect the reputation of the supplier organization by ensuring that the product has enough functionality and is of good enough quality to expose to the customer without embarrassing the supplier. The RDM is typically a senior person in the supplier organization.

The Acceptance Decision is made by a customer (or possibly several) playing the role of Acceptance Decision Maker based on information gathered by people playing the role of Acceptance Tester as well as any Readiness Assessment information shared by the supplier. The ADM is usually a key person in the customer organization.The acceptance testers may be end users, representatives of the end users such as the specifiers of the requirements, or testers) when end users are not available to do acceptance testing. When the customer organizations has commissioned the production of a product and they have no internal test capability, they may choose to outsource some of the acceptance testing to a third-party test lab.

2.6 References

[SCM] Berczuk, Stephen P. and Brad Appleton. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison Wesley Professional. 2002.

[BS] Taleb, Nassim Nicholas. *The Black Swan: The Impact of the Highly Improbable.* Random House. 2007

Chapter 3 Project Context Model

The Project Context Model is a way to better understand the goals of the project and the constraints under which it must operate. It is not a formal model; instead, it is a set of information to be collected and factored into other activities. It includes information such as the following:

Usage Context. What is the overall context in which the software will exist?

Business goals. An example question to gather information for these is, "What is the business value to be provided by the system and how is it to be achieved (strategy)?"

Scope. An example question to gather information for this is "What functionality is in scope and what is out of scope for this project?"

Stakeholders and users. An example question to gather information for these is "Who are the intended users and who are the other project and system stakeholders?"

Budget. An example question to gather information for this is "How much money is available to achieve the business goal?"

Hard deadlines. An example question to gather information for these is "What deadlines must be met for the project to be considered a success?" Examples of hard deadlines include trade shows, contractual deadlines, and regulatory deadlines.

Constraints. An example question to gather information for these is "What resources (such as people, space, and equipment) are available to the project?" A follow-up question might include "Which resources are negotiable and which are hard constraints?"

The next sections provide more information about each of these.

3.1 Usage Context

Software can be built for many different kinds of usage and the organization that builds or commissions it affects how the software is built and accepted. Figure 1 – The Product Context illustrates one way of classifying the software-intensive systems we build.

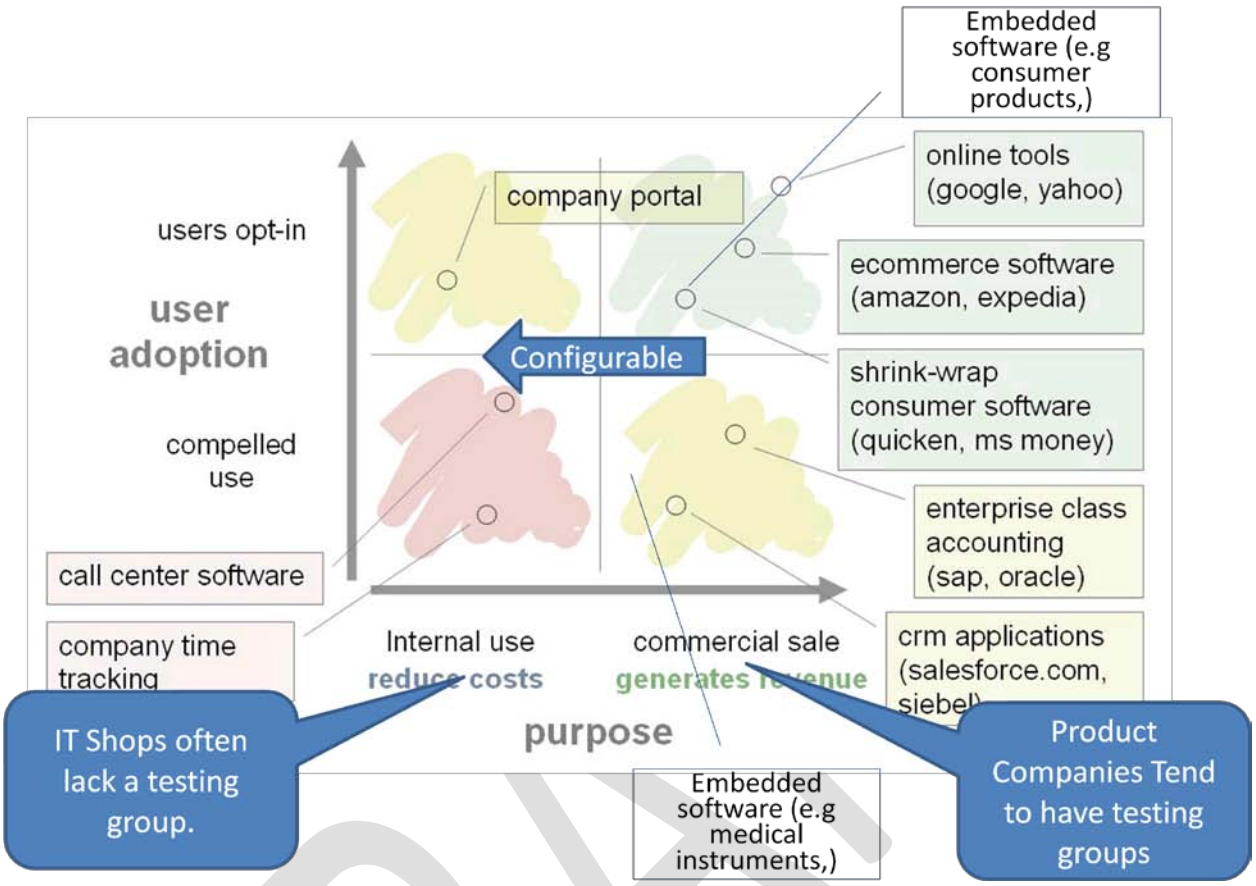


Figure 1 – The Product Context (courtesy of Jeff Patton)

Some organizations build software for their own internal usage to reduce costs while others build products to sell to others to generate revenue. The left side of the quadrant diagram describes what is commonly called enterprise software; software that is built by the enterprise for its own use. The right side of the diagram represents software that is built for sale to other parties either as software in its own right (which may be installed on site or software-as-a-service) or as software embedded in a hardware product. Some of the products built on the right side of the diagram show up as tools, components, frameworks or products used in the construction of the enterprise products on the left side of the diagram. Independently of this, the users of the software may be captive users who are compelled to use the product once it is available or they may choose to use the product or a competing product. The process of accepting software can vary based on this context.

3.1.1 Enterprise Systems

Enterprise systems are commissioned by an enterprise for its own internal use. The systems automate all or parts of various business processes. The use cases of each system implement one or more steps of a business process, either in whole or in part as illustrated by the left side of Figure 2. They may be built from scratch by writing code, composed of purchased components or created by configuring a packaged product purchased from a vendor (on the right side of Figure 2.) The work may be done by a part of the

enterprise, often called the Information Technology (IT) or Information Systems (IS) department, or the development may be outsourced to a third party. Many of these systems tend to be integrated with other systems within the enterprise (which often include legacy systems) and a large part of the challenge of building and accepting these systems is ensuring that the integration works correctly as illustrated by the right side of Figure2. The integration requirements may take many forms including data pumps and transactional interactions to name just a few, but all should be driven from the business requirements. Acceptance of enterprise systems involves determining whether or not the system will meet its business-enablement or cost reduction targets.

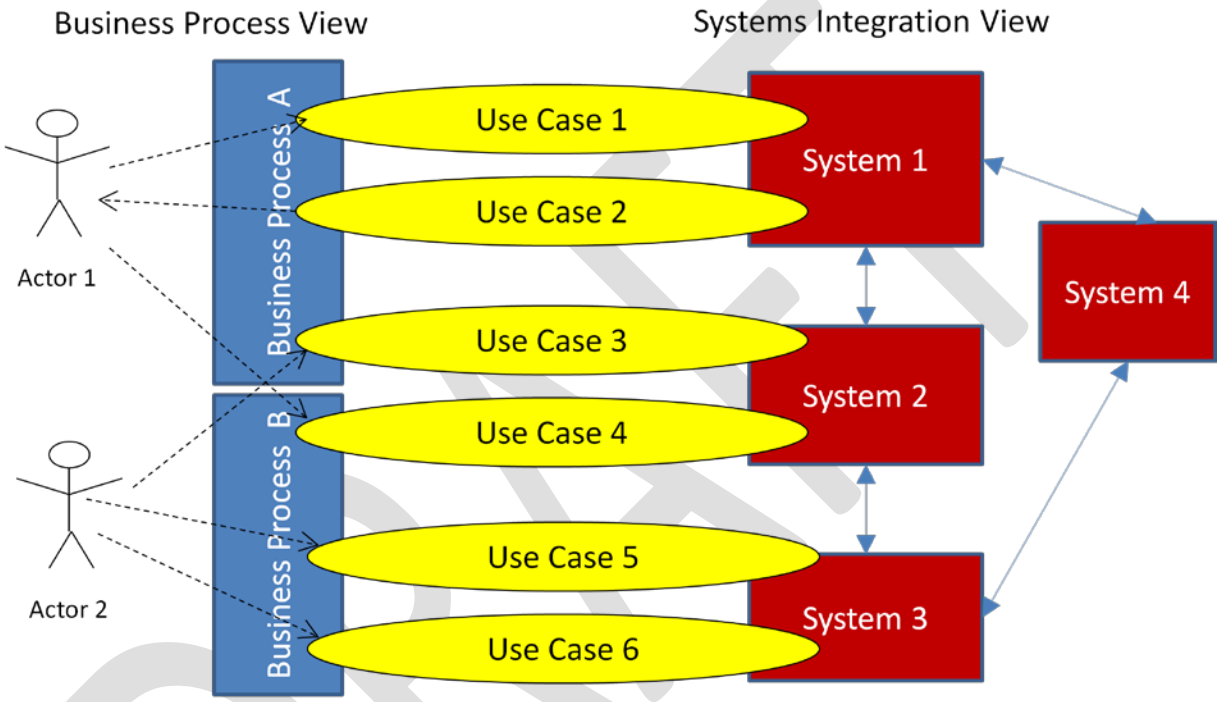


Figure 2 – Business Process vs. Systems Integration View

Because the software is commissioned by the enterprise itself, there is the potential for a large degree of business involvement in the development process. This can result in software that is a very good fit for the business. Unfortunately, many businesses delegate the hard work of defining their requirements to the supplier, whether internal or outsourced, and effectively abdicate control over what is built. This often results in barely usable systems that have long, painful acceptance phases. The best way to avoid this is to have strong business ownership and stewardship of the development of these systems in the form of a *product owner* who makes all the critical decisions about what the systems should support and what it should not. This position goes by various formal titles such as business sponsor, business lead or customer.

The software in the bottom left quadrant is typically embedded in a business workflow implementing one or many steps of the business process. The users do not have a choice of whether or not to use the software because using it is part of their job. Users are highly accessible therefore can be used for usability testing and acceptance testing but because usage is non-optional there is an unfortunate

tendency of the product owner to ignore the usability attributes of the system. The users are often involved in the acceptance testing in an activity often call user acceptance testing (UAT).

The software in the top left quadrant is different in that users are not compelled to use the product even though it was built for internal use. The product owner tries to represent the needs of the real user who may not be accessible. For example, the software may be used primarily by in-house users who are accessible, or it may be designed for the use of people outside the company including customers or business partners. The system may be standalone but is more likely to be integrated behind the scenes with other software within the enterprise.

3.1.2 Software-Intensive Products

Many products in the modern world, from toys to automobiles, have a software component and the amount of such embedded components is steadily increasing, especially with regards to the portion of the value-added to the final product. This software is what provides the complex behaviors that characterize products in the 21st century. These products have been a large part of the growth of modern economies. Acceptance of these products involves deciding whether the product as built will meet its revenue targets, contribute or detract from the brand image, and possibly other factors. The product requirements are typically driven by a product manager.

The software in the bottom right quadrant is built by one company for sale to another company. This software may be purchased and used as-is. Or it may be designed to be configured or customized by or for the customer to fit within their enterprise environment (on the left side of the diagram.) The notion of acceptance then has an interesting twist in that the generic product must first be accepted by the product manager before being made available to the customers who may then have it configured or customized before accepting it for deployment within their organizations. This is described in more detail in section Accepting Customizable Products in Chapter 1 -- The Acceptance Process.

The software in the top right quadrant may be purely software for use by consumers (so called “shrink wrapped” software) or it may be embedded in a hardware product. In either case the end user is too numerous (and is often not available) to involve in the acceptance process. Note that there may be additional specialized techniques used for building and assessing embedded products and these specialized techniques are out of scope for this guide but many of the techniques described here do apply to embedded products.

3.2 Business Goals

Too many projects are run with a majority of the staff not understanding (or even caring about) the business goals of the project. Sometimes this is the result of deliberate decisions by management to withhold information, and sometimes it is just an oversight. Either way, expect to get suboptimal results if each team member is focused on optimizing his or her own job assignment instead of ensuring that business goals are achieved.

As a bare minimum, everyone on the project should have a clear understanding of what the project is expected to deliver and how that will provide value to the business. Example of different ways to add value to the business include cost reduction, increased sales, more satisfied users, and improved market perception/branding.

3.3 Scope

The scope of the project should be directly influenced by business goals. At the broadest levels, the scope can be defined in terms on the types of users we expect to support and the types of functionality we will supply them. It is just as important to enumerate what we do not plan to deliver as what we will deliver; otherwise we risk wasting a lot of time and energy discussing and possibly building and testing functionality that was supposed to be excluded.

3.4 Stakeholders and Users

Accepting software requires the involvement of various stakeholders. Some are directly involved in the acceptance testing and decision-making process, while others need to have their interests protected even though they are not involved.

3.4.1 Users

The most obvious usage goal holders are the people (or systems) who are to use the core functionality of the system. Typically, these people are referred to as "the users." They are the ones who use a Web site for doing their banking, entertainment, or online shopping; use an application for executing one or more steps of a business process; or operate a combined software/hardware product such as a medical imaging system. But these are not the only people interacting with the system. Other users include the people who administer the system by populating the catalog of the online store or set up the content in an online entertainment system; the people who run diagnostics on and maintain the system. There are also people (or systems) who install the system, start it, monitor its status, and shut it down when the servers need maintenance. They all have requirements with respect to how they use the system. For more information about users and their goals, see the User Modeling activity in Volume 2 of this series.

3.4.2 System Stakeholders

There are also stakeholders [ACUC] who will not directly interact with the system when it is in operation but who expect the system to look out for their interests as it is used (or abused) by others. For example, a system that contains personal information about someone has that person as a stakeholder even if they themselves do not directly interact with the system. This is an example of a parafunctional requirement.

3.4.3 Project Stakeholders

Products and IT systems are usually developed within a project. There may be many defined roles within the project, some of which may be played by users or system stakeholders and some by unique parties. These project stakeholders may be involved in the acceptance testing or acceptance decision-making process without being a direct system stakeholder or user. For example, the product manager or product owner may never use the system, nor be a system stakeholder, but they definitely have a stake in the acceptance decision process. The business sponsor of an IT project may never use the system or even view a report it generates, but the person in this role has a clear stake in the outcome in the form of the expected business benefits in return for the investment of time and money.

3.4.4 Communication Between Stakeholders

The likelihood of acceptance of a software-intensive system is directly proportional to the effectiveness of the communication between who commissioned the software (loosely referred to as the "customer") and the team building the software. The requirements and the tests are part of this communication. The communication is made much more effective if there is a common Ubiquitous Language that everyone agrees to use consistently.

3.5 Budget

Most projects have a fixed budget. This tends to constrain the number of people who will work on the project both in number and in rate of pay. Once the team size and composition is defined, the team will exhibit a well-defined burn rate and any extension of the project timelines due to late delivery of the software will translate into budget overruns unless a significant buffer is held back when planning the project.

3.6 Hard Deadlines

All projects face deadlines of one sort or another. Some projects routinely miss many of these deadlines. When the business consequence of missing a deadline is significant, we should take the appropriate measures to ensure that the deadlines are not missed, such as a planning contingency buffer or cutting scope. Therefore, it is essential to understand which deadlines are arbitrary (such as in "It would be really great if we could have that functionality by September") or critical (such as in "We need to demonstrate this functionality at the trade show in September and failure to do so could significantly affect our fourth quarter sales and our share price.") In the former case, missing the deadline may result in slight or even no inconvenience. In the latter case, the value of the delivery could diminish to zero making the product and the project that delivers it completely irrelevant.

3.6.1 Recovery from Development Schedule Slippage

A major issue with hard delivery deadlines relates to the perceived compressibility of the acceptance testing phase (and testing in general.) This is particularly an issue when the delivery date is fixed and

immovable. When the development phase is running late, as it often does on waterfall project, there is great pressure on the entire team to recover the schedule slippage somehow. There are two techniques commonly employed, each with their own difficulties.

Compressing the Test Cycle

Readiness assessment and acceptance testing will typically start late but is often expected to make up the schedule slippage by reducing the length of the test cycle. This results in less testing than planned and therefore likely more bugs slip through to production. In other words, the acceptance decision is made based on incomplete data about the quality of the product.

Overlapping Test Cycle with Development

The other schedule recovery technique that is often used with hard deadlines is to ask the testers to start testing the product before it is completely finished. While this -- Incremental Acceptance Testing-- is considered normal practice on an agile project, a project planned as a waterfall project cannot adopt this practice easily mid project because several critical success factors employed on agile projects are often missing. For example, many features may be 80% done rather than 80% of the features being completely done and testable. Therefore it is hard for the testers to know what to test because there isn't a clear list of what should be working now. Another issue is that Incremental Acceptance Testing usually includes extensive automated regression testing so that the testers can focus on the newly completed functionality rather than having to manually retest everything. The automated regression testing is also a key form of bug repellent used by the developers to avoid introducing regression bugs by detecting these bugs as part of readiness assessment so they can be fixed before the new release candidate is passed on to the testers.

3.7 Constraints

Most projects operate under some kind of constraints. Constraints can include people and skills or facilities and equipment. Sometimes these constraints can be loosened, while they are strictly limited at other times. A company may not be able to hire additional staff or recruit people with specific skills; in these cases we need to create the best possible plan that allows us to meet our goals without hiring additional staff. This may result in very different plans than if we were able to hire additional staff.

3.8 Resources

[ACUC] Alistair Cockburn "Writing Effective Uses Cases" Addison Wesley

Chapter 4 System Requirements Model

The process of accepting a software-intensive system is inherently dependent on the system tested and the test cases used. Your test cases are designed to test for specific requirements. Therefore before designing your test cases it is imperative you know the requirements that they are intended to test.

4.1 Requirements and Acceptance

The acceptance of a software-intensive system is clearly related to whether it meets the requirements. Assessing whether it meets the requirements involves the process of testing the software using test cases that are in some way related to the requirements. Therefore, requirements are an important component of the acceptance process even if they are not directly a testing-related artifact. This guide series includes some key requirements-gathering techniques to illustrate how the requirements are related to testing and acceptance.

The term ”requirements“ is somewhat contentious. Some people believe that you merely need to talk to potential users and ask them for their requirements. Frequently, this is referred to as requirements elicitation or requirements gathering. This is illustrated in Figure 1.

<TBA: a picture showing users providing Requirements consisting of Use Cases or User Stories>

Figure 1
Gathering Requirements directly from users

Some people believe that requirements cannot be gathered like strawberries or mushrooms; instead, they believe requirements must be based on the definition of a product that is designed to meet the potential users' needs. This guide summarizes this process as product design which acts as a placeholder for a wide range of activities that may involve specialized product design skills. This process is illustrated in Figure 2. For a serious treatment of the topics of product design and experience design see [HSPK] as well as effective low-fidelity product experience design techniques by [BB].

<TBA: a picture showing users needs being fed into a Product Design activity which then leads to Requirements consisting of Use Cases or User Stories>

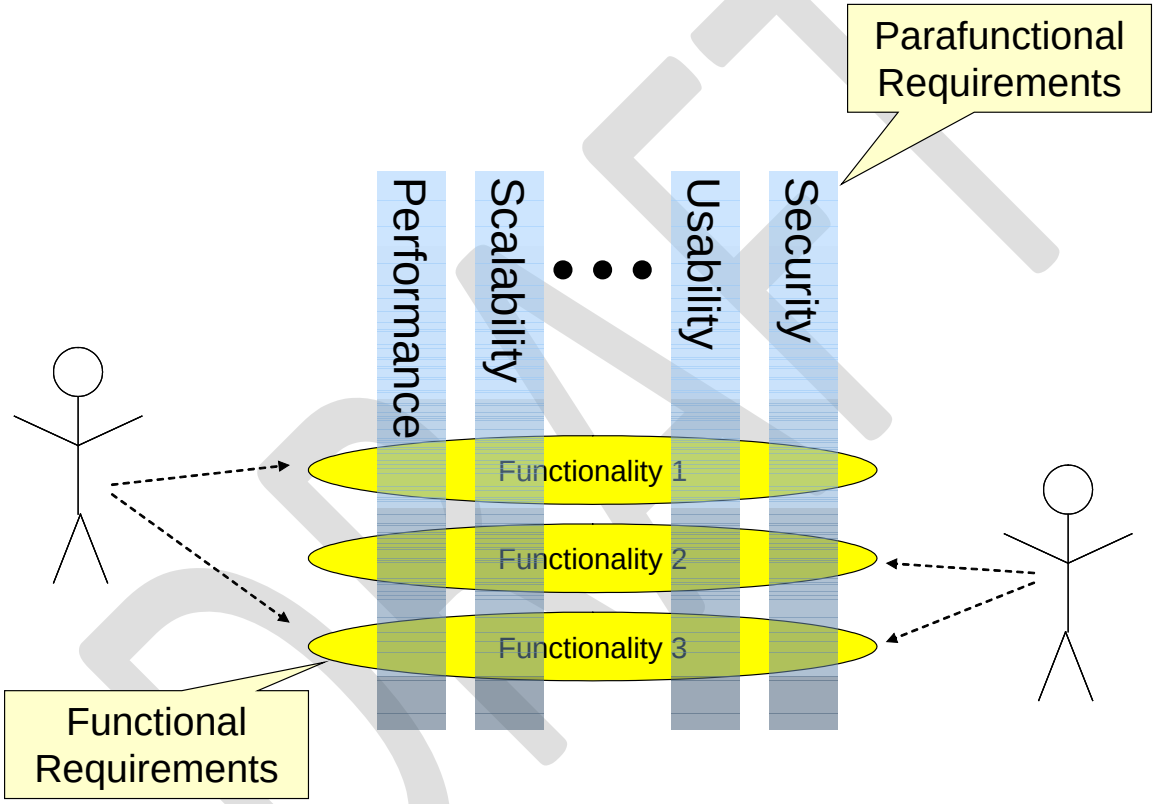
Figure 2
Getting Requirements via Product Design

A related topic is how we verify that the requirements truly satisfy the needs to the users. This can be done as part of an acceptance testing phase, but that is rather late to discover that what you built is going to require significant changes before it will meet the users' needs. Therefore, this guide advocates acceptance testing of the proposed product design (not the software design) before the software is built. Techniques such as experience sketching, paper prototyping and Wizard of Oz can be used to verify that you are "building the right system" very early in the project while there is still time to adjust

the product design. For information about these techniques, see the Usability Testing thumbnail in Volume 2.

4.1.1 Types of Requirements

The requirements, however derived, are typically divided into two broad categories, functional requirements and para-functional requirements (also known as extra-functional requirements, nonfunctional requirements, system attributes or quality attributes.) Functional requirements describe the functionality to be provided to users or administrators of the software-intensive system. Non-functional requirements transcend the functionality. Different techniques are used for describing the various kinds of requirements and for verifying that those requirements are met.



<TBA: In the final version, the figure should have the following added to it:

1) Business workflow tying together several use cases

2) System actors (integration requirements)>

Figure 3

Functional vs. para-functional requirements

4.1.2 Functional Requirements

Functional requirements describe how various types of users expect the system to help them do their jobs. The functional requirements, whether explored and gathered directly or derived from a product design, can be organized and communicated a number of different ways, including the following:

- Business processes
- Use cases
- User stories
- Feature lists
- Scenarios
- Protocol specifications
- Functional specifications
- State models

Figure 4 illustrates a very high level system requirement model that includes business workflow tying together several use cases. The requirements include the need to interact with System 4 as part of some of the use cases. It also requires Systems 1, 2 and 3 to exchange information created or modified during the workflow.

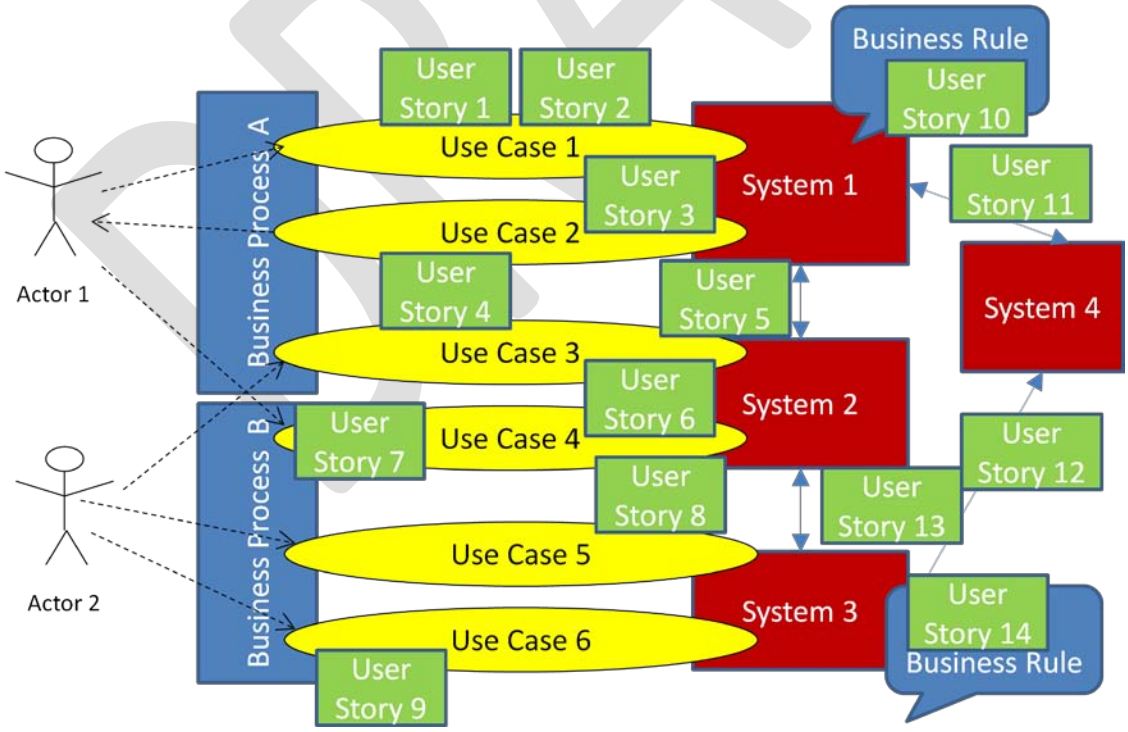


Figure 4

Requirements involving Users, Use Cases and User Stories

This guide highlights a small set of popular requirements practices (see Volume 2). They are used to illustrate how the requirements practices and artifacts are related to the tests you may use during acceptance of a software-intensive system. The User Modeling activity is used to capture salient information about the users of the software. It can help the development team understand the requirements better by understanding how the users think and what motivates their behavior. Use Case Modeling is used to capture what the users want the system to help them achieve. Each use case describes one kind of interaction (e.g. transaction) between a user and the system. A use case typically requires many test cases but often cannot be tested without testing other use cases. User Stories are used to break the functionality of the system into very small but still useful and testable chunks that can be built in just a few days or weeks. They are typically much smaller than use cases but may span 2 or more use cases. They might describe a business rule or even just one scenario of an inter-system communication. User stories map to one or more test cases.

These requirements activities and artifacts help us understand the types of tests we may need to execute to gather the data on which the acceptance decision is made.

4.1.3 Para-functionalPara-functional Requirements

Para-functional requirements are requirements that describe general qualities or behaviors of the system that span specific usage scenarios. Frequently, these requirements are traced back to protection of various stakeholders who may or may not be users of the system in question.

Unlike functional requirements, which vary greatly from system to system, there is a fairly standard list of types of para-functional requirements. Many of these requirements end with the suffix "ility" because they describe characteristics the whole system needs to support either while it is in use or as part of its overall system life cycle.

Requirements that the system needs to satisfy while it is operating include:

- **Availability** – When does the system need to be available for use?
- **Data Integrity** –The data needs to be stored and processed in a way that it can be reliably retrieved without changing its original value.
- **Safety** – The system should not cause physical or emotional harm to a user or stakeholder.
- **Recoverability** – When it fails, the system should restore its previous state without undue hardship to the users or supporters of platform.
- **Accessibility** – People with diverse limitations should be able to use the system. It may need to conform to standards like those stated by the ADA or other regulatory or standards bodies.
- **Supportability** – It should be economical to provide support to users of the product.
- **Reliability** –It should work well and resist failure in all situations.
- **Robustness** – It should take abuse and still function (fault tolerance).

- **Usability** – It should be easy to use by its intended users.
- **Security** –The product should protect against unauthorized use or intrusion.
- **Scalability** – The product should be capable of being scaled up or down to accommodate more users or transactions.
- **Performance** – What speed or throughput benchmarks does it need to meet?
- **Installability** – The product should be easy to install onto a target platform.
- **Compatibility** – With what external components and configurations does it need to work?

Requirements related to the total cost of ownership or product life cycle include:

- **Testability** – In what ways should the product be testable?
- **Maintainability** – How easy should it be to evolve, fix, or enhance the product?
- **Portability** – How easy should it be to port or reuse the technology elsewhere?
- **Localizability** – How economical will it be to publish the product in another language?
- **Reusability** – How easily can source code (e.g. classes, functions,etc.) be used in other circumstances?
- **Extensibility** – How economical should it be to enhance the product with other features and add-ons?
- **Configurability** – How easy should it be to prepare the product to be used on a variety of different platforms or for slightly different uses and operations?

The preceding list of types of para-functional requirements is not intended to be exhaustive. It also is not intended to be universal; some of these requirements may be irrelevant for some software-intensive systems. Part of the art of requirements gathering or engineering is deciding which ones are important and which ones are not. Those that are important need to be made explicit and incorporated into the test plans; those that are not important can probably be ignored (but the risk of ignoring them should be assessed before doing so.)

Most of the para-functional requirements cut across the use cases of the system. That is, they apply to many, if not all, of the discrete chunks of functionality described in the functional requirements. Note that some forms of para-functional requirements can be described at least partially in functional terms; security is a good example. You can say that User Role X should be prevented from changing the value of field F on screen S.

The key to testing conformance with para-functional requirements is the classification of each of these requirements indicating to what degree the project stakeholders care about the requirement. For example, for an application one builds for one's personal use, one may not care about scalability because there will be only one user, but on a large e-commerce application, scalability is very important. It is worth reviewing this list of para-functional requirements and consciously deciding how important

each one is to the success of your product or project. The following table lists the goals, importance, and rationale of a variety of para-functional attributes for a specific hypothetical system.

Attribute	Goal	Importance	Rationale
Web site performance under load	Less than 500 milliseconds response time	High	Major source of revenue
Web server capacity under load	At least 300 transactions per second. Graceful degradation under load	Medium	Large number of users
Reliability/availability	7x24x52	Critical	Users require instant satisfaction when worried about their money
Usability	Easily discoverable	High	Most users will use infrequently

Appendix X – Testing Activities & Responsibilities is an example of a checklist that can be used to record these decisions.

4.2 Summary

The readiness assessment and acceptance testing activities gather data about how well the system satisfies the requirements. The requirements include both functional and para-functional requirements. There are a number of alternative ways to decompose the requirements into manageable chunks. Some forms of requirements map more directly to test cases than others.

4.3 What's Next?

Things can and do go wrong when building software-intensive systems. Commonly occurring problems include misunderstood requirements and failure to meet para-functional requirements. The next chapter describes techniques for identifying what might go wrong and planning readiness assessment and acceptance testing activities that would discover them as early as possible.

4.4 REFERENCES

[HSPK] Hendrik Schifferstein, Paul Hekkert (Eds.) Product Experience. Elsevier, 2008.

[BB] Bill Buxton. Sketching User Experiences: Getting the Design Right and the Right Design. Mogran Kauffman, 2007.

Chapter 5 Risk Model

Every project inherent risks whether or not we recognize them. Determining what those risks are can be a challenge. Common risks on software-intensive projects relate to whether the development team understands what the customer wants built and how to build it. A risk model can help us understand all the risks and come up with risk mitigation plans to reduce their likelihood or impact.

5.1 What Could Possibly Go Wrong? Risk Assessment

One way to define risk is by asking what keeps us awake at night? More specifically, what might happen and what would be the consequences if it did happen?

We can make the discussion of risk more meaningful by translating nebulous concerns into concrete events that *could* happen and talking about the *likelihood* that it *might* happen and the *consequences* if it *does* happen.

For example, suppose we ordered some critical hardware which we require to conduct certain types of acceptance testing without which we are not prepared to make the acceptance decision. What could possibly go wrong? The following are some examples:

- The hardware could be destroyed in transit.
- The wrong hardware could be shipped either through an ordering error or a fulfillment error.
- The hardware could be defective.

For each of the preceding events we can estimate the likelihood that it will occur and assess the impact on our project if it did occur. Performing these two calculations separately helps us to better understand the risk as shown in the risk matrix in Figure 1. As the probability of an event increases, the risk increases. As the consequences of an event increases, the risk increases.

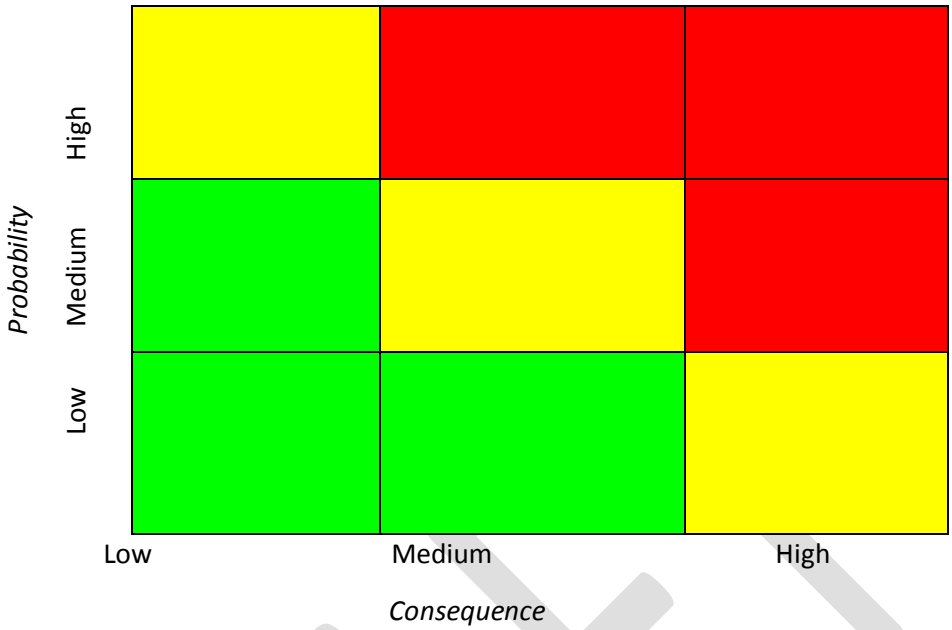


Figure 1
Risk Matrix

In Figure 1 – Risk Matrix, the two areas on either side of the diagonal and the diagonal itself represent three degrees of risk. The bottom left (green) risk regime represents low risk, the top right (red) risk regime represents high risk, and the top-left to bottom-right diagonal (yellow) risk regime represents moderate risk. In general, risks that fall in the same risk regime are equally important to mitigate.

5.2 Should We Do Something About It? Risk Management

After we understand the risks of our project, what can we do about them? There are three possible courses of action:

- We can accept the probability and consequence that a particular event might happen.
- We can perform activities to reduce the likelihood of it happening.
- We can perform activities to reduce the consequence if it does happen.

The course we choose depends on a number of factors, including the following:

- The factors we have control over, such as the following:
    - If there are no courses of action that could reduce the likelihood of something happening we may be forced to focus on trying to reduce the consequence. For example, the only way to avoid an extreme weather event might be to move to a

different area, which may simply exchange one set of extreme weather events for a different set.

    - If there is no way to reduce the consequence of an event, we need to focus on reducing the likelihood of it occurring. For example, most people agree that it is better (and economically more feasible) to try to reduce the likelihood of a heart attack by exercising and eating well than to try to improve the probability of surviving it by hiring a heart specialist to be at your side at all times.

- The relative cost of the options available to use. If it is much cheaper to reduce the likelihood than the consequence, we should first focus on lowering the likelihood and vice versa. Note that the cost is typically non-linear and gets more expensive the closer to zero we try to drive the likelihood or consequence.
- The cost of risk reduction relative to the cost we would incur if the risk occurred. For example, if a parking ticket costs twice as much as paying for the parking and there is only a 20 percent chance of getting caught and ticketed, we may choose to take the chance by not paying for parking.

5.3 How Can Testing Help? Risk Mitigation Strategies

If we decide to mitigate a risk, how we go about it depends on the nature of the risk. Risks that relate to the possibility of delivering a defective product are amenable to risk mitigation through some form of testing. Risks that relate to discovering something too late for a timely fix can be mitigated by activities that move the discovery earlier in the project.

5.3.1 Doing Something Earlier

Many risks on projects are related to time. Will something happen in time? If it happens too late, will we have time to react without affecting the project timeline?

A good example of this is the late discovery of missed or misunderstood requirements. When this discovery occurs during the acceptance testing phase of a project shortly before the product is expected to be turned over to users, the impact (of the discovery) may be a significant delay in achieving the business benefits expected from the system. In this case, we can reduce the impact of the discovery by doing the acceptance testing activities earlier in the project.

The incremental acceptance testing practice used on many agile projects is one way to move discovery of problems such as misunderstood requirements earlier in the project so there is plenty of time to address them. Document-driven projects can also reap the benefits of incremental acceptance testing by moving to an incremental delivery model where the system is built in functional modules that can be acceptance tested as they become available.

5.3.2 Doing Something Different

An extreme form of "too late" discovery is when we do not discover it at all and a problem is found by a user. If the problem is severe enough to have serious repercussions, the consequences can be disastrous. The high-profile losses or theft of customers' private information is just one example of something discovered "too late." These types of risk may require additional activities to reduce the likelihood of their occurrence. The solution often lies in doing additional kinds of testing to improve the likelihood that a certain class of defect, if it exists, is found in time. Many test authoring practices are focused on ways to define additional tests that improve the test coverage (from a risk coverage instead of a code coverage perspective).

5.4 Summary

A risk management model and a way to track risks and risk mitigation is important on all types of projects. Making risks explicit allows us to devise ways to mitigate the risks either by reducing the probability or the consequence. For risks related to misunderstanding of the requirements, earlier testing through incremental acceptance can reduce the consequence by giving us more time to rework any issues.

5.5 What's Next?

A significant risk on most projects is late discovery that the product is much farther from done than thought. The next chapter introduces a model for thinking about the degree to which the product is ”done”.

Chapter 6 Doneness Model

Determining whether the product is done is the goal of the acceptance process. Knowing how close to done we are, is a key function of project management discipline. This section introduces a model for determining how done a project is from both functional and para-functional perspectives.

A key part of making the acceptance decision revolves around deciding whether the product is "done". Coming to consensus on "done-ness" requires agreement on what is being assessed – a question of target granularity-- and what it is being assessed against – the done-ness criteria. In its most basic form, the acceptance process defines how we decide whether a particular release of functionality is done. In its more advanced forms we can determine done-ness at the level of individual features or work items. Each of these different targets has a set of done-ness criteria that must be agreed-upon. The following are some example questions to consider:

Is a software-intensive system (for example, a software product) ready for an alpha test with a friendly user community?

Is an individual feature or user story ready for acceptance testing by a business tester?

Is a software-intensive system ready for the design close milestone?

The definition of "done" is different for each of these examples.

6.1 Release Criteria – Doneness of Entire Systems

When determining the readiness of an entire software-intensive system for release to users, doneness is a very binary decision. Either we are done, or we are not. There are two main criteria for determining whether a system is done:

Is there enough high value, customer-defined features included to make the release worthwhile?

Is the quality of the feature implementations high enough for the features to be usable in their expected usage context?

The first criterion, known in this guide as minimum credible release (MCR)-- but also called minimum marketable product (MMP) or minimal marketable feature set (MMF)-- is typically determined while planning the release, Typically, the definition is revisited as the project is being executed and more is learned about the system context (such as business requirements) and the technical capabilities of the supplier (such as the delivery team).

When reviewing acceptance test results for each feature, it is fairly simple to determine the percentage of features that is done. This is the number of features that the customer has decided pass their critical acceptance tests divided by the total number of features for the release.

The second criterion, known in this guidance as minimum quality requirement (MQR), is what we constantly test against while building and assessing the software. To be able to say whether a feature has met the MQR, we need to have the acceptance tests defined for that feature; in this guidance, this is the per-feature definition of "What done looks like." Figure 1 illustrates these two criteria on two separate dates in the same project.

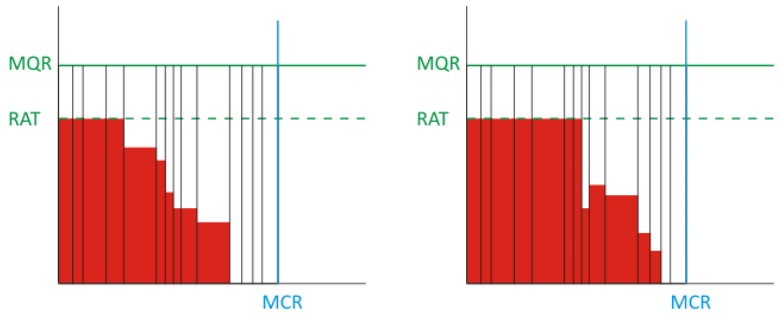


Figure 1

Doneness relative to MCR and MQR at two points in time

In Figure 1, the graph on the left shows the completeness of each feature at point *X* in time; the graph on the right shows the completeness two weeks later. Each column represents a feature with the width of each column being the estimated effort to build the feature. The line labeled RAT is when the feature is deemed ready for acceptance testing by virtue of having conducted the readiness assessment. It is the per-feature equivalent of the readiness decision that is made at the system level. The space between the RAT line and the line labeled MQR is when acceptance testing is done. The line labeled MCR is the demarcation between the features that must be present (left of the line) and those that are optional (right of the line; omitted in these diagrams) for this release.

We can see progress over time by comparing the left and right graphs in Figure 1. Features 1 through 5 (numbering from the left starting from 1) were already completed at the point in time represented by the left graph. Features 5 through 7 were already started at time x and completed (deemed ”done”) in the period ending at time x+2 weeks. Features 8 through 10 were already in progress at time x and were not completed at time x+2 weeks. Features 11 and 12 were started after the date depicted by the left graph and not finished at the time depicted by the right graph.

The product is deemed acceptable when all features pass all their acceptance tests. This is the upper-right corner of the graph where the lines labeled MQC and MCR intersect. When the rectangle to the lower-left of this point is entirely colored in, the product is accepted. To simplify the discussion, the para-functionality requirements are deliberately ignored, but each set of para-functional tests could be treated as another "feature bar" from the perspective of measuring doneness.

6.1.1 Good Enough Software

It is very easy to say “It all has to be there!” when asked what functionality is must be present to satisfy the MCR. Likewise, it is easy to reply “There can be no bugs!” when asked what MQR needs to be. But these answers are taking the easy way out. The product owner has to make a conscious decision about

the value of quality and features. The dogmatic answer may delay delivery of the product by months or even years. Meanwhile a competitor's product may be earning money and market share because that competitor had a more realistic goal for MCR and MQR. The choice of MCR and MQR should be a conscious business decision made in full knowledge of the cost of increased quality and functional in terms of actual (additional development, testing) and opportunity (delayed income and other products) costs.

For additional treatment of the topic of Good Enough Software, see [Bach] and [Yourdon]

6.2 Defining "What Done Looks Like"

For each chunk of functionality we have decided to deliver a "feature" that needs to be defined for the minimum quality requirement (MQR) in the form of a set of acceptance tests that must pass before the customer will accept the feature. The set of acceptance tests for a release is merely the aggregate of the acceptance tests for all the features ("functional tests") plus the acceptance tests for each of the para-functional requirements (the "para-functional tests") that are deemed mandatory.

6.2.1 Determining "Readiness"

"Readiness" is when the supplier believes the product is "done enough" to ask the product owner to consider accepting the product. This implies that the supplier has a reasonably accurate understanding of how the customer will conduct the acceptance testing. (In some cases, the supplier's "readiness tests" may be much more stringent than the acceptance tests the customer will run.) This understanding is known as the "acceptance criteria" and is usually captured in the form of acceptance tests. Ideally, the acceptance tests are provided to the supplier by the customer before the software is built to avoid playing "battleship" or "blind man's bluff" and the consequent rework when the supplier guesses wrong. It is also ideal for readiness tests and test results to be supplied to the customer by the supplier. This can assist in auditing the readiness testing, streamlining acceptance testing, and doing more informed 'black hat' exploratory acceptance testing.

6.2.2 Doneness of Individual Features

For an individual feature we can describe the degree to which it is done in several ways. One is how far along it is in the software development lifecycle. For example, "The design is done and we are half way finished coding it. We expect the feature to be ready for testing in two weeks." This form of progress reporting is often used on waterfall projects because the design and coding can take many weeks or even months. Unfortunately, it is not a very useful measurement once development is supposed to be complete. And on many projects it seems to stay that way for many weeks or months.

Once the coding and debugging is finished, it is more useful to define doneness in terms of which test cases are passing and which are not. In the waterfall approach, the percentage of test cases passing stays at zero for most of the life cycle and then rapidly rises to somewhere near 100%. On agile projects features are sometimes implemented one test case at a time thereby allowing the percentage of test

cases working to be used as a progress metric at the feature level instead of using the development phase.

The number of test cases for a particular feature may change over time as the feature is understood better. This may also be a sign of work creep (more effort to implement due to more complexity) or scope creep (additional functionality that wasn't originally intended). The additional tests may result from increased clarity caused by attempting to write tests or it may come as a result of feedback from other forms of testing including usability testing, exploratory testing or acceptance testing of an Alpha or Beta release. Regardless of the source of the additional tests, the percent done may take a step backwards if the number of tests increases more than the number of additional tests passing.

6.2.3 Doneness of Para-functional Attributes

Para-functional attributes of the system are a bit different from individual features in that they tend to apply across all features hence the name para-functional. Some para-functional attributes are binary in that we either satisfy the requirement or we don't. Examples include usability of systems; either we consider them usable enough or we have a list of known usability issues. We might, however, be able to break down the functionality of the system into functional areas and determine the usability of each area independently. For example, we may require a high degree of usability for customer self-service functions while administrative functions can have a much lower ease of use. Other para-functional requirements are quantifiable. For example, how many users can we support before response time exceeds our response time threshold? It could be 1 user, 10 users, or 1000 users.

We can chose whether we want to visualize progress on para-functional completeness as either additional features (support 10 users, support 100 users, support 1000 users) as illustrated in Figure Z or as a single feature (support many simultaneous users) with a minimum quality requirement (e.g. 1000 users) as illustrated in Figure Y. Our choice may be influenced by how we organize the work to develop the functionality. Agile projects often opt for the feature per threshold approach because support for additional users can be rolled out over a series of iterations. Waterfall projects may prefer to treat them as different levels of quality for a single feature as illustrated in Figure Y or as a different level of quality across all features as illustrated in Figure X. (See the discussion of varying MCR for Alpha/Beta releases in section *The Acceptance Process for Alpha & Beta Releases* for a counter argument.)

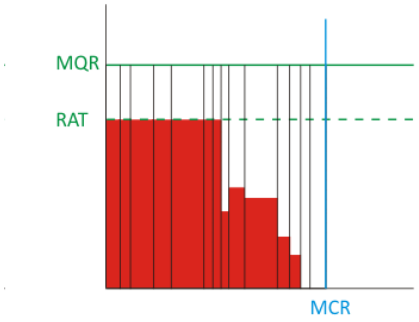


Figure Z – Incremental Para-functionality Features

In Figure Z the requirement for supporting multiple users is divided into two “features”. The feature to support at least 10 users is represented by column 6 and is ready for acceptance testing while the feature to support 100 users, represented by column 14, has not yet been started.

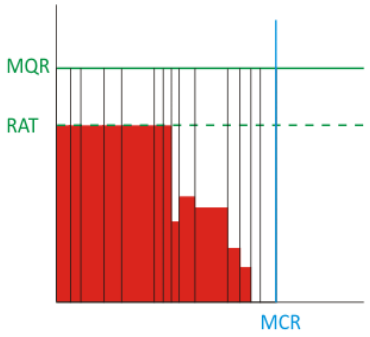


Figure Y – Single Para-functionality Feature

In Figure Y the requirement for supporting multiple users is treated as a single “feature” represented by column 10. It currently supports at least 10 users but is not ready for acceptance testing at the full requirement of 100 users.

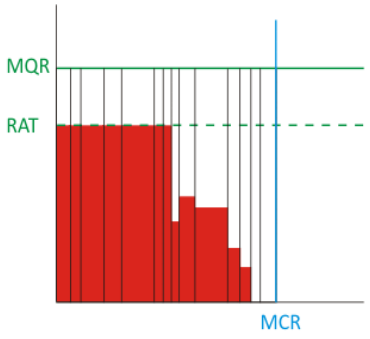


Figure X – Big Bang Para-functionality

In Figure x verification of the requirement for supporting multiple users is treated as an additional level of quality labeled PF across all features. It has not yet been started.

6.3 Communicating "Percent Doneness"

It can be said you are either "done" or "not done (yet)." But in practice, it is important to be able to clearly communicate "how close to done" you are. More specifically, it is important to be able to communicate "what remains to do before we can say we are 'done'". This is the amount of work left for each feature that has not yet passed all its acceptance tests summed over all the features that are part of the MCR. When looking at either diagram in Figure 1, we can ask "What percentage of the rectangle below/left of MQR/MCR is colored in?" This gives us a sense of how much work is remaining. A lot of white implies a lot of work; very little white means we are nearly done. Where the white is located tells us what kind of work is left to do. White across the top means we have lots of partially finished features;

white primarily on the right implies that entire features haven't been built yet but most features are either not started or fully finished. These patterns of progression depend primarily on the project management methodology we are using. The diagram in Figure 2 shows snapshots of completeness for three different project styles:

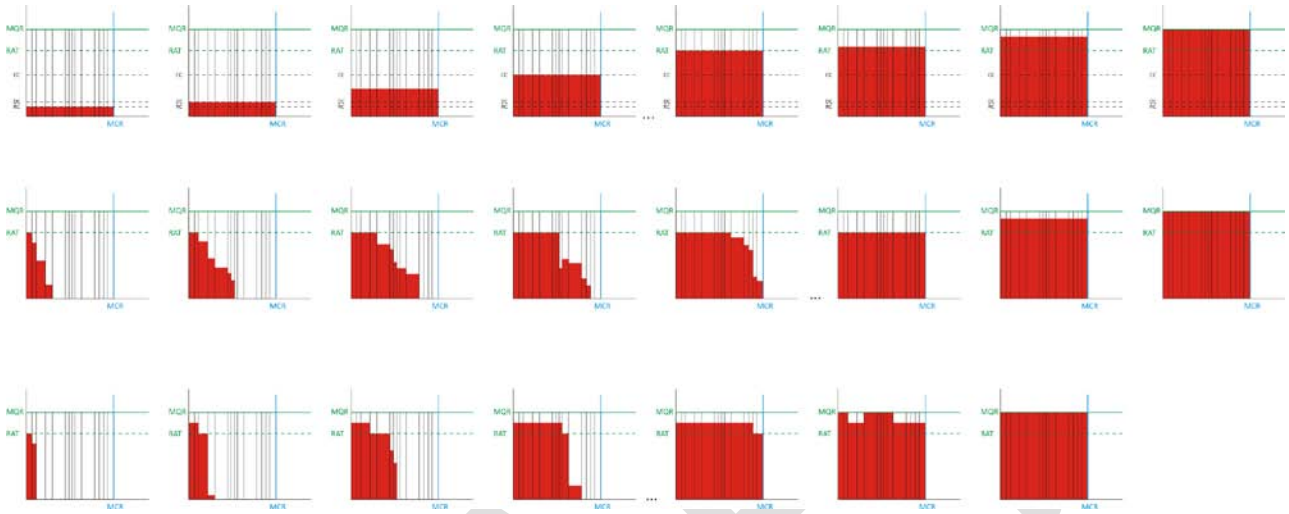


Figure 2

Completeness graphs

In Figure 2, the first row of graphs represents a classic waterfall or phased or document-driven style of project management with white across the top indicating that all features are at similar stages of completion. The bottom row represents a classic Extreme Programming project with white primarily on the right indicating that there are very few features partially done since most are either done and not started. The middle row represents a project using an incremental style of development with longer feature cycles than the Extreme Programming project. Notice the difference in how the coloring in of the graph advances toward the upper-right corner where the project would be considered fully done based on 100% of MCR and 100% of MQR. The next few sections delve more deeply into how we calculate and communicate done-ness on these different styles of projects.

6.3.1 Communicating Percent Done on Agile Projects

An agile project can simply divide the number of features that are accepted by the product owner as "done" by the total number of features schedule for the release. This provides the percent of features that are done. We can make it more accurate by weighting each feature by the estimated cost which is represented by the width of each feature column in Figures 2. The same graphic with column widths weighted by feature value is the agile equivalent of the earned value calculation.

The diagram in Figure 3 illustrates snapshots of how "done" each feature is at various points in time. Each mini graph represents a point in time. The height of the colored-in portion of each feature bar represents what degree that feature is done. A simple way to calculate this is dividing the number of acceptance tests passing by the total number of acceptance tests for that feature. Note that the number of tests could increase over time as the product owner's understanding of the feature improve.

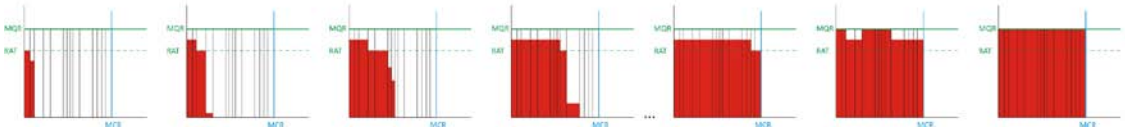


Figure 3
Doness graphs for an Extreme Programming / Agile project

Note how agile projects focus on both making the features small enough that they can be done in a cycle and not taking on too many features at the same time thereby minimizing the length of time that a particular feature is in development. (The goal is to complete each feature in the same iteration it was started in, or at worst case, the very next iteration.) This allows the customer to do incremental acceptance testing as each feature is delivered. Any bugs found can be scheduled for fixing at the appropriate time (which may be right away or in subsequent iterations.)

Figure 4 illustrates a "burn-down chart" based on plotting the number of features left to be "done" against time.

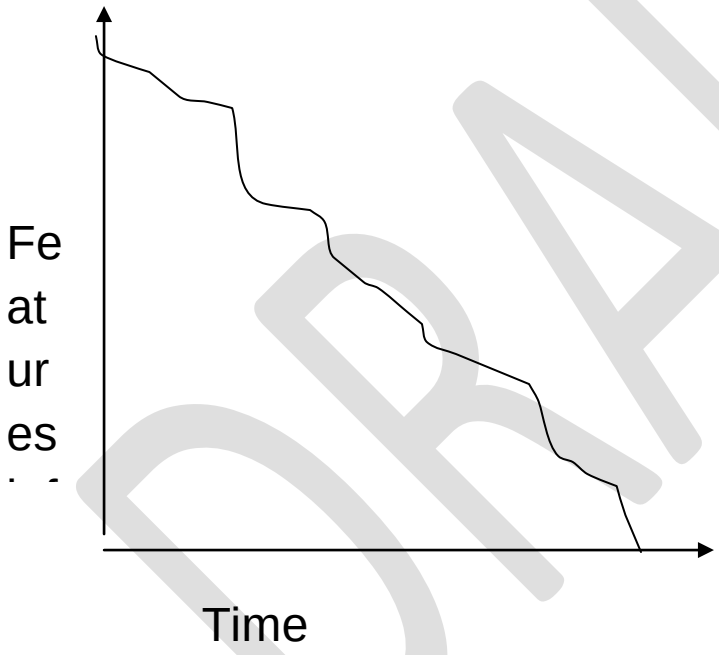


Figure 4
Burn-down chart for an agile project

Instead of having 100 percent of the features 50 percent done at the halfway point of the project, agile projects strive to have 50 percent of the features 100 percent done. This gives the customer options if specification and development of the functionality takes longer than expected, which is not uncommon. They can decide whether to adjust (reduce) the product scope to deliver on time or to adjust (delay) the delivery date to include all functionality. It also means that the work of readiness assessment and acceptance testing are spread out more or less evenly across the project. This is discussed in the section Staffing Impact.

Figure 5 illustrates graphs for a less agile project.

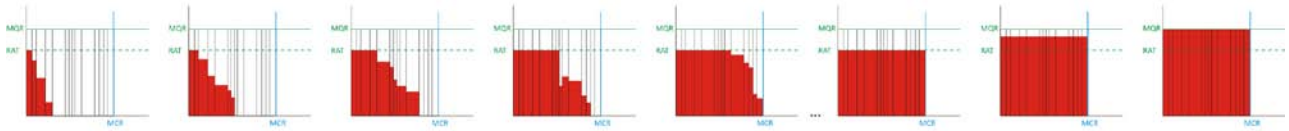


Figure 5

Graphs for a less agile project

On this example project, most features are taking several iterations to complete and acceptance testing only starts after all features are deemed ready. Because they are found very late in the project, there is less time to fix deficiencies found during acceptance testing (such as missed requirements) so they need to be fixed much more quickly or they tend to stay unfixed for this release, with workarounds, restrictions, and special cases..

6.3.2 Communicating Percent Done on Waterfall Projects

Waterfall (a.k.a. Tayloristic) projects have more of a challenge because the phases/milestones synchronize development in such a way as to ensure that all functionality is available for testing at roughly the same time. This prevents using "percent of functionality accepted" as a meaningful predictive measure of progress. Instead, Waterfall projects usually ask someone to declare what percentage each feature is done. For example, the developer may say they are 80 percent done coding and debugging (though this number is often stuck at 80 for many weeks in a row!). Considering the subjective nature of estimation techniques, waterfall projects often choose to use techniques such as "earned value" to determine a "degree of doneness" metric. Unfortunately, these techniques are prone to error and fudging, and they are both difficult and time-consuming to produce and maintain.

Figure 6 contains a typical sequence of doneness graphs for this approach.

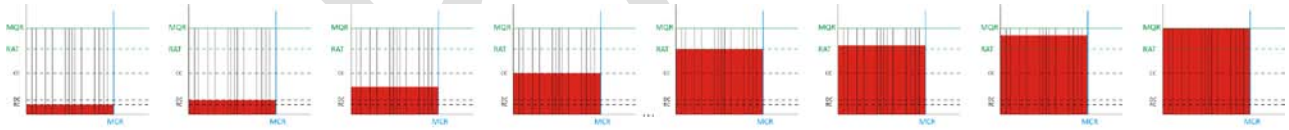


Figure 6

Graphs for a Waterfall project

In this waterfall version of the graphs, we can see how phased/waterfall development encourages us to work in parallel on many features because each feature is synchronized by gating mechanisms such as the milestones Requirements Frozen, Design Complete, and Coding Complete. This means that all the features are available for acceptance testing at roughly the same time and must be finished acceptance testing in a very short period of time because it is on the critical path of the project. This has implications for the staffing levels required for the readiness assessment and acceptance testing roles. When development is late, the period for readiness assessment and acceptance testing is further shortened and the resources further stressed. It also has implications on the impact of finding bugs during the testing because the fixes are on the critical path to delivery.

Figure 7 illustrates a "burn-down chart" based on plotting the number of features left to be "done" against time.

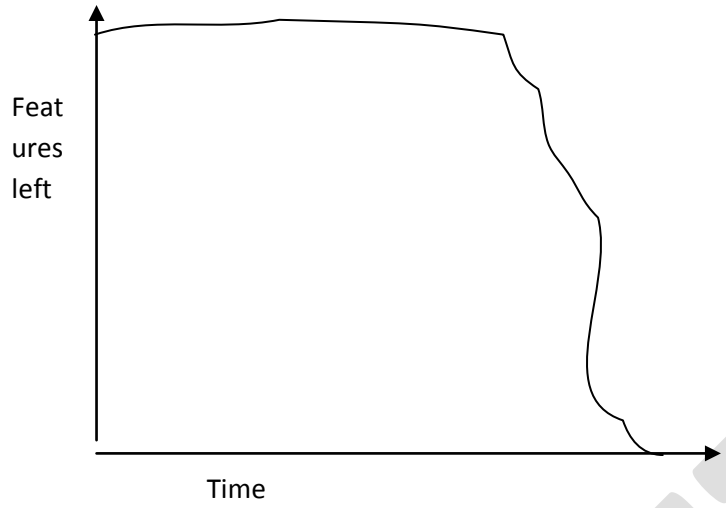


Figure 7
Burn-down chart for a Waterfall project

6.3.3 Staffing Impact of Process Selection

The test resources on a waterfall project are typically involved mostly at the back end of the project. Even when they are involved in fleshing out the requirements at the beginning of the project, their involvement while the software is being built tends to be minimal. The staffing profile of a hypothetical 10 week project is illustrated in Figure 7 – Waterfall Test Specialist Staffing Profile.

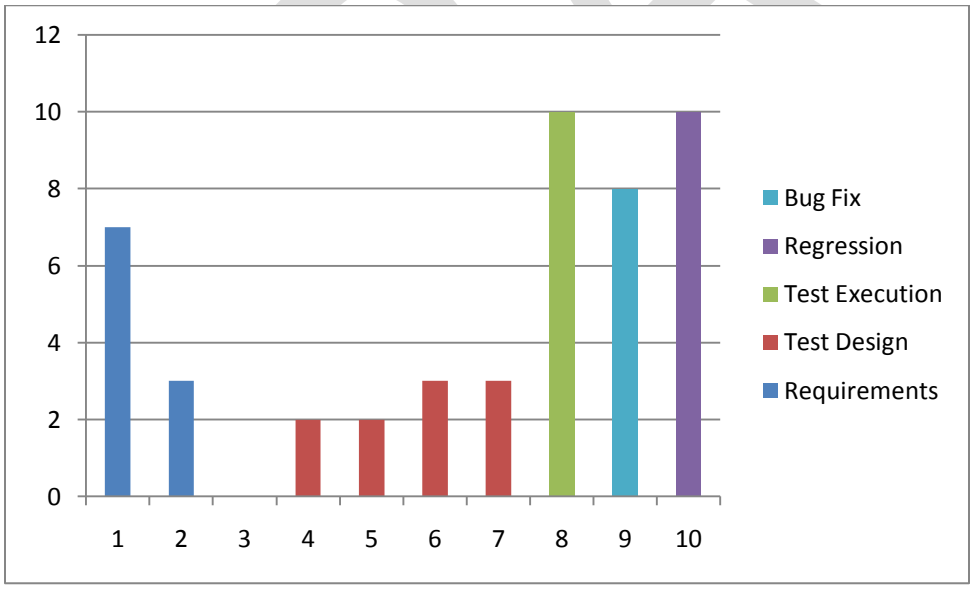


Figure 7 – Waterfall Test Specialist Staffing Profile

This project involves ten features and each feature requires one person week of requirements, one person week of test design and one person week of test execution followed by a person week of bug fixing by development and one person week of regression testing. Note how the bulk of the activity occurs at the back end of the project and requires ten test specialists for weeks 8 and 10 (week 9

involves primarily development resources.) This bursty nature of testing is usually accommodated by testers splitting their time across several projects. This makes it hard for them to keep up to date on what is being built and how the requirements are evolving. Development needs to be done in week 7 to leave time for two 1-week test cycles with a week of bug-fixing in between.

If the same ten features are built by an agile team using Incremental Acceptance Testing, the profile looks more like Figure 8 – Agile Test Specialist Staffing Profile.

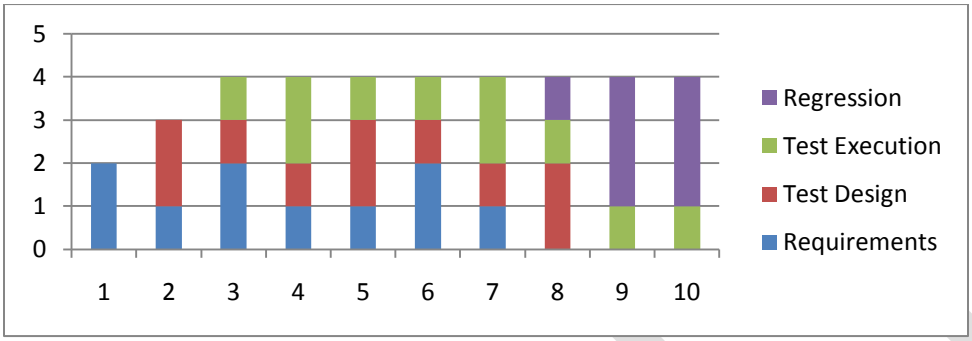


Figure 8 -Agile Test Specialist Staffing Profile.

On this project we start out by specifying two features in the first week, design the tests for those two features in the second week as well as specifying a third feature. In the third week we test the first feature and do test design for the third feature. Note how the staffing ramps up to four person weeks of work and stays there for the rest of the duration. Towards the end as new feature testing ramps down the slack is taken up by regression testing of the early features. Because the testers are part of the development team they are able to keep abreast of how the requirements and product design are evolving. Since the same people will be designing and executing the tests the need to write detailed test scripts are reduced and more effort can be spent on exploratory testing and test automation. Since the testing is done soon after the code is developed the developers can fix the bugs immediately so there is no separate "testing and bug fix phase" at the end therefore development can continue closer to the delivery milestone.

Sidebar: What it takes to do Incremental Acceptance

Agile projects routinely do Incremental Acceptance Testing; the onsite customer (the product owner) is shown the working feature as soon as it is deemed ready by the developers, often just a few days into a development iteration. Yet waterfall projects encounter many obstacles when they try to overlap testing with development, especially when this is done to recover the schedule when development is running late. Why the difference?

There are a number of factors at play here, each of which may contribute to the situation. The problems include difficulty communicating what is testable and what is not; this results in frustration amongst testers that the testing results in many bugs to which the developer responds "that feature isn't done yet; you should not be testing it."

First, agile projects usually have cross-functional teams. That is, the customer or product owner, any requirements analysts or usability people, the developers, and the testers all sit together and talk with each other daily. This means that there is much better awareness amongst testers, whether they be readiness assessors or acceptance testers, of what can be tested and what should be left for later.

Second, agile development teams are very disciplined in how they manage the code base. Most successful agile teams are doing test-driven development and have a continuous integration server which rebuilds the entire code base after every check in. Part of the build process is to run all the automated unit tests and any test failures are treated as a "stop the line" event where the top priority becomes getting the build working again. This ensures a stable code base at all times; the team simply won't tolerate members dumping half-finished code into the build.

Another form of discipline is the decomposition of large features into smaller features and even smaller tasks which can be completed in under a day. This allows working, tested code to be checked in several times a day avoiding the overhead of branching and merging.

Agile teams are also very disciplined about keeping the number of features in progress as low as possible. This practice, borrowed from the world of lean manufacturing and lean product design, minimizes the amount of multi-tasking people need to do thereby improving focus, flow and productivity. A very useful side effect is that the length of time from deciding on what is required to when the functionality is ready to be acceptance tested is kept to a minimum; often just a few days. This reduces the likelihood of the requirements changing while development is in progress or the customer forgetting what it was they meant by that requirement.

6.4 Summary

It may seem like a simple concept to determine if a feature or product is done, but in practice it can prove to be somewhat slippery. We must know how to determine if we are done, what criteria to use to establish whether we are done and how to test those criteria. It involves determining if the proper features or functionality is present, if it works as expected or required and does it meet the expectations of the customer. Having a process in place will increase our chance of successfully delivering a product we are happy with.

6.5 What's Next?

This finishes our introduction to the models we can use as tools for thinking about acceptance. The next few chapters describe the activities involved in planning and executing the acceptance process. We start with a more detailed look at the practices involved in planning.

6.6 References

[Bach] Bach, J. The Challenge of “Good Enough Software”.
http://www.satisfice.com/articles/gooden2.pdf

[Yourdon] Yourdon, E. "When Good Enough Software Is Best," *IEEE Software*, vol. 12, no. 3, pp. 79-81, May 1995.

Part II - Perspectives on Acceptance

The process leading up to the acceptance decision involves a number of different players and each has their own perspective on the process. The chapters in this part describe acceptance from a number of these perspectives including several parties that may play the role of acceptance decision maker (Business Lead, Product Manager, Enterprise Architect, Legal Department and Test Manager) and other parties that may play the role of readiness decision maker (User Experience Specialist ,Development Manager, Solution Architect and Test Manager.) Each of the perspectives that follow focus on aspects of acceptance that are particularly salient to the player. It is worth reading all the perspectives even if one seems to fit the reader like a glove.

Note to editors: The text in Part 2 deliberately speaks directly to the person in each of the roles. That is, it addresses the person as though we were in conversation with them. It uses the "you" pronoun on purpose as opposed to the Part 1 chapters which typically used "we must …" or "we do …".

Chapter 7 The Business Lead's Perspective

An organization may commission the construction or customization of software for its own use. The construction would be done by another organization which may be an internal department (such as the Information Technology department) or another company (an outsourced development company.) The Business Lead (also known as the Project Sponsor, Business Sponsor, Product Owner, Business Owner, or Business Partner) is the person who brings to the project team the vision of what the software is intended to do.

See the Business Lead persona in the company stereotype called Software Built by an IT Department for the Business in Appendix X – Reader Personas for an example.

The decision to commission *custom* (or *bespoke* as it is called in some locales) software is a business decision made based on the expected business benefits of the investment - the return on investment (ROI). The benefits may be increased income through additional sales or services, cost reduction through fewer staff, more efficient work processes or reduced errors – essentially, these benefits are intended to contribute to the economic bottom-line of the organization. So now that you've made the decision to commission construction of the software, now what? Before you can realize the benefits you will be asked to make the decision to accept the software as built or provide a list of deficiencies that must be addressed before you will accept it. Acceptance is a significant stage of any contractual process (See Legal Perspectives on Acceptance). How you approach this acceptance decision can have a big influence on how painful or painless the process will be. It will also impact your post-acceptance activities (including actual usage, operational support, maintenance etc.)

The wrong approach can be summarized with the famous quote "I cannot describe art but I'll recognize it when I see it." Taking this approach to the acceptance decision is sure to result in late delivery, cost overruns, delivery of software that does not bring value to your org and a strained relationship with your supplier because the supplier will have a poor chance of meeting your needs if they don't know what they are. When they deliver the software you wanted, you'll be sure to find all sorts of issues with how it works. You'll report these "bugs" to the vendor and expect most of them to be fixed before you accept the software and put it into production. That will take additional time and resources thereby increasing costs and deferring the benefits. In the worst case, the vendor may be unable to fix all the defects to your satisfaction and you'll either have to accept software you are not happy with or write off the system entirely.

A good customer has a clear vision of "what done looks like" and shares that vision with the supplier as early as possible. A good customer also recognizes that communication is inherently flawed and that you need to regularly verify that what you intended to communicate has been understood. Therefore, a good customer is prepared to try out and provide feedback on the software under development at frequent intervals; a good customer doesn't simply "throw a specification over the wall" and say don't

bother me until you are ready for acceptance testing. They take an active interest in the emerging software and encourage the supplier to provide frequent opportunities for feedback.

To quote Ludwig Wittgenstein, one of the leading philosophers of the 20^{0th} century, "there is a gulf between an order and its execution. It has to be filled by the act of understanding." [Wittgenstein] To be successful, do not underestimate this "act of understanding". Successful projects typically have a clear agreement between the business lead and the supplier on what constitutes a successful outcome for the project. This project charter provides a high-level view of the business lead's expectations of the project and how each party will behave throughout the project. It also lays out any key constraints regarding schedule, cost and any other relevant factors. Don't focus the charter (or contract, Statement of Work, etc.) on what remedies you will have if the supplier fails to deliver; by then the project has already failed! Instead, focus on the behaviors that will ensure project success. All parties should treat the project as a miniature joint venture and strive for a win-win outcome. If any of the parties is trying to win at the other's expense, all parties will lose. Dysfunctional relationships lead to dysfunctional products. See the sidebar "Target cost contracting" for description of an alternative to traditional fixed price or time & materials contracts.

[Side Bar] Target Cost Contracting : from adversarial contracts towards partnering contracts

Lean Software Development visionaries Tom and Mary Poppendieck advocate the "partnering" or "relational" approach to contracting, in which parties agree to work in a collaborative way to carry out the project [Poppendieck & Poppendieck, Ch.9]. Target Cost Contracts are a type of incentives-contracts that can be used to support sharing of both financial risks and rewards. This helps to align the best interest of each party with the best interests of the joint venture. The Poppendiecks encourage forging synergistic relationship across company boundaries. They envision that those who do will see at least the 25 percent to 30 percent cost reduction of the cost of doing business predicted by Peter Drucker in [Drucker].

Target Cost Contracts are an approach to identifying and constraining uncertainty. It's done in the following way. A target cost for the project is agreed at the beginning and both parties are proactively working on achieving the target cost. Upon project's completion, the actual cost is measured against the originally set target cost. If the actual cost is great than the target cost, the supplier and the customer each bear an agreed element of that cost overrun. If the actual cost is lower than the target cost, the supplier shares the savings with the customer. The share of the cost overrun/underrun may be a constant proportion or may vary depending on the size of the delta.

Note: The analysis of Perry & Barnes reveals a strong case for setting the supplier's share of cost overrun or underrun at a value that is not less than 50% [Perry et al]).

- The key characteristics of the Target Cost Contracts are as follows:Target cost includes all changes

- Target is the joint responsibility of both parties
- Target cost is clearly communicated to workers
- Negotiations occur if target cost is exceeded with neither party benefiting.
- This entices workers at all level to work collaboratively, compromise, and meet the target.

For additional case study of using Target-Cost Contracts in software development, see this experience report [Eckfeldt et al].

7.1 Test Planning

Accepting software sounds like a discrete event but there is much to be done leading up to it. As a customer you need to be prepared to participate in the development of your software. Anyone who has had a house built specifically for them knows that there are a lot of decisions to be made and inspections to be done throughout the entire construction process. Software should not be any different. Expect to get as much out of your software as you are prepared to put in. The absentee customer gets the results they deserve; an unsuitable system designed for someone else.

7.1.1 Communicating Expectations

To avoid this fate you certainly must have a plan for making the acceptance decision. But you must also plan for regular communication with the supplier about the criteria for that decision. One of the major acceptance criteria will be successful implementation of the requirements you have specified. These requirements may take many different forms including:

Functional specifications written in plain language (e.g. The system shall ….)

Use case models and documents

Protocol specifications for interfaces to other systems

User interface mockups, prototypes, cognitive walkthroughs, or existing systems to emulate.

Descriptions of business rules and algorithms

These requirements documents form an important basis both for the vendor building the system and your acceptance testing plans. But recognize that these documents and models will never answer all the questions the vendor may have. Nor will they exhaustively describe all your expectations. In fact, you may not even be aware of some of your expectations until they fail to be satisfied by the software. No amount of up-front requirements planning will flush out all your expectations therefore it is crucial to interact with the software early and often to give yourself time to discover these unrealized expectations early enough to afford the supplier time to implement them.

Expect the supplier to discover many ambiguities and unforeseen circumstances in whatever requirements and expectations you have described to them. Plan to be available to the supplier on a regular basis, the more frequent the better. If you cannot participate personally, assign someone to act as your customer proxy and ensure that they have full understanding of what you want to achieve and a clear mandate on what they can decide themselves and what needs your approval. On agile projects, expect to provide at least one person full time for the entire duration of the project to clarify requirements and do continuous incremental acceptance testing.

7.1.2 Planning When to Test

A key part of the agreement with the supplier should be the frequency with which you are supplied working software on which you can perform acceptance testing. In the "waterfall" model, you will likely need to wait until near the end of the project before you receive any software. This may be a single "final" release that the vendor believes is complete, high quality code, or there may be a series of releases with names such as Alpha, Beta or Release Candidate that are provided early enough to allow revision of how the software works based on the results of acceptance testing of each release. In most cases, the preliminary releases are either of lower quality or contain less functionality than the final release. Make your incremental acceptance testing plans accordingly; don't expect everything to be available in the early releases. What is to be available in each release should be the subject of discussions with the supplier.

In the "agile" model, you should expect to receive working, potentially deployable software on a regular schedule throughout the entire project timeline. You'll be asked to decide what functionality the supplier should build in each iteration (A.K.A. sprint) and you'll need to be prepared to provide the detailed acceptance criteria for each feature or user story to the supplier early in the iteration. This process is known as *Acceptance Test Driven Development*. When the software feature is finished, you'll be asked to try out the newly built functionality and decide whether it works to your satisfaction or requires changes before you will accept it. In this *incremental acceptance testing* model you will need to test additional functionality each iteration. You may also need to dedicate one or more iterations at the end of the project to the final end-to-end acceptance testing and release activities.

Regardless of the process model you should expect the supplier to do adequate testing as part of their readiness assessment. The expectation should be clearly understood by all parties and you should plan to provide the acceptance criteria and detailed acceptance tests to the supplier so that they can run the tests during their readiness assessment activities.

You will also need to have plans in place for how you will assess the software when it becomes available. At a minimum, you will need to know how many people and how much time will be required to do the assessment. The details of what tests they would run may also be included or this may be determined nearer to the actual testing.

7.1.3 Regression Testing

Assuming you will receive a new version of the software more than once, and this is a very safe assumption regardless of whether you use a waterfall or agile process model, you'll need to retest not

only the newly introduced or changed functionality but also all the previously built functionality. This can turn out to be a lot of effort. One strategy is to choose the tests to be run based on what was changed but this increases the risk of missing newly introduced bugs. Another strategy is to have a suite of automated tests (see Automated Testing in Ch.16: Planning for Acceptance) that can be run quickly after every change to the software. This can greatly reduce the effort involved in this *regression testing*; make sure to ask your supplier to include automated tests as part of their costing to ensure you don't have an ever-increasing workload as more and more functionality is delivered. The automated tests are really for the supplier's benefit as it helps them ensure that they are not introducing defects into the existing software. Also, you may want to request the regression tests suites to be shipped as part of the project deliverables. Tests are assets and therefore you should plan on keeping them current for as long as you plan to use the product. Having the acceptance tests in place may reduce the cost of your future maintenance/extension projects (for which you may choose to use a different supplier). But the acceptance tests may themselves require *test maintenance or refactoring* as the behavior of the system under test evolves.

7.1.4 Test Resourcing and Outsourcing

The testing you plan to do may be influenced by your knowledge of what testing the supplier will have done prior to giving you access to the software and by the skills and experience of the people you have available to test. When you don't have the necessary expertise and you cannot or choose not to bring it in house you can elect to use a third-party organization to do your acceptance testing on your behalf. Be aware, however, that very few *test outsourcers* have perfected mind-reading! You will need to provide them with detailed direction regarding the nature and objectives of the testing you want them to do and the expected behavior (i.e. requirements) of the system to be tested. You'll also want to be very clear about the acceptance criteria and whether they are making the acceptance decision on your behalf or providing you with data to support your acceptance decision. Test outsourcing is common for specialized type of testing (e.g. security testing). Test outsourcing may also be mandated by a certification body/regulatory authority that requires an independent verification/audit to be performed. For example, all tax preparation software packages may need to be tested for compliance to the tax authorities' standards for tax return information.

7.1.5 Test Estimation

Estimating the amount of time and effort to allocate to acceptance testing is difficult without a good understanding of the nature of the functionality involved and the thoroughness of the readiness assessment that will be done by the supplier organization.

A common strategy is to time-box the final *acceptance test phase* and guess the number of cycles of acceptance testing plus bug fixing that will be required. This requires past experience to come up with reasonable guesses. If in doubt, guess high and allow for several cycles of testing to give the vendor time to fix bugs. Warning: There is always a lot of pressure to guess low to allow earlier delivery of the software and the lower cost that this implies. But guessing too low will typically result in having to choose between accepting the software on schedule with many known deficiencies or delaying the

release to allow additional test&fix cycles to be performed until all important deficiencies are fixed and retested. The testing phase is often squeezed by late delivery of the software by the supplier and the testers are often asked to make up the delay by reducing the elapsed time for testing.

The alternative strategy involves *incremental acceptance testing*; acceptance testing that is done frequently throughout the project. This improves the predictability of the delivery date in two ways. First, it finds bugs earlier allowing them to be fixed long before the final acceptance test cycle. Second, it provides you the opportunity to learn how much testing is really required and how much time and effort it will take to do it. This learning happens early enough in the project that you can adjust your plans for later iterations and the final test cycle so that the project can be delivered on time. Adjustments could include increasing or decreasing the planned number of test cycles, add staff or skills, increasing or decreasing the amount of test automation, contracting a third party for specialized testing and so on.

7.2 What Do You Need to Test?

Ensure that you have a clear understanding of what will be tested by the supplier and what you are expected to test. Typically, the supplier is expected to do thorough testing of all capabilities and *quality characteristics* (such as performance, reliability, usability etc.) before handing off system for AT. Of course, this is based on the suppliers understanding of your expectations which may vary from yours if the communication is less than perfect which is almost always the case. Your responsibility during acceptance testing is ensuring the system is suitable for your own purpose.

The test cases you run during acceptance testing should cover everything you care about. This can include business functionality, operations functionality and para-functional characteristics of the system. Each of these areas may require different expertise to test and possibly different stakeholder signoffs. Deciding who will sign-off based on what testing, done by whom, is an important part of acceptance test planning. The business lead may need to negotiate with other parties to determine the acceptance decision process and the testing that provides the information.

7.2.1 Functional Testing

Functional testing should include *business workflow tests* that verify that all reasonable business scenarios or workflows function properly. More detailed *use case tests* can be utilized to verify that each business transaction or use case is implemented correctly. The happy path or main success scenario of each use case is not where bugs will typically be found; they tend to lurk in the dark corners of the less commonly exercised scenarios. *Business rule tests* can focus on verifying that the business rules and algorithms are implemented correctly. Applications with rich user interfaces typically require extensive testing of the user interface itself. This testing can either be bundled with the use case test or you can have separate test sessions dedicated to *exploratory testing* the behavior of the user interface.

The degree to which you need to test should be based on the level of confidence you have in your supplier. This can be based on past experience on previous projects or on what they have demonstrated on this project. A supplier that has provided full visibility of their internal processes and especially their readiness assessment activities may inspire much more confidence in the quality of the product. This

could change the focus of your acceptance testing activities from looking for obvious mis-implementations of the business logic or rules to higher value-adding activities like usability testing and business workflow testing. In particular, the presence of an extensive suite of business-readable automated workflow, use case and business rule acceptance tests could change the emphasis from conducting extensive manual testing of this functionality to spot checking. See Automated Execution of Functional Tests for how to automate these tests.

7.2.2 Para-Functional Testing

- Para-functional testing (aka Non-functional testing) is often done by the supplier during the readiness assessment or by a separate test organization. It should include verifying any para-functional characteristics of the system that are deemed critical for acceptance as per System Model , such as:
- Performance (capacity and response time) and stress (behavior when overloaded) tests,
- Security testing (enforcement of privacy policies, resistance to attack by hackers, etc.)
- Scalability (the ability to handle additional users or transaction loads in the future)
- Usability (the efficiency, effectiveness and satisfaction with which users can employ the software system in order to achieve their particular goals)

and others.

Ensure that you have a common understanding with the supplier as to what para-functional testing they will do and how much information you expect to receive as input into the acceptance decision.

The people charged with operating and sustaining the system (herein called the operations department) will have very specific requirements that need to be addressed. You'll need to be aware of these as you work with the supplier exploring and defining the requirements for the system. In a larger organization you may have a counterpart in the operations department who can provide the requirements and act as the acceptance decision maker from the operations perspective. See The Operations Manager's Perspective later in this chapter.

At a minimum you'll need to be aware of the time and resources complying with these requirements will entail. In some cases you'll need to give up new functionality in order to ensure that the operational requirements are met. Some examples of operational requirements might include:

- The system needs to integrate with the specific systems monitoring software in use at your company.
- Data needs to be migrated from a previous version of the system or from a system this one replaces.
- The system needs to have upgrade, startup and shutdown scripts that can be used during automated server maintenance windows.

- The system needs to be built using an approved technology stack. The operations department may not have the skills to support some technologies. E.g. A .Net shop may not have the skills to support an application built in Java or PHP. Some technologies may be incompatible with technologies used at your company.
- The operations group may have specific windows during which they cannot install or test new versions of software. This may affect the release schedule you have devised and it may make on-time delivery of software by the supplier even more critical as small schedule slips could result in long in-service delays.
    - There could be requirements for performing such service operations without degrading the current system
- Reliability requirements dictate that the system meets the agreed upon service level agreements (SLAs) such as uninterrupted uptime, transaction latency, .
- The system must provide self-monitoring, error detection, and system logging capabilities.
- The system must provide for robust recoverability (including the ability to verify multiple failovers and recoveries), disaster recovery, data backup & recovery, and all other aspects of system recovery.
- A full range of documentation must be provided including support documentation, both pre and post roll-out, training manuals, and other reference documentation.
- The system must be migrated to the new version of the platform.
- Multiple versions of the system must run side-by-side.

7.2.3 Other Parties Involved in Acceptance Testing

If there are other parties involved in the acceptance decision ensure that you have a common understanding about what they will test and how that feeds into the acceptance decision. Common parties include the operations department who would have a say in whether the system can be installed and managed, usability practitioners who might need to ensure that the system complies with any usability (including accessibility) standards, and corporate security departments who need to ensure the system is secure before it can be accepted.

7.3 Where Will You Test?

A key part of test planning is deciding what environments will be used for what kinds of activities and how the software is moved between environments.

7.3.1 Test Environments

Expect to have many activities happening concurrently as you do acceptance testing. The supplier will likely be fixing the bugs that you report while you continue to look for further issues. This requires a separate environment. When the supplier has fixed enough bugs to warrant delivering a new version of the software they will need to do readiness assessment. This may require yet another environment if bug fixing is to continue in parallel. If a test organization needs to test the software before passing it on to you for acceptance testing they will need to have their own test environment as well.

7.3.2 Software Delivery Process

Given that different groups will be working in separate environments, how does a new version of the software get installed in the environment? For example, you've finished acceptance testing on release candidate 1 (RC1) of the software, reported a number of bugs, and the supplier has fixed enough bugs to propose delivering a new version, RC2 to your test environment. What is the process for replacing the RC1 software in the acceptance testing environment with RC2? Does the supplier push the new version with or without any advance notice? Does the supplier notify you of the availability of RC2 and wait for you to ask for it to be installed in the acceptance testing environment when you are ready to receive it? And how long will the acceptance testing environment be unavailable while the upgrade is being installed?

7.4 Test Execution

When the software is delivered for acceptance testing, acceptance test execution must begin in earnest. It is important to know where you stand at all times with respect to the progress of testing and your current understanding of the suitability of the software.

7.4.1 Progress Monitoring

As acceptance testing is invariably time-boxed by either contracts or customer expectations, it is important to ensure that the testing activities are progressing in a time fashion. The nature of the test plan determines the nature of progress monitoring. Agile projects have acceptance testing spread out over the duration of the project and the results of testing are highly visible in the form of feedback to the supplier team. There is plenty of time for the acceptance testers to learn from their mistakes and misassumptions in time for the next round of acceptance testing. Traditional test-last (Waterfall) projects, on the other hand, have all testing highly concentrated at the end of the project. It is critical that these test activities be very closely monitored to ensure that any deviations from the test plan are quickly identified and rectified. There isn't much time for learning better approaches to testing so everyone must be highly trained before the acceptance test period.

7.4.2 Record Keeping

The nature of the product being built and the context in which it will be used has a large bearing on the nature and degree of record keeping. It may be sufficient to simply run tests and track any bugs found in

an informal manner. In other environments, records need to be kept about what tests were run, against which version of the software, when, by whom and the actual results that were observed.

7.4.3 Defect Tracking

At a minimum, any defects that were found during acceptance testing need to be recorded and triaged. Each bug should include which version of the software it was found in, whether it will prevent acceptance of the software (severity level) and who found it. Bugs that must be fixed before the software will be accepted must be clearly communicated to the supplier. Less severe bugs should also be recorded but should also be clearly labeled as not gating acceptance.

7.4.4 Regression Testing

Each time the software is changed, whether to add new functionality or to fix bugs found during acceptance testing, new bugs may have been introduced. Therefore, any tests performed before the software was changed *may* need to be executed again on the new version of the software. This can be very time consuming if the regression tests are not automated.

7.4.5 Maintaining Multiple Versions of the Software

There will be times that you may choose to maintain several versions of the software at the same time. A common situation is after release, during the warranty period, if you are also developing a subsequent release at the same time. Another situation is when you choose to support multiple version of software in production and need to apply any bug fixes to all supported versions. Be forewarned that there is an extra cost to maintaining versions in parallel as any bug fixes done in the support/warranty stream will need to be propagated into the development stream at some point and retested there.

Chapter 8 Product Manager's Perspective

A product manager in a product company plays a very similar role to that of the business lead on an IT project with a few exceptions. While the business lead is typically engaged only for the duration of the project to deliver the new functionality, a product manager is typically responsible for the profitability of the product over its entire lifetime. They are more inclined to think in terms of multiple releases of the software/product. Their strategy may also be focused on a product line, not individual products, in which case the integration and interoperability among these products is even more important. Companies that build products are also more likely to have dedicated testing resources in a separate test department. Many of the factors discussed in The Business Lead's Perspective also apply to the product manager. This section will focus on the differences.

See the Product Manager persona in the company stereotype called "A Product Company" in Appendix X – Reader Personas for an example.

As a Product Manager, you should feel total ownership of the success of the product in the marketplace. You survey the market for what it needs and define the product to be built. You decide how much revenue the product is likely to generate and how much money it is worth investing to build the product. It is your job to ensure that the supplier team understands what you want built and the relative priorities of the pieces of functionality.

8.1 Defining Done

A key part of the relationship between the product manager and the supplier organization is having a commonly understood definition of what constitutes "done". This comes in two dimensions: functionality and quality.

8.1.1 Defining MCR

MCR stands for Minimum Credible Release (also known as Minimum Marketable Product (MMP) or Minimum Marketable Functionality (MMF)); it is the smallest amount of functionality that you could deliver to your customers without losing credibility. Anything over and above the MCR adds value but should not delay the release if it cannot be completed by the delivery deadline. It takes self-discipline and a good understanding of the market to define the MCR; it is so much easier to just throw in everything any customer has every asked for and call it a release but this is almost sure to result in spending too much and taking too long to deliver the product.

You may need help to translate the high level requirements into a detailed product definition. Business analysis expertise may be needed to understand the business processes used by the potential users. Product design expertise (e.g. interaction designers and graphic designers) may be needed to help design the functionality. Product development expertise may be needed to understand what is technically possible and consistent with how the product already works. You may want to include people with these skills as part of the product management team or you can enlist the help of other organizations. Whichever way you choose to deal with the issue you'll need to ensure that everyone understands your vision for the product and how it satisfies specific market needs.

8.1.2 Defining MQR

You don't like receiving unsatisfactory software and the supplier team hates being told that their carefully crafted software is not very good. Establishing the minimum quality requirement (MQR) is an important step in building a trusting, productive relationship. A jointly crafted Done-Done Checklist posted in the supplier organization's office is a great way for the supplier team to keep the focus on quality front and center. A release-level checklist as shown in Figure 1 is a good way to set the overall expectations of what the supplier should be providing before acceptance testing can start

Sample Done-Done Checklist for Release 1.0

- All MCR features are included in the RC build.
- A security review has been conducted and a sign off obtained.
- Test team is confident that none of the included features has a significant risk of causing problems in the production environment, i.e. MQR met, including:
  - config testing, side-by-side
  - performance
  - localization
  - globalization
  - user experience.
- Business compliance achieved.
- There are clear, concise deployment and rollback instructions for the operations team.
- There are clear trouble-shooting scripts and knowledge base articles for use by the help desk representatives.
- All included features have been demo'ed to and accepted by the customer.
-

Figure 1

Sample Done-Done Checklist for a Release

The quality criteria described in a release –level "done-done checklist" as shown in Figure 1 may require several readiness assessment cycles before the criteria are satisfied. To reduce the number of readiness assessment test&fix cycles we need to influence the level of quality delivered by individual developers to the readiness assessors. This can be affected by a feature-level "done-done checklist" as shown in Figure 2.

- The acceptance criteria are specified and agreed upon
- The team has a test/set of tests (preferably automated) that prove the acceptance criteria are met

- The code to make the acceptance tests pass is written
- The unit tests and code are checked in
- The CI server can successfully build the code base
- The acceptance tests pass on the bits the CI server creates
- No other acceptance tests or unit tests are broken
- User documentation is updated
- User documentation is reviewed
- The feature is demoed to the customer proxy
- The customer proxy signs off on the story

Figure 2

Sample Done-Done Checklist for individual features

Done-done checklists such as these are commonly used on agile projects.

8.1.3 Managing Your Own Expectations

The most effective way to ensure that poor quality software gets delivered is to ask the supplier to over commit. Software requirements are notoriously hard to pin down and just as hard to estimate. Forcing a supplier to deliver to an unrealistic schedule is sure to backfire as you will likely get a late delivery of poor quality software that you then need to invest extra time into bringing the quality up to barely sufficient level. This is not a recipe for success!

It is far better to define the level of quality required and manage the scope of functionality to ensure it can all be finished in the time and money allotted. Time-boxed incremental development is a proven technique for achieving on-time delivery of quality software. It is to your advantage to work closely with the supplier organization to select the functionality to be implemented during each iteration. You'll have much better visibility of progress towards the release and much earlier warning if the full slate of functionality cannot be completed by whatever delivery date you have chosen. This gives you time to prune the functionality to fit into the time available rather than trying to cram in too much functionality and thereby sacrificing quality.

8.2 Who Makes the Acceptance Decision?

As product owner, you need to be responsible for making the business decision about whether to ship the product as-is or to delay delivery to allow more functionality to be added or to improve the quality by fixing known bugs. You will likely rely on data from other parties to make this decision but delegating this decision to one of those parties is at your peril because they are unlikely to understand the business tradeoffs nearly as well as you do.

It is reasonable for you to expect that the parties providing information about the quality of the release candidate do so in terms that you understand. The impact of every missing feature or known bug should be described in business terms, not technical jargon. Ideally, this should be easily translated into monetary impact (lost or delayed revenue) and probability (likelihood of occurrence).

8.3 Operational Requirements

Most software products have operational requirements. These are the requirements of whoever will have to provision the runtime environment, install the software, monitor its performance, install patches and upgrades, start up and shut down the software during server maintenance windows, and so on. Even the simplest shrink-wrap product has operational requirements (such as automatic updating) and complex server products often have very extensive operational requirements. Ensure that you have the appropriate people engaged during the product definition and product acceptance process to ensure that these requirements are not missed. Make sure you have the right people involved in the decision making process to ensure that the requirements are satisfied. Operational requirements may require specialized testing and test tools.

Some examples of operational requirements for Software-as-a-Service are similar to those of an IT shop:

- The system needs to integrate with the specific systems monitoring software in use at your company.
- The system needs to have upgrade, startup and shutdown scripts that can be used during automated server maintenance windows.
- The system needs to be built using an approved technology stack. The operations department may not have the skills to support some technologies. E.g. a .Net shop may not have the skills to support an application built in Java or PHP. Some technologies may be incompatible with technologies used at your company.
- The operations group may have specific windows during which they cannot install or test new versions of software. This may affect the release schedule you have devised and it may make on-time delivery of software by the supplier even more critical as small schedule slips could result in long in-service delays.

The operational requirements of a software product include:

- The software will need to integrate with a variety of systems monitoring software frameworks.
- The software may need to work in a variety of hardware plus software configurations for both server and client (possibly browser) components. This may require extensive

compatibility testing. It may also constrain the features to be delivered due to cross-platform issues that restrict the functionality to a lowest-common denominator.

8.4 Dealing with Large Products

When a product is too large, complex or includes too many different technologies, it can be difficult to commission a single team to develop it for you. Another challenge is the integration of several existing products each of which has its own supplier organization. If you have this issue you have several ways to deal with it. The more obvious solution is to break down the product requirements into separate requirements for each subcomponent and commission component teams to do the work. The main issue with this from a product management perspective is that it makes product architecture and the subsequent subcomponent integration ***your*** problem. See the sidebar Component Teams for a more detailed discussion.

The other alternative is to commission feature teams that work across components and divide the product requirements into subset of product requirements for each feature team. This has the advantage of dealing with any integration issues within the feature team. The supplier organization may need to provide a mechanism to ensure consistency of approach across the feature team such as a virtual architecture team or standards team. See the sidebar Feature Teams on Microsoft Office for a more detailed discussion. Regardless of which approach you choose, you'll want to have some mechanism for coordinating the actions of the teams. Two common approaches are the use of project reviews and the Scrum-of-Scrums approach.

Sidebar: Component Teams

A large client had organized their teams around the components of their architecture which consisted of many components organized in several layers. The bottom layer is called "The Platform". Built atop The Platform are many reusable components and services we can simply refer to as A, B and C. These components are used by the customer-facing components X, Y and Z. When marketing receives a new feature request the product manager decomposes the requested functionality into requirements for each of the customer-facing components X, Y and Z. The architects of the corresponding teams are each asked to do a feasibility study of the functionality and to provide an estimate of the effort. This may require requesting new capabilities from one or more of components A, B or C which may in turn require new capabilities from The Platform. Once the feasibility study was completed and an estimate was available from each of the affected teams, marketing would decide whether to build the feature and if so, which release to include it in. In the appropriate release, each of the teams involved in the design would do the work required in their individual components. When all the components were finished, "big bang" integration would start. This would often discover serious mismatches between expectations and what was built.

Sidebar: Feature Teams on Microsoft Office

A feature is an independently testable unit of functionality that is either directly visible to a customer (customer-facing feature) or is a piece of infrastructure that will be consumed by another feature. Feature teams are small interdisciplinary teams (5-12 people) that focus on delivering specific product features. A sample feature crew could be the one in charge of Ribbon in MS Word. Another one is the one in charge of Spellcheker. A third one is in charge of Address Labels.

The idea is for all disciplines (development, PM, UxD, testing) to work closely together on private builds (in their own, isolated private *feature branches*) and only add the feature to the product (the *main branch*) when it is "Feature Complete" (similarly to "Done-Done" described earlier). The feature in "Feature Complete" state is expected to be fully implemented, sufficiently tested and stable enough for either a) dog-fooding (using one's own product in-house) and sharing with customers in a CTP (Community Technology Preview) for customer-facing features, or b) coding against by another feature crew – for features that deal with underlying infrastructure. The typical duration for feature delivery is between three to six weeks – the idea is to ensure a steady flow of features and regular, frequent feedback from the customers. Importantly, each feature team is free to decide what development approach/process to use, as long as their output meets all quality gates (common for all feature crews) and doesn't destabilize the product. (For a case study on the use of the Feature Crew methodology at Microsoft Visual Studio Tools for Office product unit and details on integration of features into parent and main branches, see [Miller & Carter])

8.5 Bug Tracking and Fixing

Users and testers will report bugs no matter how good a job you do building the software. Some of these "bugs" will be legitimate defects introduced into the software accidently. Others will be requests for new functionality you consciously chose to leave out of the release. Except from a contractual (who pays) perspective, bugs need not be treated any differently from feature requests. Either way, you need to decide what timeframe you want the functionality of the system changed and therefore in which stream of software you want to include it. The Concern Resolution Model provides a common way of thinking about bugs, feature requests and other project issues.

8.6 Maintaining Multiple Versions of the Software

There will be times that you may choose to maintain several versions of the software at the same time. A common situation is after release, during the warranty period, if you are also developing a subsequent release at the same time. Another situation is when you choose to support multiple version of software in production and need to apply any bug fixes to all supported versions. Be forewarned that there is an extra cost to maintaining versions in parallel as any bug fixes done in the support/warranty stream will need to be propagated into the development stream at some point and retested there.

Chapter 9 Test Manager's Perspective

Some organizations have a separate Test Manager role while others have the testers reporting directly to the Development Manager or even the Business Lead or Product Manager. When the Test Manager role exists, the test manager is usually responsible for planning the bulk of the testing activities and managing/coordinating their execution. The test manager may or may not act as the gate keeper – that is, make the acceptance decision on behalf of other stakeholders. It is important for all parties to understand the role of the Test Manager in this regard.

See the Test Manager persona in the company stereotype called "A Product Company" in Appendix X – Reader Personas for an example.

The Test Manager's role can be a difficult one to do well because so many project factors are outside your control. Test planning is essential for all but the most trivial projects; what the test plan specifies depend heavily on the role of testing as it relates to the supplier organization and the product manager.

9.1 Test Manager's Role in Acceptance Decision

The role of testing seems simple enough: use the product in various ways and report any bugs that you find. But this is not the only responsibility of the test organization in many enterprises. Make sure you understand whether you and the testing team are doing readiness assessment, acceptance testing or making the acceptance decision.

9.1.1 Testing as Acceptance Decision Maker

In some circumstances. The test organization is expected to collect data to help the product own decide whether or not the software is ready to be released to users. Make sure you have a clear understanding of whether you are the gate-keeper (acceptance decision maker) or are supplying information to someone else who will be making the decision (preferred.) Recognize that the acceptance decision is a business decision. If you are going to make it, you had better understand the business factors well enough to make a good decision that you can explain to everyone. Otherwise, you'll just be the bad guy holding up the show.

If you are to act as the Acceptance Decision Maker, you must have access to all the business-relevant information to make a good decision. That is, you must understand the business consequences of:

- accepting the software given the nature of the bugs that are known to exist and the level of confidence that all serious bugs have already been found, and,
- **not** accepting the software thereby delaying the delivery.

Each of these choices can have serious negative business consequences and the acceptance decision cannot be made without a full understanding of them. The Test Manager should not accept responsibility for making the acceptance decision lightly. In most cases it is better for this business decision to be made by a business person, typically the product manager, business lead or operations manager based on information provided by the test manager.

9.1.2 Testing as Acceptance Testers

When you are providing information to another party to make the acceptance decision, make sure you have common understanding of what information that party (or parties) will require to make the decision. Ensure the testers understand the users for whom they are acting as a proxy so they can define test that reflect realistic (not necessarily always typical) usage of the software. User Models (either user roles or personas) can be an effective way to gain understanding of real users.

9.1.3 Testing as Readiness Assessors

If you are doing readiness assessment to help the Dev Manager decide whether the software is ready for acceptance testing by the business users, you'll want to build a good relationship with the dev team so that you can help them understand the quality of what they have produced and how they can improve it. Embedding testers with the development team (often used in conjunction with a technique called pair testing) can be a good way to help them learn how to build quality in by testing continuously rather than tossing untested software over the wall to the test team. (Kohl describes benefits of pair testing in this experience report [Kohl])

9.2 Test Planning

You will probably be expected to co-ordinate all testing activities, test environments, software version management and bug tracking all the way from readiness assessment through to acceptance testing. This work typically goes by the name Test Planning. You'll need to communicate this plan to all interested parties; especially those who are expected to execute parts of the plan and those who are interested in knowing what kinds of testing will be done by whom, where and when. This communication often takes the form of a Test Plan document but it can also be done or complemented orally through targeted presentations, discussions or workshops. Many of the best test plans come about by inviting the interested parties to participate in the test planning process either in one-on-one sessions or via workshops.

9.2.1 Test Strategy

You will likely end up being the de facto owner of the overall test strategy. The only other contender would be the supplier organization and if they want to be involved, probably on the test automation side, by all means encourage their involvement because it will make test automation much easier.

The test strategy includes the major decisions such as whether testing will be done in a final test phase or incrementally, whether testing will be primarily script-based testing or exploratory testing or a

combination of the two. Exploratory testing is a very effective way of finding bugs fast but it isn't intended to be repeatable. Script-based testing is better for ensuring a known degree of test coverage and should be automated to allow rapid regression test execution.

Another key decision is regarding the role and degree of use of automated tools including automated execution of functional and para-functional tests, the use of model-based test case generators and the use of automation as power tools (including the use of automated comparators or verifiers) to support manual test activities. Some of these decisions require long lead times to decide whether to choose & acquire or build tools or to hire the appropriate resources and therefore must be made relatively early in the project. This doesn't imply that you won't refine them over time as more detailed information comes to light.

The test strategy will be heavily influenced by the nature of your relationship with the organization developing the software. The ideal case is when they are prepared to collaborate with you by doing incremental delivery of functionality to support incremental acceptance testing. Collaboration on test automation and design for testability is also a key factor in reduce test execution time and cost. If this level of collaboration is not available you may need to hire test automation engineers and test toolsmiths with strong programming skills to support your test automation activities.

9.2.2 Test Automation

Another area that greatly benefits from collaboration with the supplier organization is the area of test automation. A key success factor is design for testability of the system. Only the supplier organization can do this so you need to work with them to build it in. A good way to incent them to collaborate is to provide them with access to the 1-button automated execution of the acceptance tests you define. This provides the supplier organization with an instant scorecard that tells them how they are doing at delivering quality software. Everyone hates surprises and nothing annoys a development team more than working hard to deliver quality software and being told after the fact that it wasn't good enough.

While the development team needs to make the system amenable to testing, the test automation tools can be built by either the supplier organization or the test team. The latter will require technical testers, sometimes called test automation engineers. Either way, the tests should be developed primarily by the testers or business people/analysts to ensure they reflect requirements rather than design decisions.

9.2.3 Agreeing on Expectations/Requirements

One purpose of testing is to verify the system meets the requirements. Unfortunately, the requirements are often vague or ambiguous and this makes verifying them a difficult proposition. One person's bug can be another person's feature. Testers think differently from analysts or business people and this is both a blessing and a curse. They'll come up with all manner of test scenarios that the requirement specifiers never imagined. This results in a de facto divergence between the requirements and the specifications that need to be managed. The product owner (Product Manager, Business Lead, etc.) needs to agree that these additional test scenarios are in fact requirements.

The requirements need to address the needs of all stakeholders, not just the users. Operational requirements need to be tested. If the test team doesn't have the skills to execute the para-functional tests then make sure that the supplier executes them and provides test plans to review ahead or time and test results as part of the handover to testing. The acceptance decision maker will to know that the para-functional testing was done and may even want to see the results.

The preferred solution is to treat test scripts as an extension of the requirements by using them to illustrate how specific requirements play out in the product. This implies that the product manager or business lead (or their team members) need to agree that the tests interpret the requirements correctly. Ideally, the test cases would be articulated before the software is built so that the development teams can use the test cases to understand how the system should behave. This is known as Acceptance Test Driven Development.

9.2.4 Agreeing on Done

Are we done yet? That is a question that needs to be asked and answered continuously. And done doesn't always mean the same thing. One person's definition of done may be another person's definition of "not even close to done!" It is critical to get agreement on how done software needs to be before it goes through each of the quality gates of the gating model. When is software considered "ready for acceptance testing"? When is it considered acceptable? These need to be agreed upon between the various parties involved. The supplier and testing need to agree on the minimum quality requirement of software before it is ready for testing. This becomes the exit criteria for the readiness acceptance phase that the supplier team must ensure is met. A prominently posted Done-Done Checklist is a good way for the supplier team to keep this quality criteria front and center.

9.2.5 Estimating Test Effort

Estimating the amount of time it will take to get through one complete cycle of testing is a challenge but one that can be overcome with experience. The real challenge, however, is guessing how many test&fix cycles will be required before the software is of good enough quality to release to acceptance testing or to the market. The need to guess can be avoided by delivering software incrementally and testing each increment as soon as it is available. This helps in several ways:

- It reduces the amount of software that hasn't gone through the test&fix cycle at least once when the final round of testing occurs. This is a form of inventory reduction where inventory is considered a form of waste. (See [Poppendieck & Poppendieck])
- It provides data on how many test&fix cycles are required for software delivered by this supplier organization early enough to allow the test plans to be adjusted to ensure on-time delivery of software.
- It provides feedback to the supplier team on the quality of the software they are delivering early enough to allow the supplier team time to learn how to deliver better quality software that requires fewer test&fix cycles. The Done-Done Checklist may need to be updated based on what was learned.

9.3 Concern Resolution

Testing will invariably result in a number of concerns being raised. Some of these concerns will be based on failed test cases and others may be based on general impressions about the product. All concerns, however, are based on expectations and it is important to ensure that the expectations are correct. Test cases that don't map to actual requirements can result in bugs that will be closed as "Design Intent". Some failed test cases may point out inconsistencies between how the system operates and how the tester expects the system to operate. These may point to legitimate usability issues if the tester's mental model of the system matches that of real users.

Part of the role of many test organizations is to act as the second opinion, the house of sober second thought, about the software being built. It is important to balance this form of requirements elaboration with the need to get the product out in a timely, and even more importantly, predictable fashion. Discovery of such requirements issues during a final testing phase makes the product manager's job harder because the information comes too late to address without compromising the schedule. Therefore, the test manager should strive to find such issues as early as possible so that the product manager can make decisions without their back to the wall.

The Concern Resolution Model describes a generic way to think about all manner of concerns including bugs, requirements issues, project issues and other mismatches between expectations and actual or perceived behaviors.

Chapter 10 Development Manager's Perspective

The development manager may go by many official titles but for our purposes, this is the person in the supplier organization who is accountable for delivering software to the product owner (PO) whether the PO is the internal business sponsor of an IT project or the product manager in a product company. Where a separate test organization (often called Quality Assurance, Independent Verification or some variation on these) exists for the purpose of doing acceptance testing, the development manager is responsible for making the readiness decision, that is, the decision to deliver the software to whoever is responsible for making the acceptance decision. For larger products the development manager may be assisted by a solution architect; for smaller products the development manager should understand the solution architects responsibilities and either carry them out themself or delegate to some in the development team.

See the Development Manager persona in the two company stereotypes in Appendix X – Reader Personas for an example.

As the Dev Manager your key responsibility is to manage the development of the software in such a way that the product owner (product manager, business lead, etc.) will accept the software. You should do this in a way that maximizes the value (or utility) that the software will provide to the product owner (by means of value provided to their users or customer) while minimizing the cost. The best way to minimize cost is to build the right software and to build it the right way the first time and deliver it to the acceptance testing organization and/or product owner as soon as possible. This avoids expensive and time consuming test&fix cycles in which someone else tests the software, finds bugs which they then ask you to fix. This delay (a form of waste) contributes significantly to churn and cost. Whether this is the product owner or someone looking out for their interests, the fewer bugs they find, the less rework you need to do, the lower the overall cost and the more predictable the schedule becomes.

10.1 The Role of Readiness Assessment

Most development teams want to do good work. They don't want to deliver substandard software to their customer. Most customers want to receive good quality software.

Delivering good quality software is what the customer expects from the supplier. This is a reasonable expectation, one that a professional software development organization should be prepared to meet. Delivering good quality working software isn't easy or trivial; if it were, the product owner probably wouldn't need you!

So, what does it take to deliver good quality software, first time, every time? What does it take to be a professional software developers? The complete details of a sound software engineering process are beyond the scope of this book. But, the relationship between the software development organization and the parties involved in acceptance the software are not. Part of the development process needs to be an honest self-assessment of the software the team has produced. If the team says it is ready, give it to the test team. If the team says it isn't ready, ask them what still needs to be done to make it ready. A good development team will always be able to say *something* is ready and should be able to clearly articulate to the test team what is ready and worth testing. According to Robert C. Martin, it is irresponsible for a developer to ship a single line of code without a test. Furthermore, tests must be kept to the same level of code quality as the production code. [Martin]

10.2 Effective Readiness Assessment

10.2.1 Who's Job is Quality?

How would you feel about being the first person to try a brand new product that no one, not even the builder had tried before. How many people you know would be comfortable in that situation? If we apply the same logic to the software we build, how many people do you know that would want to be the first one to try a piece of software that no one else has tried before? Yet this is exactly what many deve*lopment t*eams do when they throw untested or poorly tested software "over the wall" to the test organization or the customer. Many people would argue that this is just plain unprofessional.

All software should be tested by the development organization before being handed to anyone else for testing.

10.2.2 Building Quality In – Start with the End in Mind

Building software should not feel like a guessing game where developers guess what's required and testers or users shout out "wrong!" The development organization needs to have a clear understanding of what success looks like before it starts building the software. Part of the problem is that development teams often don't have a good sense of how the software will actually be used. Developers do not think like a user because most developers thrive on complexity while most users abhor it. This makes it challenging for developers to do a good job of readiness assessment without outside help. Acceptance tests provided by the product owner or testing done by testers can bridge this gap. The former is proactive, positive input that helps development teams understand "what done looks like" before they build the software. The latter is negative, reactive feedback that tells the development team "you haven't done a good job." What kind of guidance would *you* prefer?

A project charter is one way to start the process of developing this common understanding. Requirements documents, user models, use cases and user stories are all ways that we try to develop a common understanding between the supplier and the customer. But these static documents written in natural language typically contain many ambiguous statements and in most cases do not provide enough to sufficiently describe what success looks like. The designs and work estimates provided by the

development team therefore likely contain assumptions, many of which are either wrong or will result in a product that is more complex and expensive to build. It is critical to clarify these potential misunderstandings as quickly as possible.

As the development manager you should work with the product owner and acceptance test team to ensure that everyone on the team has access to the definition of success in the form of acceptance tests before the software is built. This should include anything that is used as an acceptance criteria including both functional tests that verify the behavior of the system feature by feature and para-functional quality criteria such as security, availability, usability, operational criteria etc. This process is known as Acceptance Test Driven Development (aka Example Driven Development or Storytest Driven Development). This may require that product owner or testers prepare acceptance tests earlier in the project than they might have traditionally done. It may also require them to be more involved for the duration of the project rather than just at the beginning and the end so that they can answer any questions that come up during the interpretation of the requirements and also do incremental acceptance.

10.2.3 Defect Prevention before Defect Detection

Traditional approaches to testing focus on defect detection. That is, the emphasis is on finding bugs rather than preventing their occurrence in the first place. How can the emphasis be changed to prevention? It isn't enough to define the tests ahead of time; we must also run them frequently. By doing so, we always know the score. The test results tell us how far from "done" we are. They provide a very clear indication of the progress towards done, one that doesn't require a lot of extra work to calculate and one that is hard to fudge. The test results, available throughout the project, provide the stakeholders with visibility into the project in a much more transparent fashion than traditional metrics measuring progress against a phased-activity project plan.

To prevent defects we define the tests before building the software, automate them, and run them frequently while building the software. We institute simple team norms to avoid regressing: tests that have passed before must continue to pass as new functionality is added. No exceptions! Either the test has to be changed (changes to functional acceptance tests may require the customer's agreement) or the code has to be changed to return the test to passing status.

What role do the various kinds of tests play in this process? Functional tests defined by the acceptance testers define what done looks like and the software cannot be delivered to them when any acceptance tests are failing. Automated unit and component tests define the design intent of the software; writing these first and writing just enough code to make them pass ensures we don't build unnecessary software and that the software we do write satisfies the design intent. It also ensures that the software you build is designed for testability, a critical success factor for test automation. Another benefit is that they act as a large change detector that will inform the team of any unexpected changes in behavior of the software. This helps the team catch regression bugs before they can sneak through to the users.

10.2.4 Reduce Untested Software

In lean thinking such as exemplified by the lean manufacturing paradigm used by Toyota [Shingo], unfinished work (inventory) is considered a form of waste. In software development, software that has been written but has not been accepted is unfinished work. We should strive to finish this work as soon as possible by doing acceptance testing as soon as possible after the software is written and readiness assessment is completed. This incremental acceptance testing requires collaboration between development, testing and the product owner as all parties must be prepared to work feature by feature rather than waiting for the whole system to be available before any acceptance testing is started.

The minimum quality requirement (MQR) should be agreed upon ahead of time with the testing organization or the product owner. Ideally, any and all tests that will be run as part of the acceptance testing should be run by the supplier organization before making the software available for acceptance testing. Known defects should be identified ahead of time to avoid wasting people's time testing broken functionality.

10.2.5 What Kinds of Tests Are Required?

Functional tests should be run as soon as the corresponding functionality is built. This should include *business workflow tests* that verify end to end business processes, *use case tests* that verify the various scenarios of a single use case or business transaction, and *business rule tests* that ensure that business rules are implemented correctly.

Operational requirements also need to be verified by ensuring that acceptance tests provided by the operations stakeholders are run and results are analyzed regularly.

Para-functional testing should be done on a regular basis as soon as enough software is built to allow them to be run. The earlier they are run, the more time the supplier has to correct any deficiencies that are discovered. This is especially important with para-functional tests because changing the para-functional attributes of the system may require a change in architecture, a proposition that may get more expensive the later the change is made.

Many of these tests can be done much earlier in the project if the early focus of the project is to build a walking skeleton of the application. The walking skeleton implements the full architecture in a very minimalist way. For example, all the logic might be hard-coded thereby supporting only a single highly-simplified business transaction or workflow. But all the major architectural components would be present to ensure the runtime characteristics were truly representative of the finished product even if the functional behavior was not implemented.

10.2.6 Sharpening the Saw

Any bugs that are found during acceptance testing should be a surprise and should prompt the question "how did that slip through *our* readiness assessment? Clearly, the software wasn't truly ready." If the answer is "because they used it a different way from what we expected" then the supplier organization has to do a better job understanding the users of the software. Therefore, every bug found becomes a learning opportunity for the team by prompting them to look for ways to improve how they build

software. For example at Mircrosoft's patterns & practices group, during the Web Service Software Factory: Modeling Edition project, the team did just that on several occasions, modifying their CI system to prevent issues from reoccurring.

As manager of the development organization you need to create the conditions in which the development team *can* produce the right software built the right way. This may require overcoming organizational and cultural hurdles. You need to work with your counterparts in the test organization to break down the traditional adversarial relationship (if one exists) and work together to create a more collaborative relationship. As Ade Miller, patterns & practices development lead, eloquently put it: "I think one of the key things a Dev Manager can do here is try to send the clear message that testing and the test organization are important. In far too many cases testers are seen as some sort of lesser function because they "lack" the technical skills of developers." The test organization has the skills to help your team build better quality software, not just to tell you when you haven't.

Chapter 11 User Experience Specialist's Perspective

Some organizations, especially those that build consumer products, have a separate role that is responsible for ensuring a good and consistent user experience. What is the role of people with this kind of skillset in the making of the acceptance decision?

See the User Experience Specialist persona in the company stereotype called "A Product Company" in Appendix X – Reader Personas for an example.

As a person with the responsibility of user experience you may be responsible for designing the user experience associated with the product. This may involve various kinds of activities to better understand and characterize the potential users including ethnographic research, user modeling, task modeling, etc. It likely also includes verification activities such as usability testing at various points throughout the design and development process.

11.1 The Role of User Experience in the Acceptance Decision

It is important to have a clear understanding with the product owner (product manager or business lead) as what your role shall be in making the acceptance decision. It could be any of:

1. Being an input into the product definition process but no direct involvement in the acceptance decision. That is, helping the product owner define the product but leaving it to other parties to define the acceptance criteria.
2. Being an input into the software development process with no involvement in the acceptance decision. This is similar to the first point but working more closely with the software development team rather than the product ownership / definition team.
3. Providing usability testing data for the acceptance decision made by the product owner.
4. Having a say, or even a veto, in the acceptance decision either as part of the Product owner's team or as a separate parallel acceptance decision.

For products where usability is a key differentiator, the preferred role would be either 3 or 4. For products with captive users such as systems developed for internal users it is more likely to be 1 or 2.

In any of these models it is important to work with the business lead or product manager to define the usability component of the acceptance criteria as early as possible to give the development team as much guidance as possible. Early in the project this may consist of descriptions of the kinds of testing procedures that will be used to verify usability; later in the project, it may be specific functionality to be

tested such as the novice user's top 5 transactions, specific usability guidelines to be complied with, and so on.

Where usability is important and how to achieve it is unclear, it is crucial to address the usability risks as early as possible. Some of these risks may be addressed through the use of low cost usability techniques such as paper prototyping and wizard of oz testing. Others may require the construction of electronic prototypes for higher fidelity usability testing. Construction of special purpose high-fidelity prototypes may be avoided if the actual system can be built and delivered incrementally sequenced by usability risk. This may require working with the product owner to help them understand the risks and consequences and to use these to influence the order in which functionality is built. Usability-related risks can be mitigated by defining an internal milestone at which sufficient functionality is available to allow usability testing of the actual product with real or proxy users. This requires collaboration between the product owner and the supplier organization to do incremental development and delivery of the software based on the usability test schedule. The areas with highest impact of usability issues should be built and tested first to ensure enough time for any usability-related issues, which may be pervasive enough to have a wide-ranging impact, to be addressed before the cost of change has become too high.

Where the product owner has not specified incremental delivery it may be possible to work directly with the development manager to arrange for opportunities to do usability testing using early versions of the actual software system.

Chapter 12 Operations Manager's Perspective

Some organizations build or commission software that they install and run to provide a service to their customers or internal users. In these organizations the acceptability of the software may be determined by two entirely separate communities, those that benefit from the functionality that the software provides and those for whom operating the software is a cost. In these organizations the acceptance decision must include operational criteria and the role of the operations group in the making of the acceptance decision must be clear. This category could also be expanded to encompass other groups such as security, data architecture, IT architecture and infrastructure that provide stewardship of corporate interests to counterbalance potential project-pressure induced shortcuts such as security, data architecture, IT architecture and infrastructure. (See Chapter 14 - Enterprise Architect Perspective.)

See the Operations Manager persona in the company stereotype called "IT Building Software for the Business" in Appendix X – Reader Personas for an example.

As an operations manager you likely have a completely different perspective on the deployment of a new or upgrade software system. Other parties care about whether the new functionality works properly and whether it will provide sufficient business benefit in the near future; you care about what it will do to your support costs and whether it will be sustainable in the longer term. Therefore, your idea of acceptability will be quite different and it should influence the acceptance decision.

12.1 Role in Acceptance Decision

It is crucial that your role in the making of the acceptance decision is clear to everyone concerned. Do you provide information to the acceptance decision maker or are you one of several acceptance decision makers who operate by consensus, voting or veto? Or perhaps you participate in the readiness decision before the software is made available for acceptance testing?

As with other forms of acceptance criteria it is highly desirable for these criteria to be clearly defined before the software is built so as to avoid unnecessary test&fix cycles before the software can be deployed. It is also desirable to being able to conduct the testing incrementally throughout the project rather than just at the end. Incremental Acceptance Testing decreases project risks by uncovering issues early enough to have them fixed before the final acceptance test phase which is then much more likely to pass without uncovering any show-stopper problems.

12.2 Operational Acceptance Criteria

The actual criteria that an operations group will use to determine acceptability will be dependent on the nature of the organization and the services it provides as well as the service level agreement for the software application in question. The base level requirements typically include:

- Installer and uninstaller testing
- Auto-update or patch installation
- Compatibility with your Systems monitoring framework.
- Systems management scripts for startup, restart and shutdown.
- Etc.

Additional acceptance criteria may include:

- Frequently-asked-questions and answers
- Trouble-shooting scripts for the helpdesk
- Failover testing for high availability applications
- Penetration testing for public facing applications
- Stress testing for applications with potentially large numbers of users
- Data migration from a previous version of the system or from a system this one replaces.
- IT Architecture approval of the technology stack.
- There could be requirements of performing such service operations without degrading the current system
- Reliability, availability, … meeting certain SLAs(uptime etc.)
- Self-monitoring, error detection, system logging
- Recoverability (must verify multiple failover and recovery), disaster recovery, data backup & recovery etc…
- Support documentation (pre and post roll-out), training…
- The system must be migrated to the new version of the platform
- Multiple versions of the system must run side-by-side

Chapter 13 Solution Architect's Perspective

The solution architect is typically part of the supplier team. Their involvement in the acceptance process usually involves approving design and technology decisions, designing data models and domain models, and ensuring that the product as built satisfies the needs of the end users. The role transcends different parts of the supplier organization as the solution architect is responsible for the whole solution whereas individual component teams or feature teams are each only responsible for a part.

As a solution architect, your job is to ensure the entire solution satisfies the product owner and the potential users of the system. Your role in the acceptance process will primarily focus on the readiness assessment activities but you should also be involved in helping the product owner understand what they want the supplier team to build and facilitate the communication of that information to the supplier team. Your involvement in acceptance testing may be minimal but you may be called upon to interact with acceptance testers which may include enterprise architects assessing compliance to architectural guidelines and standards.

13.1 Understanding the Solution

You need to work with the product owner to understand their vision of the product and to help them understand what options are available to them with respect to building that product. Less technical product owners may need your help to define the product based on more general needs. You will likely want to engage user experience experts if good usability is a requirement. You will also need to help the product owner understand the relative costs of various options presented to them so they can make the right tradeoffs based on return on investment. This may require unbundling of some functionality into smaller, more atomic, features that can be prioritized separately.

Be aware that the product owner may not be aware of all the requirements for the product; it may be up to you to help them engage the operations manager, security architects, and other stakeholders so that all the requirements for the product become known. Discovering security requirements or mandatory service level agreements late in a product development cycle is sure to result in a late product.

For larger products you may be called up to help make the project manager or product owner decide whether to divide the supplier team into feature teams or component teams and to subdivide the feature backlog into feature team backlogs or decompose the feature backlog into component backlogs. Where a mix of technologies is involved (including mixed hardware/software products), you must be prepared to be involved in negotiating the interfaces between the various technologies.

Regardless of the size of the supplier team, it is important for everyone to understand the overall solution and how their part fits into it. This can help avoid situations where each component works

according to specification but when integrated fail to provide the seamless experience the customer would expect.

13.2 Ensuring Readiness

As solution architect, the onus is on you to do everything you can to ensure that the product is ready before the readiness decision needs to be made. Things will go a lot smoother if everyone on the team understands what done looks like. This is especially true for component teams whose work needs to be integrated before acceptance testing can be done. With component teams, you may need to do a form of acceptance testing of the components followed by integration testing of the integrated components before you can conduct readiness assessment. This will be easier to do if you have been doing design reviews of each of the components and have defined clear interfaces and acceptance criteria.

Some of the acceptance criteria may be derived from enterprise requirements. Where enterprise architects exist, it is important to engage them early in the product development lifecycle to ensure you understand what standards the product needs to comply to, what reusable components are available and/or are expected to be used and what enterprise data impacts the product (and vice versa.) This will avoid nasty surprises during readiness assessment and acceptance testing.

13.3 Assessing Readiness

As solution architect, you will likely be asked for your opinion as to whether the product is ready for acceptance testing. You'll want to make this recommendation based on hard data and will therefore want to conduct architecture reviews and security code inspections and to run static code analysis and coverage tools. You will also want to understand how well the para-functional characteristics of the system comply with the requirements; that will require testing of the para-functional characteristics. You may be involved in the actual testing, either in person or through planning and delegating, or you may review results provided by someone else such as the test organization. Either way, you will want to ensure that the system meets any operational requirements, including SLAs, in addition to the functional requirements.

Chapter 14 Enterprise Architect's Perspective

The enterprise architect is responsible for ensuring consistency between products employed in or built by an enterprise. Their goal is to reduce total lifecycle costs (including operating costs, technology licensing costs, etc.) and to maximize the ROI of any common infrastructure. The enterprise architect often has veto powers over individual products that fail to comply with enterprise architecture principles. This sometime makes them part of the acceptance decision– when the product is built by an external supplier – and sometimes part of the readiness decision – when the product is being built in-house by an IT department.

As an enterprise architect, your job is to ensure the all the products used by the enterprise work well together and provide acceptable performance at an acceptable cost. Depending on the nature of your organization you may be called upon to provide data for either the readiness decision or the acceptance decision. In some organizations you might even be asked to be involved in making the acceptance decision. The role you play will depend on where in the organization you report relative to the supplier organization. In out-sourced development situations you are more likely to be involved in acceptance testing and the acceptance decision; for in-house software development and product companies you may be more involved in readiness assessment. Who you interact with will also depend on the nature of the organization. If there is a solution architect for each product, much of your interaction with the supplier team will be with the solution architect. In other cases you might be interacting primarily with a development lead or a project manager or even the product owner.

14.1 Understanding the Solution

While it is primarily the supplier team's responsibility to work with the product owner to understand the desired product, you may need to be involved in these discussion if the supplier team doesn't have a solution architect. Otherwise, you would likely work with the solution architect to undersand how the requirements of the specific product relate to the rest of the enterprise architecture. You would determine (along with Solution Architect) what forms of integration are required and how that integration should be implemented from a technical point of view. You would also collaborate to understand how the new product impacts any common infrastructure such as network, application and database servers, etc.. If the product has any impact on the long-term vision of the enterprise architecture, either by changing the vision or by implementing any key components, that information needs to be communicated to all parties concerned as the latter may become part of the readiness and/or acceptance criteria.

14.2 Ensuring Readiness

In some organizations you will be asked to provide readiness assessment information to the readiness decision maker. You will need to engage the solution architect or development leads to ensure they understand what standards or guidelines are applicable to the product and what reusable components should be used. You may also be involved in defining extensions to the corporate data models to support the new product. It is crucial that you communicate any expectations of how the supplier team would engage the enterprise architects during the readiness assessment and/or acceptance testing (whichever is applicable.)

14.3 Assessing Readiness or Acceptability

The nature of your organization will influence whether you are formally involved in readiness assessment or acceptance testing but the nature of the activities you do will probably be the same either way. Unfortunately, the later you are involved the bigger an impact a non-compliance discovery will have on the overall delivery schedule. Therefore it is preferable to engage the supplier teams earlier, before even readiness assessment, to ensure that the readiness assessment or acceptance testing/reviews are just a formal signoff of information that was previously discovered rather than an unpleasant surprise for everyone involved.

Some of the activities you might want to start doing even while development is in progress include design and architecture reviews that focus on standards compliance para-functional attributes such as security, capacity, scalability, and infrastructure impact. In some cases you may even be called upon to arrange for testing of para-functional characteristics of the system such as penetration testing.

If you find yourself on an acceptance decision committee, you will need to be prepared to gather and review the results of the para-functional reviews and testing and cast your vote on the acceptability of the product from a technical perspective. Let the product owner vote based on the functionality provided and the operations manager on their satisfaction with the operational requirements.

Chapter 15 Legal perspective

The decision to accept software is often governed by terms in a legally binding contract. The acceptance decision may therefore have legal ramifications which may override considerations related to functional and para-functional acceptance criteria

Disclaimer: This chapter is not meant to be used as legal advice.

Since acceptance is a major legal milestone of commercial software system development, keep in mind the following considerations:

15.1 Acceptance May Have Legal Ramifications

As part of the contractual process, acceptance typically results in rendering a payment as well as it affects the application of any warranty provisions and potentially any remedies available to the customer. When defining the contract, make sure that the appropriate factors are taken into consideration. The relationship between the acceptance of the product into a production environment and the contractual acceptance of the product as "complete" should be clearly spelled out.

Acceptance can be approached from two positions: explicit provisions specified in the software development contract and the terms implied into the contract by law (which also depends on the geographic jurisdiction). Implicit provisions may also differ based on whether the software is rendered as a product or as a service. To see what provisions (explicit and implicit) do apply to your case, consult your legal counselor.

15.2 Contract Should Stipulate Acceptance Process

When software development is outsourced, the contract should clearly stipulate the acceptance process and how the acceptance criteria will be determined. It is important to also specify any expectations about how much readiness assessment will be conducted by the supplier and how the criteria for readiness assessment will be determined. For example, will the customer be providing the supplier with the acceptance tests, a representative sample of the acceptance tests or leaving the supplier to determine what tests to run (the "Battleship ™ approach".)

The contract should not just stipulate when the supplier must deliver the system but also how long the customer can take for acceptance testing. Acceptance testing is usually time-boxed. The customer is given so much time to accept software and report deficiencies, after which the software is deemed complete if there are no "show stopper" bugs. The contract must stipulate the criteria used to determine whether a bug is indeed a "show stopper" or "gating" bug.

The contract should also stipulate where the different steps of the acceptanc e process will be carried out. In case of custom build software system, preliminary acceptance testing may be performed at the

location of supplier, while final acceptance testing (both functional and operational) is performed when the software is deployed to the customer. The contract may stipulate that, as part of the acceptance procedure, the software system be successfully run with current data in a live operating environment for a *period of time*, with minimum bugs, before the system is contractually accepted.

The contract should not be one-sided; it should encourage both parties to get to a mutually agreeable acceptance as quickly as possible. Contracts that penalize one party often lead to dysfunctional behavior that results in lose-lose outcomes.

Part II - References

[Wittgenstein] Wittgenstein, L. Philosophical investigations, 3/e, Oxford: Blackwell, 1967.

[Poppendieck & Poppendieck] Poppendieck, M., Poppendieck, T. Implementing Lean Software Development: From Concept to Cash. Addison-Wesley, 2007

[Drucker] Drucker, P. Management Challenges for the 21st Century, Harper Business, 2001, p. 33.

[Perry et al] Perry, J., Barnes, M. "Target cost contracts: an analysis of the interplay between fee, target, share and price". In J. Engineering, Construction and Architectural Management, 7(2): 202-208, 2000.

[Eckfeldt et al] Eckfeldt, B.; Madden, R.; Horowitz, J. "Selling Agile: Target-Cost Contracts." In Proc. Agile 2005. IEEE Computer Society: 160-166, 2005.

[Miller & Carter] Miller, A. & Carter, E. Agile and the Inconceivably Large. Proc. Agile 2007, IEEE Computer Society: 304-308, 2007.

[Kohl] Kohl. J. Pair Testing. Better Software Magazine. http://www.kohl.ca/articles/pairtesting.pdf

[Martin] Martin, C. Professionalism and Test-Driven Development. IEEE Software 24 (3): 32-36 2007.

[Shingo] Shingo, S. A Study of the Toyota Production System", Productivity Press, 1989.

Part III - Accepting Software

Part I introduced several models to help us think about the role of acceptance testing in the overall context of the software development lifecycle. Part II described the acceptance process from the perspective of commonly occurring roles played by the people involved in making the acceptance and readiness decisions on typical projects. Part III addresses the process of accepting software, applying the concepts introduced in Parts I and II. In Part III, we outline the main activities that lead to the acceptance decision and describe strategies for their effective management.

Chapter 14 – Planning for Acceptance introduces the practices used by one or more of the roles described in Part I to plan the activities that will provide the decision makers with the data that they need to make an acceptance decision.

Chapter 15 – Assessing Software introduces the acceptance testing and verification techniques used in software assessment, with particular emphasis on test definition, execution and maintenance. It addresses both functional and para-functional acceptance testing.

Chapter 16 – Managing the Acceptance Process introduces the practices used to manage acceptance test execution and acceptance decision-making and addresses the tradeoffs between different approaches to these activities.

Chapter 17 – Streamlining the Acceptance Process describes the organization of the acceptance process to optimize the acceptance schedule and the resources used in the acceptance decision by focusing on three main strategies: incremental development, waste reduction by avoiding "overproduction" and concurrent execution of the activities underlying software acceptance.

Chapter 16 Planning for Acceptance

Part I introduced several models to help us think about the role of acceptance testing in the overall context of the software development lifecycle. It also introduced the abstract roles of the people involved in making the acceptance and readiness decisions. This chapter introduces practices used by one or more of those roles to plan the activities that will provide the data on which the acceptance decision maker(s) can base their decision.

Acceptance is an important part of the life cycle of a system; it is important enough that it should be the result of a carefully thought through process. The test plan is the end result of all this thinking. Like most such documents, it can serve an important role in communicating the plans for testing, but the real value lies in the thinking that went into producing it.

Test planning builds on the work done during project chartering, which defines the initial project scope. In test planning, you we define the scope of the testing that will be done, select the test strategy, and drill down to detailed testing plans that define who will do what, when, and where.

Most projects prepare a test plan that lays out the following, among other things:

- The scope of the acceptance process and the breakdown into readiness assessment and acceptance testing phases
- The overall test strategy including both manual and automated testing
- The activities, testing and otherwise, that will be performed in each phase (the readiness phase and the acceptance phase)
- The skills that are required to perform the activities
- The other resources that will be utilized to carry out the activities (such as facilities and equipment)
- The timeframe and sequence, if relevant, in which each activity will be performed

16.1 Defining Test Objectives

We need to have a clear understanding of the scope of the project and the software before we start thinking about how we will accept the software. Most organizations have some kind of project chartering activity that defines the product vision or scope. It may also include a risk assessment activity. One form of risk assessment activity involves brainstorming all the potentially negative events that could cause grief for the project. For each possible event we classify each of the likelihood and the impact as low, medium, or high. Anything ranked medium/high or high/high needs to be addressed. Some risks may cause us to change the way we plan our project, but other risks may cause us to take on

specific test planning activities. Together, the vision/scope and risk assessment help drive the test strategy definition and test planning.

16.1.1 Lessons Learned from Agile

The agile software development community has shown that it is possible to produce consistently high-quality software without significantly increasing the effort by integrating testing throughout the development life cycle. This has led to a rethinking of the role of testing (the activity) and of test teams.

Brian Marick, a leading contributor to the agile community "with a testing slant" has defined a model (See Figure 1) that helps us understand the purpose behind the different kinds of tests we could execute.

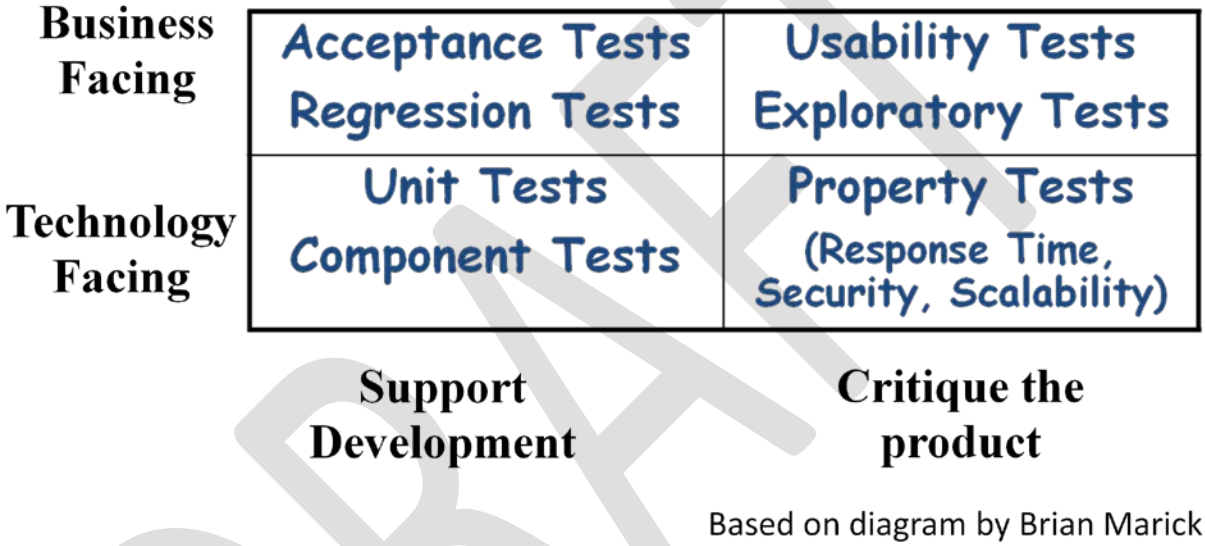


Figure 1
Purpose of Tests

The diagram in Figure 1 classifies various types of testing we can do along two key dimensions:

- Whether the tests are business-facing or technology-facing
- Whether the tests are intended to support development (by helping them get it right) or to critique the product after it is built

16.1.2 Tests That Support Development

Tests can support development by helping us understand what the product is supposed to do before we build it. These are the tests that we can prepare in advance and run them as we build the system. As part of the readiness assessment, the supplier team can run these tests to self-assess whether the system implements the necessary functionality.

The tests in this column fall into two categories: the business facing tests that describe what the system should do in terms understandable by the business or product owner and the technology-facing tests that describe how the software should work beneath the covers.

Business-Facing Tests

The business facing tests that drive development are the functional tests (these are also known as acceptance tests or customer tests). These tests elaborate on the requirements and the very act of writing these tests can expose missing or ambiguous requirements. When we prepare these tests before development starts, we can be sure that the development team understands what they need to build. This is known as acceptance test–driven development.

If we prepare the functional acceptance tests after development is complete or we prepare them in parallel with development and do not share them (this is referred to as "independent verification"), the tests do not help us build the right product; instead, the tests act as an alternative interpretation of the requirements. If they fail when we finally run them, we need to determine which interpretation of the requirements is more accurate: the one implemented by the development team in the code base or the one implemented in the functional tests by the test team. If the latter, time will be consumed while the development team reworks the code to satisfy this interpretation, rework that could have been avoided if we had shared the tests. Either way, we have set up an adversarial relationship between development and testing. It is highly preferable to prepare the tests before the software is built so that testing can help development understand what needs to be built rather than simply criticize what they have built.

These tests may be run manually or they may be automated. The latter allows the supplier to run them throughout the development cycle to ensure that all specified functionality is correctly implemented. The customer will want to run additional acceptance tests to make the final acceptance decision, but supplying a set of tests to the supplier organization early so they can drive development goes a long way toward building the correct product. This is much more likely to happen when the tests are easy and cheap to run—and that requires automated execution (for more information, see the Automated Functional Test Execution thumbnail). These tests may be implemented as programmatic tests, but they are more typically implemented using Keyword-driven Test Automation.

Technology-Facing Tests

There are many tests used by development that are not business-facing. Developers may prepare unit tests to verify that the code they wrote has successfully achieved the design intent. This is how they determine that they correctly built the code (as opposed to building the correct product). Test-driven development (TDD) is when developers implement automated unit tests before they build the code the tests verify. This development process has been shown to significantly improve the quality of the software in several ways including better software structure, reduced software complexity, and fewer defects found during acceptance testing. These tests are ever more frequently automated using members of the xUnit testing framework. For more information, see [Meszaros]

16.1.3 Tests That Critique the Product

Assuming that the product has implemented the correct functionality, we need to know whether the product meets the para-functional requirements. These tests support the acceptance decision. We do this by assessing the para-functional attributes of the system after it has been (at least partially) built. These tests critique the product instead of driving the development process. They tell us whether it is good enough from a para-functional perspective. We can divide these tests into the two categories: business-facing and technology-facing.

Technology-Facing Tests

Technology facing tests that critique the product measure how well the product meets technically-oriented quality attributes (scalability, availability, self-consistency, etc.)

These tests provide metrics we can use when deciding whether the product is ready to be shipped. In most cases, these tests will be run as part of readiness assessment because of their technical nature. However, a customer charged with deciding whether to accept a product may be interested in seeing the results and comparing them with the minimum requirement. They may even hire a third-party test lab to conduct the testing on their behalf. For more information, see the Test Outsourcing thumbnail.

Business-Facing Tests

If functional tests are used to drive development to build the product per the requirements, how do we make sure we are building the right product? The business facing tests that critique the product fulfill this role. These tests assess the product (either as built or as proposed) for “fitness for purpose”. Usability tests are examples of tests that critique the product from a business perspective

Typically, these tests cannot be automated because they are highly subjective and some even require us to observe people trying to use the product to achieve their goals.

16.2 What Testing Will We Do? And Why?

Now that we have been introduced to a way of reasoning about the kinds of tests, we can decide the types of tests we need to run and which tests to automate. This is an overall test strategy that helps us determine how to best address our testing needs at the lowest cost.

16.2.1 Test Strategy

Defining the test strategy may be considered to be part of the test planning process or a distinct activity. Either way, the purpose of defining a test strategy is to make some high-level decisions about what kinds of testing need to be done and how they will be executed. One of the key decisions is what kinds of tests should be automated and which approach to testing should be used for manual tests. The goal of these decisions is to try to minimize project risk while also minimizing the time and effort spent testing the software.

Previous sections introduced the concepts of functional and para-functional requirements. As part of the test strategy we need to decide where to focus. Testing cannot prove that software works correctly; it can only prove that it does not work correctly. Therefore, we could spend an infinite amount of time testing and still not prove the software is perfect. The test strategy is about maximizing the return on investment (ROI) of testing by identifying the testing activities that will mitigate the risks most effectively. This implies that some requirements may be tested less thoroughly, by choice.

We also need to decide whether we will do all the acceptance testing at the end of the project (test last acceptance; this is also known as testing phase or big bang testing) or incrementally as functionality becomes available (incremental acceptance). Incremental acceptance requires changes in how the project is planned and how the software is developed to ensure a continuous stream of functionality is delivered starting fairly early in the project. The payback is that misunderstood and missed requirements are discovered much sooner, thereby allowing time for remediation off the critical path of the project.

Another strategic decision may relate to test oracles; our source of *truth*. How will we define what a correct outcome looks like? Is there a comparable system that we can use as an oracle? (For more information, see Comparable System Test Oracle.) Can we hand-craft expected results? (For more information, see Hand-Crafted Test Oracle.) Or will we need to use a Human Test Oracle? If so, what can we do from a design-for-testability perspective to reduce the dependency on human test oracles?

16.2.2 Manual Testing Freedom

For functional testing, the key strategy decisions relate to how we will execute the tests. When we manually execute the tests, we need to decide how much freedom to grant the testers. Figure 2 contains the "freedom" scale used by Jon and James Bach to describe the choices we have.

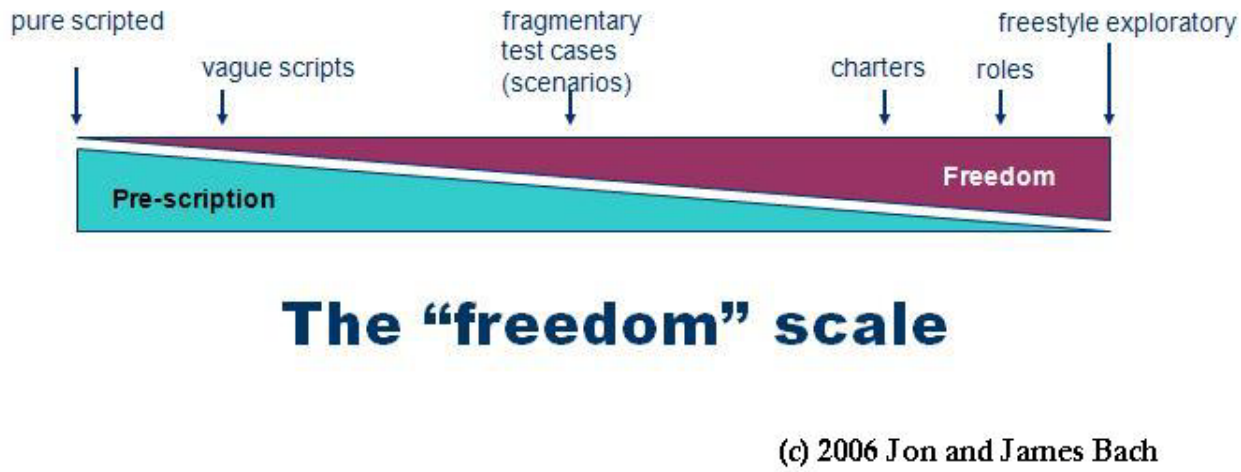


Figure 2
"Freedom" scale for testers

At one extreme of the test freedom scale, there is freestyle exploratory testing, in which the testers can test whatever they think is important. At the other end of the scale is scripted testing, in which testers attempt to follow a well-defined test script. In between, there is chartered exploratory testing, which has charters of varying degrees of freedom including scenarios, user roles/personas, and charters. Scripted testing involves having an expert prepare detailed test scripts to be executed much later by someone else (or a computer when automated). There is very little opportunity for test design during test execution. Exploratory testing is a powerful approach to testing that leverages the intelligence of the tester to maximize the bugs found in a fixed amount of time. Unlike with scripted testing, testers are encouraged to conceive new things to try while they are executing tests. Some people describe it as ”concurrent test case design and execution with an emphasis on learning”. For more information, see [Kaner et al] and [Bach].

16.2.3 Automated Testing

Automated testing covers a wide range of topics. Automated execution of functional tests is one. Some kinds of para-functional tests require automated execution because of the nature of the testing being performed. A commonly overlooked area for automation is the use of "power tools" while performing manual testing. Tools can also be used to generate test data. The various uses of test automation need to be determined on a project-by-project basis. For more information about this process, see the Planning Test Automation thumbnail of this guide and [Fewster & Graham].

Maximizing Automation ROI

An effective test automation strategy strives to maximize the ROI of the investment in automation. Therefore, the tests we automate should cost less, at least in the long run, than we would have spent manually executing the comparable tests. Some tests are so expensive to automate that we will never recoup the investment. These tests should be run manually.

Automating the Right Tests

To ensure that we get the best possible ROI for our test automation investment, we need to focus our energies on the following:

- Tests that have to be automated by their very nature
- Tests that are inherently easier to execute using a computer than a human
- Tests that need to be run many times
- Tasks (not tests) that can make manual (or automated) testing faster and more effective

16.2.4 Automated Execution of Functional Tests

Automated functional test execution is a powerful way to get rapid feedback on the quality of the software we produce. When used correctly, it can actually prevent defects from being built into the product; when used incorrectly, it can rapidly turn into a black hole into which time and effort are

sucked. When automated regression tests are run frequently, such as before every code check-in, they can prevent new defects from being inserted into the product during enhancement or maintenance activities. Providing the supplier with automated acceptance tests ahead of time can ensure the supplier builds the correct product the first time instead of as a result of test and fix cycles. For information about how this works, see the Acceptance Test Driven Development thumbnail.

A common strategy on projects that have an extensive suite of automated tests is to run these tests first as a form of regression test as the first activity in a test cycle; this is a form of extended smoke test. This ensures that the software functions properly (to the extent of the automated test coverage) before a human tester spends any time doing manual testing.

The key to effective automated functional testing is to use an appropriate tool for each type of test—one size does not fit all. The two most common approaches to automated test preparation are test recording (see the Recorded Test thumbnail) and test scripting. Recorded tests are easy to produce, but they are often hard to maintain. Scripted tests can either be programmatic test automation, which involves technical people writing code to test the code, or keyword-driven test automation, which non-technical people can use to write tests using a much more constrained testing vocabulary. Because keyword-driven tests are typically written in the ubiquitous language defined for the product, they are also much easier to understand than most programmatic tests. Whatever approach we use, we should think beyond the initial test authoring and also consider the life cycle costs of the tests. Recorded test tools do have some valuable uses. They can be used to quickly record throwaway test suites to support the development team while they refactor testability into the system under test. They can also be used in a record and refactor style as a way of quickly building up a collection of keywords or test utility methods to be used in keyword-driven tests or programmatic tests, respectively.

Keyword-driven testing involves specifying test scripts in a non-programmatic style. The steps of the test are data interpreted by a keyword language interpreter. Another style of data-driven test automation is the reuse of a test script with multiple data sets. This is particularly effective when we can generate test data, including inputs and expected outputs, using a comparable system test oracle. Then we run the data-driven test one time for each set of inputs/outputs. Commercial recorded test tools typically provide support for this style of testing and often include minimal support for refactoring of the recorded test scripts into parameterized scripts by replacing the constant values from the recorded test with variables or placeholders to be replaced by values from the data file.

Test Automation Pyramid

The test automation pyramid is a good way to visualize the impact of different approaches to test automation. When test automation is an afterthought, the best we can usually do is to use graphical user interface (GUI)–based test automation tools to drive the system under test. This results in a distribution of tests as shown in Figure X – Inverted Test Pyramid

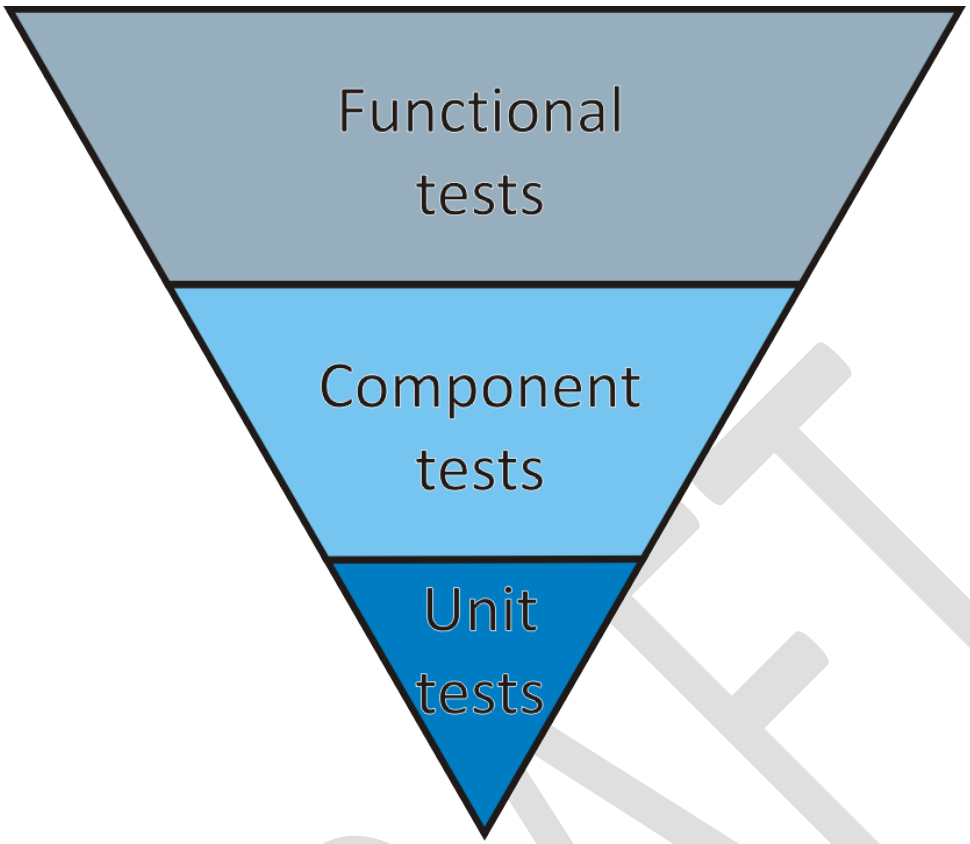


Figure X

Inverted Test Pyramid

Frequently, these functional tests are very difficult to automate and very sensitive to any changes in the application. Because they run through the GUI, they also tend to take a long time to execute. So if test automation is an afterthought, we end up with a large number of slow, fragile tests.

An important principle when automating tests is to use the simplest possible interface to access the logic we want to verify. Agile projects that use test-driven development techniques attack this problem at multiple levels. They do detailed unit testing of individual methods and classes. They do automated testing of larger-grained components to verify that the individual units were integrated properly. They augment this with use case or workflow tests at the system level. At each higher level, they try to focus on testing those things that could not be tested at the lower levels. This leaves them with much fewer use case and functional tests to automate. This is illustrated in Figure X – Proper Test Pyramid

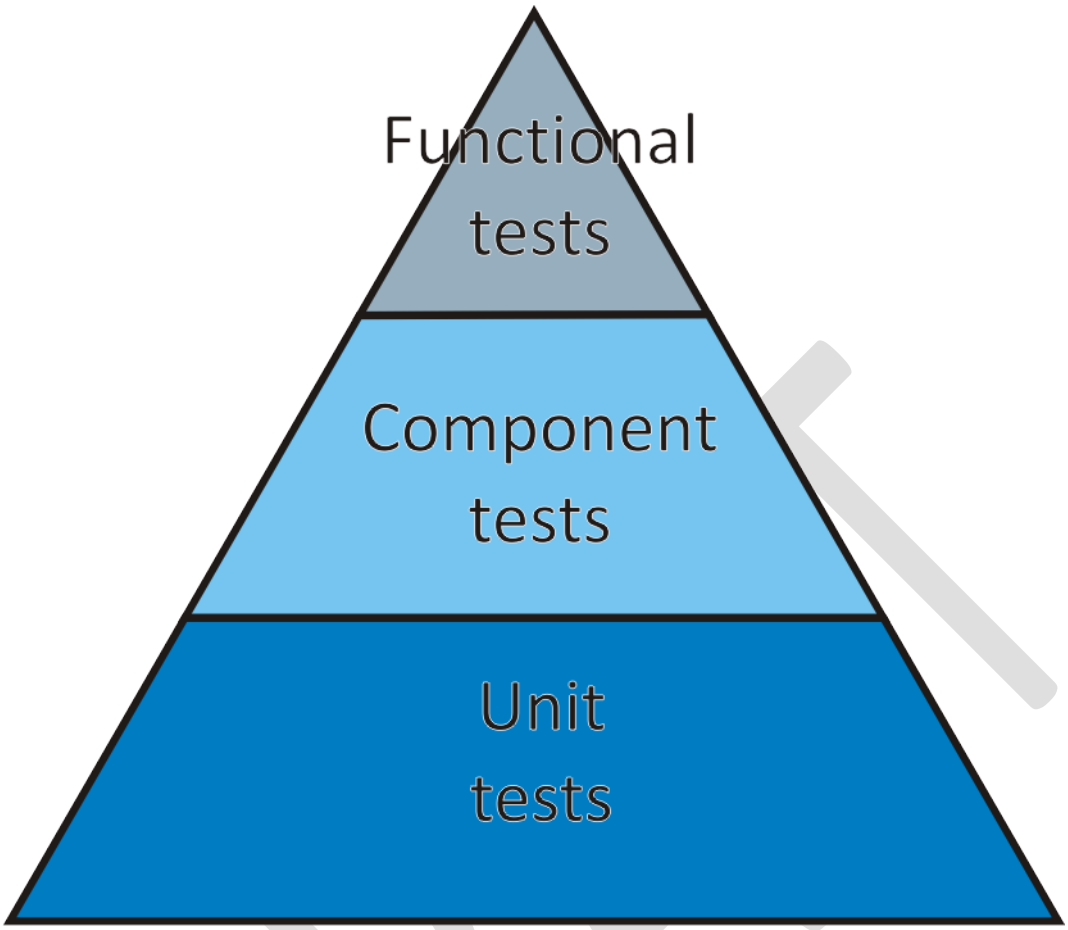


Figure X

Proper Test Pyramid

Agile projects are always looking for ways to reduce the effort involved. One way they achieve this is to minimize the overlap between the unit tests and the functional tests. A specific example of this is the use of business unit tests to test business logic without having to go through the user interface. Another technique is the use of subcutaneous workflow tests to test business workflows without being forced to access the functionality through the user interface. Both of these approaches require the system to be designed for testability.

16.2.5 Automated Testing of Para-functional Requirements

Many types of para-functional tests require the use of automated test tools. Many of these tools are specially crafted for the specific purpose of assessing the system with respect to a particular kind of para-functional requirement. Common examples include performance testing tools that generate load to see how the system copes with high transaction rates.

16.2.6 Automation as Power Tools for Manual Testers

Automated tests provide a high degree of repeatability. This works very effectively as a change detector, but it probably will not find bugs that have always been there. For that we need human testers who are continually looking for ways to break the software. For human testers to be effective, they must be able to focus on the creative task of dreaming up and executing new test scenarios, not the mundane tasks of setting up test environments, comparing output files, or generating or cleansing large amounts of test data. A lot of these tasks can be made fairly painless through appropriate use of automation.

We can use automated scripts for the following:

- To set up test environments
- To generate test data
- To compare actual output files or databases with test oracles
- To tear down test environments

16.2.7 Automated Test Generation

One of the grand objectives of software testing is automated test generation. The industry is still some distance from being able to push one button and have a tool generate and run all the tests we will ever need, but there are some selected situations where automated test generation is practical. One example is combinatorial test optimization. For example, if we have a module we are testing that takes five different parameters, each of which could be any one of four values, each of the values causes the module to behave somewhat differently, but in different way. To test this effectively, we would have to test 1024 (4*4*4*4*4) different combinations, which is not very practical. We can use a tool, such as AllPairs, that analyses the five dimensions and generates a minimal set of five-value tuples that will verify each interaction of a particular pair of values at least once.

If we need a large dataset, we can write a program to generate one with known characteristics. If we need to test how particular transactions behave when the system is stressed, we can write a program that uses up all the memory or disk-space or CPU on command. These are all examples of power tools that make human testers more effective.

16.2.8 Readiness vs Acceptance

As described Chapter 1 - Tthe Acceptance Process and Chapter 2 - Decision Making Model in Part I – Thinking about Acceptance , the acceptance of software can be divided, at least logically, into two separate decisions. The readiness decision is made by the supplier organization before giving the software to the customer who makes the acceptance decision. A key decision is determining which tests are run as part of readiness assessment and which are run as part of acceptance testing. In most cases, the readiness assessment is much more extensive than the acceptance testing. When functional tests are automated, it is likely they run in readiness assessment, and the software is not released to

acceptance testing until all the tests pass. This results in a better quality product being presented to the customer for acceptance testing.

16.3 Who Will Accept the System?

Ultimately, the acceptance decision belongs to the customer. In some cases, the customer may not be a single person. In these cases, we may have a customer team or committee that makes the acceptance decision using some sort of democratic or consensus-based process. Or, we may have a set of acceptance decision makers that each can veto acceptance (see Chapter 1 - The Acceptance Process in Part I) In other cases, the customer may be unavailable. In these cases, we may need a customer proxy to act as the "goal donor" who both provides requirements and makes the acceptance decision. The proxy may be either a delegate selected by the single customer, a mediator between a group of customers, or a surrogate who acts on behalf of a large group of anonymous customers. The latter role is often referred to as the product manager. For more information about this process, see the Customer Proxy Selection thumbnail.

A related question is who will do the acceptance testing? And by extension, who will do the readiness assessment. This very much depends on the business model and the capabilities and skill set of the parties involved. The sidebar "Decision-Making Model Stereotypes" enumerates several common scenarios. When either the supplier or the customer feels they need assistance conducting the readiness assessment or acceptance testing, they may resort to a Test Outsourcing model. The third-party test lab would do the assessment, but the readiness decision or acceptance decision still belongs to the supplier and customer.

16.4 When Will We Do the Testing?

The test plan needs to address when the testing will be done. Some testing activities will be done by the supplier as part of readiness assessment, while others are the responsibility of the customer who will be deciding whether to accept the product. The test plan needs to include this in further detail to the point where we have an understanding of how much time we need for readiness assessment and acceptance testing and what we will use that time for.

One way to plan testing is to define a testing phase of the project after all new functionality development is complete. This testing phase would consist of several time-boxed test cycles, each of which contains both readiness assessment and acceptance testing activities as illustrated in Figure A.

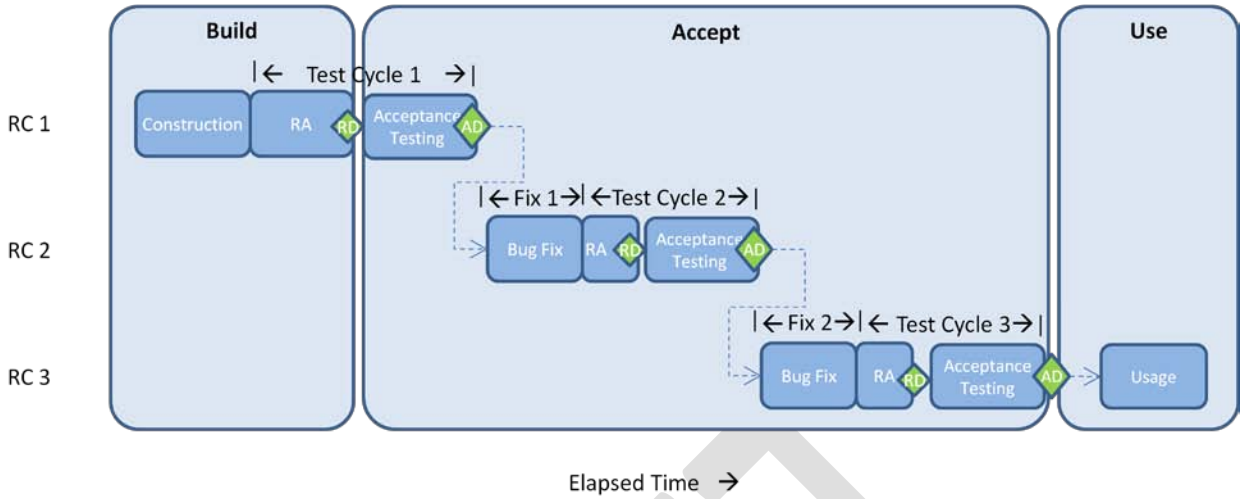


Figure A – Multiple Test Cycles in Accept Phase of Project

Each test cycle would be focused on one release candidate (RC) version of the software. Within the test cycle we make a readiness decision before involving the customer in the acceptance testing. In the early test cycles, this readiness decision is made by answering the question, "Is it good enough to bother having the customer test it?" It is valuable to get customer feedback on the software even if we know there are some defects. The test cycles each result in concerns that need to be investigated. Any concerns that need software changes are then addressed by the supplier, and a new release candidate is built. This sets the stage for the next test cycle. We repeat this process until the release candidate is accepted by the acceptance decision maker(s).

Within each test cycle, it is likely we will have a predefined set of testing activities; these may be laid out as a Pert chart or a Gantt chart to reflect timing and interdependencies. We may decide to reuse the same plan for each of the test cycles, or we could define a unique plan for each cycle. In practice, with good automated regression testing in place, we should find fewer and fewer defects each test cycle. Because of this, a risk-based approach to planning the subsequent cycles could result in shorter cycle times and faster time to market. We may also deliberately choose to defer some kinds of testing activities to later test cycles or to do them in earlier test cycles.

An alternative to using a plan-driven approach within the test cycle is to use a more iterative style known as Session-Based Test Management. We create a prioritized backlog of testing activities that we address in a series of test sessions. As new concerns are identified in test sessions, we may add additional test activities to the test backlog. The key is to keep the backlog prioritized by the value of the testing. This value is typically based on the expected degree of risk reduction. The depth of the backlog gives us an idea of how much testing work we have left (for example, by creating a testing burn-down chart) and if we are making headway by addressing concerns or if you are losing ground (if this is the case, the backlog is increasing in depth). Session-Based Test Management is commonly used with exploratory testing.

16.4.1 Where Will We Do the Testing?

The test plan needs to identify where the testing will be performed. When all the testing will be performed in-house, the primary consideration is which physical (or virtual) environments will be used. This is particularly important when new environments need to be created or shared environments need to be booked. If we lack physical resources or the skills to do the testing, we may choose to do test outsourcing to a third-party test lab.

We also need to define the criteria for moving the software between the environments. The transition from the readiness assessment environment to the acceptance environment is governed by the readiness assessment criteria. When developers have their own individual development environments, we also need criteria for when software can be submitted into the team's integration environment where readiness assessment will occur. These criteria are often referred to as the Done-Done Checklist because the definition of "done" is more stringent than what a developer typically refers to as done.

16.5 How Long Will the Testing Take?

Because testing is usually on the critical path to delivery of software-intensive-systems, project sponsors and project managers usually want to know how long testing will take. The most common answer is "How much time do we have?" Frequently, the time available is not long enough to gather enough data to make a high confidence readiness or acceptance decision. This answer is not quite as flippant as it sounds because of the nature of testing. We cannot prove it works correctly—we can only disprove it by finding bugs. Even if we have an infinite amount of time to test, we probably will not find many more defects than we would in half an infinity. So, have to determine what is barely sufficient to get enough confidence that we understand the quality level. This requires at least a minimal level of test estimation to establish the lower bound of the time and effort we need to expend.

We also need to know how long we'll need to wait for any defects we need addressed to be fixed by the supplier. Will there be "dead" time between test cycles while we wait for fixed software as illustrated in Figure Y – Alternating Test & Fix?

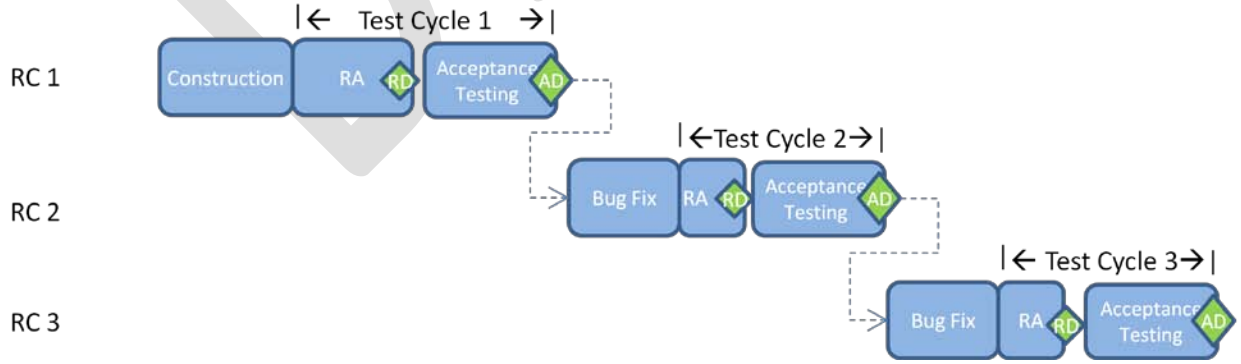


Figure Y – Alternating Test & Fix

The dead periods occur if the acceptance decision is the first point at which the supplier is provided the list of "must fix" bugs after which the supplier starts the process of characterizing and fixing the bugs. This may take days or weeks to develop a new release candidate which must then undergo readiness assessment before being presented to the customer for the next round of acceptance testing.

The duration of the acceptance phase (consisting of several test cycles) can be reduced significantly if the supplier is notified of bugs as soon as they are found. This allows them to be fixing defects continuously and delivering a new release candidate on a regular schedule as shown in Figure X – Continuous Test & Fix.

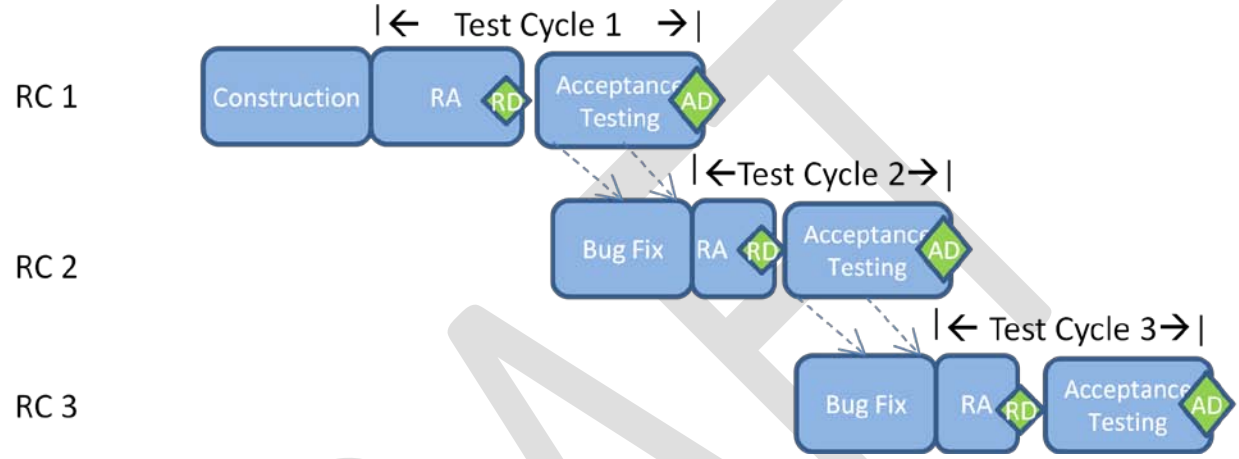


Figure X – Continuous Test & Fix

This depends on the customer (or proxy) doing bug triage on all newly found concerns so that the supplier is made aware of all gating bugs (bugs that must be fixed before accepting the software) as soon as practicable.

If we are planning to automate tests, we should have an effort estimate for the automation. In most cases, we want separate estimates for the construction of the automation infrastructure and the preparation of the tests because of the different skills and knowledge needed to do the two jobs. For more information, see the Test Automation Planning thumbnail.

16.6 How Will We Determine Our Test Effectiveness?

A learning organization is one that is constantly striving to improve how it works. This involves understanding how well we are doing today and trying new approaches to see whether they make us more effective. Measuring effectiveness requires Test Metrics. These metrics measure two key areas of performance:

- They measure how far along are we in executing our test plan. That is, how much work is left before we know enough to make the readiness or acceptance decision? For more information, see the Test Status Reporting thumbnail.

- They measure the effectiveness of our testing. For more information, see the Assessing Test Effectiveness thumbnail.

16.7 How Will We Manage Concerns?

The purpose of testing is to identify any concerns with the software. Many of these concerns will require changes to the software either because something was incorrectly implemented (a bug) or because the customer realized that what they had requested will not satisfy the business need (an enhancement or change request). The test plan needs to lay out how these concerns will be managed and tracked; it also needs to include the process for deciding what needs to be changed and what is acceptable as is.

As we conduct the various readiness assessment and acceptance activities, we note any concerns that come up. Investigating these concerns more closely reveals that, each concern falls into one of the following categories:

- Bug or defect
- Requirements change
- Project issue
- Non-concern

Figure 3 illustrates the lifecycle for each of the major types of concerns.

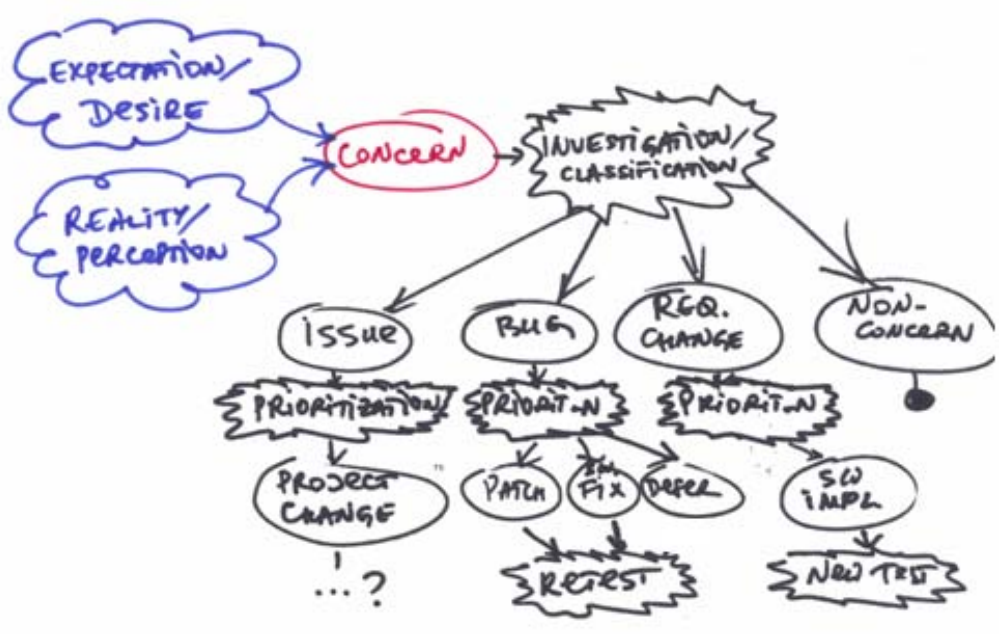


Figure 3
Concern Resolution Model

Part of the advantage of categorizing these concerns is to help identify how they will be addressed. Each category of concerns has its own resolution process and each must be addressed differently.

16.7.1 Bugs or Defects

Bugs are problems found in the software that require a software change. The bugs need to be understood well enough to make decisions about what to do about them. A common process for doing this is known as Bug Triage, which divides the bugs into three categories with respect to the next milestone or release: Must Fix, Would Like to Fix (if we have time), and Will Not Fix. Of course, the software must be retested after the fixes are made, which is why there are typically multiple test cycles. Fixes may also need to be propagated into other branches of the software. If the bug is found to exist in previous versions that are currently in use, a patch may need to be prepared if the bug is serious enough. Security-related bugs are an example of bugs that are typically patched back into all previous versions still in use. Likewise, a bug fixed in the acceptance test branch may need to be propagated into the current development branch if new development has already started in a separate development branch.

16.7.2 Requirements Changes

The customer may have realized that even though the supplier delivered what the customer asked for, it will not provide the expected value. The customer should have the right and responsibility for making the business decision about whether to delay the release to make the change or continue with the less useful functionality. Once we've decided to include a change in this release we can treat it more or less the same as a bug from a tracking and retest perspective.

16.7.3 Other Issues

Some concerns that are exposed do not require changes to the software. They may be project issues that need to be tracked to resolution, additional things that should be tested, and so on. These typically do not get tracked in the bug management system because most projects have other means for tracking them. Other concerns may be noted but deemed to not be concerns at all.

16.8 Summary

This chapter introduced the activities and practices involved in planning a testing effort. A key activity is the definition of a test strategy because this is what guides us as we strive to maximize the ROI of our efforts. There is a place for both automated testing and manual testing on most projects because the two approaches are complementary. Automated functional tests are highly effective change detectors that go a long way toward preventing new bugs from being introduced during software maintenance activities. Care has to be taken to use the appropriate functional test automation tools to avoid the slow, fragile tests quagmire. Manual testing, especially exploratory testing, is highly effective at finding unforeseen bugs that we may not even think might exist. The use of power tools by human testers can significantly increase the effectiveness of manual testing.

16.9 What's Next?

In this chapter we have described the practices involved in planning for the acceptance of the product. In the next chapter we will examine the practices we use while executing the acceptance process.

16.10 References

[Meszaros] Meszaros, Gerard. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Professional. 2007.

[Kaner et al] Kaner, Cem, Jack Falk, and Hung Q. Nguyen. *Testing Computer Software, 2nd Edition*. Wiley. 1999.

[Bach] Bach, James. What is Exploratory Testing? http://www.satisfice.com/articles/et-article.pdf

[Fewster & Graham] Fewster, M., Graham, D. Software Test Automation, Addison-Wesley, 1999.

Chapter 17 Assessing Software

In the previous chapters we have introduced many of the concepts around how we plan the assessment of the product against the minimum credit release (MCR) and minimum quality requirement set (MQR) to which we have agreed. In this chapter we will introduce the various techniques that we use as we do the assessment including test conception, test design and test execution.

Assessment is a generic term we can use to describe the activities, testing or otherwise, that we use to evaluate the system-under-test. Some of these activities are focussed on preparing and executing tests while others may be review activities. Some of the activities are done as part of readiness assessment by the software supplier while others may be done by or for the customer under the banner of acceptance testing. Where the same practice is used in both forms of testing, the mechanics of the practices typically don't change very much although the emphasis of how much each practice is done and the objectives of applying that practice might vary.

We start the discussion with an overview of the lifecycle of an individual tests, something that both functional and para-functional tests do share and then move into a discussion of the practices used in each state of the individual test lifecycle where we introduce the techniques that are unique to specific of kinds of requirements.

17.1 Individual Test Lifecycle

Every single test, however simple or complex, whether manual or automated, goes through a number of stages during its lifetime. This lifecycle is illustrated in Figure 1.

Figure 1
Individual Test Lifecycle

The states of the lifecycle are:

Conception	An acceptance test is conceived to address a particular risk and/or achieve a specific goal.
Authoring/Design	The test is written either in detailed step form or some kind of outline of what needs to be done, by whom, when, where, and how.

Scheduled	The execution of the test is planned or scheduled for a specific timeframe and resources (people, test environment(s), etc.)
Execution	The test is executed against the system under test.
Result Assessment	The results of the test are assessed against the expectations. (This may occur as part of execution or separately.)
Reporting	The assessed test results are aggregated and reported.
Actioning	The test results may result in either further testing being identified and/or bug reports being created and triaged.
Maintenance	Each test is an asset and most must be maintained so that it can provide value in the future. Some tests, however, e.g. when doing exploratory testing, ,are not meant to be maintained or repeated.
End-of-Life	The test has outlived its usefulness. It is abandoned and no longer run or maintained.

Note that test conception and test authoring are often lumped together under the label of test design. Some forms of testing, often called static testing, involve inspecting artefacts that describe the system under test rather than running the actual code. The terminology used for these forms of testing is somewhat different (for example. they are often called reviews rather than tests) but for the purpose of discussing test lifecycle, we shall use the test terminology.

Let's examine each of these states in a bit more detail.

17.1.1 Test Conception

At some point, someone decides that we need to verify one or more aspects of the system-under-test's behavior; we call these "things to test" *test conditions*. At this time the test is just a figment of someone's imagination. It starts its transition from an implicit requirement to one that is much more explicit when it gets written down or captured in a document. It might appear in a list of test conditions associated with a feature, requirement or user story. Typically, it will just be a phrase or name with no associated detail. Now that the test exists in concept, we can start moving it through its lifecycle.

17.1.2 Test Authoring / Test Design

Test authoring or test design is the transformation of the test or test condition from being just a named item on a list into concrete actions. It may also involve making decisions around how to organize test conditions into test cases which are the sequences of steps we execute to verify them. Note that test authoring/design may happen long before test execution or concurrently with test execution.

17.1.3 Test Scheduling

Once a test case has been identified and authored or a test charter defined, we must decide when it will be executed. The schedule may indicate the one time the test is run or the frequency with which it is run

and the triggering mechanism. It may also identify who or what is executing the test and in which test environment(s).

17.1.4 Test Execution

Once authored and scheduled, we need to actually execute the tests. For dynamic tests this involves running the system-under-test; static tests involve inspecting various artefacts that describe the system-under-test but do not involved executing the code . Depending on the kind of test in question the test may be executed manually by a person, by an automated testing tool, or by a person using tools that improve tester productivity through automation.

17.1.5 Result Assessment

Depending on the tools involved, the pass/fail status of the tests may be determined as the tests are executed or there may be a separate step to determine the test results after the test execution has been completed. We determine whether a test passed or failed by inspecting the actual results observed and determining whether they are acceptable.

17.1.6 Test Reporting

Once a suite of tests has been executed and assessed, we can report on the test results. A good test report helps all the project stakeholders understand where the project stands relative to the release gate. Chapter 1 - The Acceptance Process provides details on what information might affect this decision. Test reporting includes both test status reporting to indicate how much test effort remains and test effectiveness reporting which describes our level of confidence in our tests.

17.1.7 Test Actioning

The purpose of executing tests is to learn about the quality of our product so that we can make intelligent decisions about whether it is ready for use or requires further development or testing. The Acceptance Process describes the process for deciding whether or not to accept the software but before we can make that decision we may need to fix some of the defects we have found. The Bug Triaging process is used to make the “Is it good enough?” decision by determining which bugs need to be fixed before we can release. (See the “Doneness Model” for more details.)

17.1.8 Test Maintenance

Some tests are only run once while others may need to be run many times over long periods of time. Tests that will be run more than once may require maintenance between runs as a result of changes to the parts of the system-under-test that they interact with (for example, the database state). These kinds of tests may warrant more of an upfront investment to ensure that they are repeatable and robust. Tests intended for manual execution will need to be updated whenever the parts of the system being tested undergo significant changes in functionality. Whereas human testers can usually work around minor changes, fully automated tests will typically be impacted by even the smallest changes

(with tests that interact with the system-under-test through the GUI being the most fragile) and therefore may require significantly more frequent maintenance.

17.1.9 End of Life

Sooner or later a test may no longer be worth executing. Perhaps the functionality it verifies has been removed from the system-under-test or maybe we have determined that the functionality is covered sufficiently well by other tests so that we no longer get much additional value from running this test. At this point the test has reached its end of life and no longer warrants either execution or maintenance.

17.2 Variations in Test Lifecycle Traversal

Some tests spend a lot of time in each state of the test lifecycle while others may pass through the states very quickly. For example, in automated functional testing we might spends weeks preparing a complex test, wait several weeks before we can first execute it, and then run it several times a day for many years. In contrast, during a single one hour exploratory manual testing session, the tester may conceive of several test conditions, design a test to explore them, learn something about the system-under-test, conceive several more test conditions and design tests to explore them also, – all in the space of a few minutes. The automated tests will spend most of their lifetime in the maintenance state while exploratory tests are very ethereal; there isn't a concrete representation that needs to be maintained.

17.2.1 Highly Compressed Test Lifecycle - Exploratory Testing

Exploratory testing is summarized by Cem Kaner as "Simultaneous test design and execution with an emphasis on learning" [ref]. From this description it should be clear that there is no clear separation of the various stages of the test lifecycle when doing exploratory testing. The tester learns about the system-under-test by using it and forming hypothesis about how it should behave. Based on these hypotheses the tester conceives of one or more test conditions to which they might subject the system-under-test. They rapidly design, in their mind's eye, a test case they could use to achieve this. They exercise the test case and observe the result thereby forming more hypotheses which in turn lead to more test conditions. There is not an attempt made to have the test persist beyond the test session unless it revealed a bug. This removes the need to document and maintain the tests (though a good exploratory tester keeps a set of notes/journal of key points and discoveries during his or her testing); the obvious consequence is that exploratory testing is not intended to be very repeatable.

This process is very lightweight with very little overhead getting in the way of the tester interacting with the software. This makes it possible for exploratory testers to formulate a lot of hypotheses, test them and find a lot of bugs in a very short period of time. The tester takes notes as they go focussed primarily on the following outputs:

1. What functionality they have tested and their general impressions of what they have seen.
2. What bugs they found and what they had done to cause them.

3. What test conditions they had conceived that they were not able to get to. These may be used as the charter for a subsequent exploratory test session.
4. How much time was spent actually testing versus how much was spent getting ready. This information is useful when deciding what kind of power tools would make the exploratory tester more efficient in the future.

Despite its somewhat chaotic appearance exploratory testing can be quite disciplined and methodical even though it is not very repeatable. Exploratory testing can range from completely unstructured to highly disciplined. The more disciplined forms of exploratory testing use a sequence of time-boxed test sessions to structure the testing activities.

Planning of exploratory testing consists of defining an initial list of high-level test conditions (as in "kinds of things we should test") for use as test session charters and deciding how many test sessions to budget for executing the charters. Examples of charters might include the following:

- Pretend that you are a hacker and try breaking into the system (a persona-based charter)
- Try out variations of the invoicing workflow focussing on rejected items (a scenario-based charter)
- Try using the user interface using only the keyboard (a device-based or persona-based charter).
- Try scenarios where several users try accessing the same account at the same time. (a scenario-based charter often called "tug of war".)

The test charters are prioritized before being scheduled via assignment to a tester executing a specific test session. Upon completion of the test session, the tester may recommend additional charters be added to the backlog of charters based on what they had learned about the system-under-test, business domain, users' needs etc. Exploratory testing is often done in an iterative style with many of the test charters for later iterations being discovered during execution of the test sessions in earlier iterations. This allows exploratory testing to focus on the areas of the software that have been found to be the most suspicious rather than providing the same amount of effort for all areas regardless of the quality level actually observed. This, combined with the low overhead nature of the simultaneous test design and execution, is what allows exploratory testing to be such an effective way of finding the bugs we know must lurk in the software.

17.2.2 A Spread Out Test Lifecycle – Scripted Testing

In scripted testing, the average test lifecycle is much longer that in exploratory testing. The tests may be conceived as part of the test planning exercise or in more detailed test design activities. The actual tests are then documented or programmed, and potentially reviewed, often before the software is available for testing. The tests may require maintenance even before they are first executed against system-under-test if the design of the software has evolved since the tests were designed. Eventually, we determine the schedule for executing the tests and the tests are executed at the appropriate time (which may be weeks or even months later.) Any bugs we find are logged and the test results are

reported to the stakeholders. The bugs are actioned, often weeks or even months after they were found. If the tests are to be repeated at a later time, usually against a subsequent version of the system-under-test, the tests may require maintenance to track changes in the system they test. Eventually, someone decides that this particular test is no longer adding any value and the test is abandoned.

This cycle could take anywhere for several days or week to many years. The test team for Microsoft Office prepares extensive automated test scripts to verify the behavior of features in applications like MS Word. The tests for a specific generation of the product (e.g. Word 2003) have a lifetime of over 10 years because of Microsoft's commitment of 5 years of mainstream support and a further 5 years of limited extended support for each business and development software product, followed by a minimum of 1 year of self-help online support via the knowledge base [Ref - http://support.microsoft.com/lifecycle/]. New builds are typically created every week through this product lifetime and the tests are run against each new build. In this example the maintenance phase of the test dominates the individual test lifecycle. Microsoft patterns and practices team use continuous integration that potentially produces several builds a day with continuous automated test execution.

17.2.3 Intermediate Test Lifecycles – Hybrid Test Approaches

The two previous sections described the two extremes of test lifecycle duration. In practice, the test lifecycles can fit anywhere between these two extremes. There could also be a mix of test lifecycle durations even in the same testing session. For example, a manual tester could be following a detailed test script that was written months ago. They notice something odd that isn't specifically related to the script and decide to go "off-script" to explore the oddity. In this off-script excursion they are doing simultaneous test design and execution (in other words, exploratory testing). At some point they may return to the original script after either confirming that the system is working properly or logging the bugs that they have found.

17.3 Practices for Assessing Software

In Chapter 16 – Planning for Acceptance we introduced many practices in the context of planning the readiness assessment and acceptance testing activities. Now it is time to look at the practices that we use while designing, executing and actioning the individual tests. At this point we focus on the practices and not on who does them; it really doesn't matter whether they are done as part of readiness assessment or acceptance testing as the practices themselves are not changed by when and who does them. Volume II contains a collection of thumbnails and job aides for each practice described here with Volume III presenting sample artefacts of applying those practices in a project.

17.3.1 Test Conception Practices

There are quite a few practices for conceiving tests or test conditions. Some are more structured or formal than others. They all share the goal of creating an extensive to-do list for our subsequent test design and execution efforts. Most start with either requirements, whether functional or para-

functional, or risks (concerns about something that might go wrong.) Some of the more common techniques include the following:

- Risk-based test identification
- Heuristics or checklists
- Use case based testing – Define tests based on specific use cases of the system-under-test (see the Functional Testing thumbnail.)
- Business rule testing –Define tests for various combinations of values used as inputs to business rules or business algorithms.
- Interface-based testing – Define tests based on the characteristics of the user interface (human-computer interface) or computer-computer interface protocol.
- Scenario-based testing – using real-world usage scenarios to inspire the design of test cases.
- Soap-opera testing – Using exaggerated real-world usage scenarios to inspire the design of test cases.
- Model-based test generation – Building one or more models of key characteristics of the system-under-test and generating tests from the model(s).
- Group Brainstorming.
- Paired/collaborative testing – Working together to design better test cases.

Risk-Based Test Identification

In risk-based test identification we do risk modeling to identify areas of functionality or para-functionality that we are concerned might not be implemented correctly or might have been adversely affected by changes to the functionality. We use this information to identify test conditions we want to ensure are verified by tests. A good example of a kind of test that might be identified through risk analysis is the fault insertion test. For example, the risk we identified was “The network connection fails.” We ask ourselves “Why would the network connection fail?” and come up with 3 different possible causes: Unplugged cable, network card failure, network card out of service due to maintenance activity in progress. These are three test conditions we would want to exercise against our system-under-test. Other forms of risk might relate to potential mistakes during software development. E.g. “This shipping charge algorithm is very complex.” This might cause us to define a large number of test conditions to verify various aspects of the algorithm based on the kinds of mistakes a developer would be likely to make. For example applying the various surcharges in the wrong order.

Ensuring Secure Software

Another form of risk modeling is threat modeling. The potential threats identified by the threat model can lead us to choose from a set of security assurance practices. We might define specific penetration test scenarios to ensure the software repels attempts at penetration. We might conduct security reviews of the code base to ensure safe coding practices have been followed. We can decide to do Fuzz

Testing to verify that the software cannot be compromised by injecting specially chosen data values via user input fields.

Use Case Based Test Identification

When we have business requirements defined in the form of use cases we can identify test conditions by enumerating all the possible paths through the use case and determining for each path what input value(s) would cause that path to be executed. Each path constitutes at least one test condition depending on how many distinct combinations of inputs should cause that path to be executed.

Interface-Based Test Identification

Another source of test conditions is the design of the interface though which the use case is exercised whether it be a user interface used by a human or an application programming interface (API) or messaging protocol (such as a web service) used by a computer. The interface may have very detailed design intricacies in addition to the elements required to exercise the use case. These intricacies are a rich source of test conditions. For example, a user interface may have pulldown lists for some input fields. Test conditions for these pull-down lists would include cases where there are no valid entries (empty list), a single valid entry (list of 1 item) and many valid entries (long lists of items.) Each of these test conditions warrants verification.

Business Rules and Algorithms

Business rules and business algorithms are another rich source for test conditions. For a rule that validates a user's inputs we should identify a test condition for each kind of input that should be rejected. Rules that describe how the system makes decisions about what to do should result in at least one test condition for each possible outcome. Rules that describe how calculations should be done should result in at least one test condition for each form of calculation. For example, when calculating a graduated shipping charge with three different results based on the value of the shipment, we would identify at least one test condition for each of the graduated values.

Scenario-Based Test Identification

Scenario-based testing is the use of real-life scenarios to derive tests. There are various kinds of scenarios. The scenario-based testing thumbnail describes a wide range of scenario types this introduction touches on only a few of them. A common type of scenario-based test is the business workflow test. These tests exercise the system-under-test by identifying common and not-so-common end-to-end sequences of actions by the various users of the system as a particular work item is passed from person to person as it progresses towards successful completion or rejection. We can examine the workflow definitions looking for points in the workflow where decisions are made, and define sufficient workflow scenarios to ensure that each path out of each decision is covered.

A particularly interesting form of scenario test is the soap opera test in which the tester dreams up a particularly torturous scenario that takes the system-under-test from extreme situation to extreme situation. The name comes from its similarity to a soap opera television program which condenses many

days, months and potentially years of extraordinary events in peoples' lives into short melodramatic episodes. This form of test identification is good for thinking outside the box.

Scenarios about how software is installed by a purchaser could lead to identification of potential compatibility issues and the need for compatibility testing.

Model-Based Test Generation

In model-based test generation we build a domain or environmental model of the system-under-test's desired behaviour (expressed in mathematical terms or in some other abstract notation) and use it to generate all the relevant test cases. For example, when testing a function that takes 4 parameters each with 3 possible values, we could generate 81 test conditions (3*3*3*3) by iterating through each value for each parameter. For large and complex systems, the number of such cases will be huge. Various "guiding" or selection techniques are used to reduce the test case space. To fully define the expected result, we would need to have an independent way to determine the expected return value, perhaps a Comparable System Test Oracle or a Heuristic Test Oracle. Generation of the tests from the model may be fully automated or manual.

Identifying Para-Functional Tests

The test cases used to assess compliance to the para-functional requirements can be identified in much the same ways as those for functionality requirements with the main difference being how the requirements are discovered and enumerated.

Other Test Identification Practices

All of these practices can be applied by a single person working alone at their desk. But a single person can be biased or blind to certain kinds of test conditions; therefore, it is beneficial to involve several people in any test identification activities. One way to do this is to review the test conditions with at least one other person, a form of test review. Another practice is paired testing in which two people work together to identify the test conditions (or to write or execute tests.) We can also use group techniques like listing, brainstorming or card storming to involve larger groups of people.[ref3]

All these practices can benefit from the judicious use of checklists and heuristics. These can be used to trigger thought processes that can identify additional tests or entire categories of tests. They are also useful when doing exploratory testing.

17.3.2 Test Authoring / Test Design Practices

Now that we have of list of things we want to test, our to-do list, we can get down to designing the test cases. A key test design decision is whether we will prepare a separate test case for each test condition or address many test conditions in a single test case. There is no single best way as it depends very much on how the tests will be executed. When human testers will be executing the test cases it makes a lot of sense to avoid excessive test environment setup overhead by testing many test conditions in a single test case. The human tester has the intelligence to analyse the impact of a failed test step and decide whether to continue executing the test script, abandon test execution or to work around the failed step.

Automated tests are rarely this intelligent therefore it is advisable to test fewer test conditions per test case. In the most extreme case, typical when automating unit tests, each test case includes a single test condition.

Tests as Assets

Tests are assets (not liabilities) that need to be protected from loss or corruption. They should be managed with the same level of care and discipline that is used for managing the product code base. This means they should be stored in a version-controlled repository such as a source code management (SCM) system. The line up of tests that correspond to a particular line up of product code needs to be labelled or tagged in the same way the product code is labelled so that the tests the correspond to a specific product code build can be retrieved easily.

Tests as Documentation

Regardless of whether a test case will be executed by a person or a computer, the test case should be written in such a way that a person can understand it easily. This becomes critical for automated tests when the tests need to be maintained either because they have failed or because we are changing the expected behavior of the system-under-test and we need to modify all the tests for the changed functionality.

Many of the practices used for identifying test conditions carry through to test case design but the emphasis changes to enumerating the steps of the test case and determining what the expected outcome should be for each test. For use case tests we must enumerate the user actions that cause the particular path to be exercised. We also need to include steps to verify any observable outcomes as we execute the steps and a way of assessing whether the outcome matches our expectations. (See the section on Result Assessment Practices in Chapter 17 - Assessing Software.")

Picking an Appropriate Level of Detail for Test Scripts

As we define the steps of our tests it is very important to ensure that the level of detail is appropriate for the kind of test we are writing. For example, in a business workflow test, each step of the test case should correspond to an entire use case. If we were to use the same test vocabulary and level of detail as in a use case test, our workflow tests would become exceedingly long and readers of the test would have a hard time understanding the test. This is a classic example of "not being able to see the forest for the trees." Soap opera tests are written much like a workflow test except that the steps and circumstance are more extreme. Again, each step in the test should correspond to an entire use case.

Business Rule Testing

Business rules tests can be designed much like use case tests by interacting with the system under test via the user interface. If there are a lot of test conditions, it may make testing proceed much more quickly if we use a data-driven test approach wherein we enumerate each test condition as a row in a table where each column represents one of the inputs or outputs. Then we can simply write a parameterized test case that reads the rows from the table one at a time and exercises the system-under-test with those values. An even more effective approach requires more co-operation with the

supplier because it involves interfacing the test case that reads the rows of the table directly to the internal component that implements the business rule (we call them “subcutaneous tests”). This approach results in automated tests that execute much faster than tests that exercise the software through the user interface. The tests can usually be run much earlier in the design phase of the project because they don’t even require the user interface to exist. These tests are much more robust because changes to the user interface don’t affect them. These tests are particularly well suited to test-driven development.

Model-Based Testing

A more sophisticated way of using models is to generate executable test cases that include input values, sequencing of calls and oracle information to check the results. In order to do that, the model must describe the expected behaviour of the system-under-test. Model building is complex but is the key. Once the model is built, a tool (based on some method or notation) is typically used to generate abstract test cases, which later get transformed into executable test scripts. Many such tools allow the tester to guide the test generation process to control the number of test cases produced or to focus on specific areas of the model.

Usability Testing

The design of usability testing requires an understanding of the goals of users who will be using the system-under- test as well as the goals of the usability testing itself and the practices to be used. The goals of testing will change from test session to test session and the practices will evolve as the project progresses. Early rounds of usability testing may be focused on getting the overall design right and will involve paper prototypes, storyboards, and “Wizard of Oz” testing. Later rounds of testing are more likely to involve testing real software with the purpose being to fine tune the details of the user interface and user interaction. All of these tests, however, should be based on the usage models defined as part of User Modeling and Product Design practices.

Operational Testing

Functional requirements tend to focus on the needs of the end users but there are other stakeholders who have requirements for the system. The needs of the operations department, the people who support the software as it is being used, need to be verified as part of the acceptance process. The specific needs may vary from case to case but common forms of operational acceptance testing include:

- Testing of installers, uninstallers and software updates.
- Testing of batch jobs for initial data loading
- Testing of data reformatting for updated software.
- Testing of data repair functionality.

- Testing of start up scripts and shutdown scripts.
- Testing of integration with system monitoring frameworks.
- Testing of administrative functionality such as user management.
- Testing documentation content.

Reducing the Number of Tests Needed

If we have too many test conditions to be able test each one, we can use various reduction techniques to reduce the number of test conditions we must verify. When we have a large set of possible inputs to verify, we can reduce the number of test cases we need to execute by grouping the input values into equivalence classes based on the expected behavior of the system-under-test. That is, an equivalence class includes all the inputs for which the system-under-test should exhibit the same or equivalent behavior (including both end state and outputs.) We then select a few representative input values from each equivalence class for use in our tests. When the values are linear we pick the values right at the boundaries between the different behaviors, a technique known as boundary values analysis (BVA). When they are not linear, we can design a single data-driven test for each equivalence class and run the test for each input value in the class.

If we have several inputs that can each vary and we suspect that the behavior of the system based on the individual inputs is not independent, we avoid testing all combinations of input values by using combinatorial test optimization to reduce the number of distinct combinations we test. Examples include:

- Many independent variations or exception paths in a use case.
- Many different paths through a state model.
- Algorithms that take many independent input values that each affect the expected outcome.
- Many system configurations that should all behave the same way.

The most common variation of combinatorial test optimization is known as Pairwise or All-Pairs testing ; it involves picking the smallest set of values that ensure that every input value is paired with every other value at least once. This technique is used so frequently that there is a website[ref1] dedicated to listing the many open source and commercial programs that exist to help us pick the values to use. It has been shown that pairwise testing is more provides better coverage and few tests than random testing [Ref]

17.3.3 Test Scheduling Practices

The overall schedule that defines which tests will be run when should have already been defined as part of the test plan. In this section we'll focus on the techniques we use for detailed scheduling of test execution. As a reminder, under test execution we include both dynamic testing (in which test cases are run against the actual system-uder-test) and static testing that is performed in the form of reviews or inspections without actually executing the system-under-test.

The traditional approach to test scheduling involves defining a period of time, sometimes called a test cycle, in which one complete round of testing can be completed. The test plan would define how many test cycles are planned. The details of what happens in each test cycle often emerge as the project unfolds. One approach is to define a detailed project plan for the first test cycle, one which defines the set of testing activities that will be done on a day by day basis. Project management tools such as Pert or Gantt charts may be used to document this detailed test execution plan. (See Plan-Driven Test Management.)

Another approach is to schedule a series of test sessions and create a backlog (prioritized list) of test session charters that are executed in these test sessions. Each session would be of a standard duration, say 90 minutes, and most charters should be executable within one session. This Session-Based Test Management is most typically used for managing exploratory testing but could also be used for execution of scripted testing.

A third approach is often used by agile or self-organizing teams. All the testing tasks are posted on a wall chart, variously called a "big visible chart" [Ref, RJ] or an "information radiator"[Ref, ACA], and team members sign up to do specific testing tasks. As they finish one task, they mark it done and pick another task to perform. This is called Self-Organized Test Scheduling.

Some forms of automated testing can be set up to run automatically at certain times (such as every night at 2am) or when certain events occur (such as when someone checks in a change to part of the code base.) The results can be posted on a web site, automatically e-mailed to all team members, or communicated by a multi-colored light display or "lava lamps" in the team workspace or icon in everyone's computer system tray. This practice is often called Continuous Integration or CI for short.

17.3.4 Test Execution Practices

The details of how the tests are executed vary greatly across the different kinds of tests but several things are common. A key outcome of test execution is the data that will be used as input to the subsequent readiness and acceptance decisions. Though various project contexts will require much more extensive record-keeping than others, it is reasonable to expect a minimum level of record keeping. This record-keeping consists of the following key pieces of information:

- What tests have been run, by whom, when and where?
- What results were observed?
- How did they compare to what was expected?
- What bugs have been found and logged?

The comparison with expectations is described in the section Result Assessment Practices.

Functional Test Execution Practices

The nature of a test determines how we execute it. Dynamic tests involve running the system-under-test while static tests involve inspecting various artefacts that describe the system. Dynamic tests typically fall into one of the following categories:

- Automated Functional Test Execution – This involves using computer programs to run the tests without any human intervention. The test automation tool sets up the test environment, runs the tests, assesses the results and reports them. It may even include logging of any bugs detected. The tests may be started by a human or by an automated test scheduler or started automatically when certain conditions are satisfied such as changes to the code base.
- Manual test execution – This involves a person executing the test. The person may do all steps manually or may use some automation as power tools to make the testing go faster. The human may adjust the nature of the test as they execute it based on observed behavior or they may execute the steps of a test case exactly as described.
- Exploratory test execution – This is a form of manual test execution that gives the tester much more discretion regarding exactly what steps to carry out while testing. Each test session is usually scoped using a test charter.

Para-functional Test Execution Practices

Para-functional tests are somewhat different from functional tests in several ways. As intimated by their name, para-funcational tests span specific functions of the system. Static para-functional tests may involve running tools that analyse the software in question or they many involve reviewers who look at the code or other artefacts to find potential design or coding defects. Dynamic para-functional tests may involve running tools that interact with the system under test to determine its behavior in various circumstances.

Static Para-functional Test Practices

Static analysis is done by examining the code or higher level models of the system to understand certain characteristics. Specific forms of static analysis include the following:

A Design/Architecture Review is used to examine higher-level models of the system to understand certain characteristics. The most common characteristics of interest include capacity/scalability, response time and compliance with standards (internal and external.)

A Security Review is a specialized form of Design or Archiecture review that involves examining the architecture and the code looking for ways a malicious user or program might be able to break into the system.

Static Code Analysis involves examining the source code either manually or using tools to ascertain certain characteristics including:

- Reachability of code segments
- Correct usage of key language constructs such as type safety

Dynamic Para-functional Test Practices

Performance and stress tests are good examples of para-functional tests that require specialized tools. Sometimes we have complex or long-running test procedures that exercise the system-under-test just to see what will happen; there need not be enumerated expectations per se.

17.3.5 Result Assessment Practices

The value in executing a test case is to determine whether or not some requirement has been satisfied. While a single test cannot prove the requirement has been met, a single failing test can certainly prove that it has not been met completely. Therefore, most test cases include some form of assessment or checking of actual results against what we expect. This assessment can happen in real time as the test is executed or it may be done after the fact. This choice is a purely pragmatic decision based on the relative ease of one approach versus the other. There are the following three basic approaches to verifying whether the actual result is acceptable.

- Compare the actual result with a predetermined expected result using a comparator which may either be a person or a computerized algorithm.
- Examine the actual result for certain properties that a valid result must exhibit. This is done using a verifier.
- Just run the tests and not check the results at all. This may be appropriate when the testing is being conducted expressly to gather data. For example, the purpose of usability testing is to find out what kinds of issues potential users have using the product. We wouldn't report an individual usability test session as having failed or succeeded. Rather, we aggregate the findings of all the usability test sessions for a specific piece of functionality to determine whether the design of the system-under-test needs to be changed to improve usability.

Using Comparators to Determine Test Results

The most convenient form of assessment is when we can predict what the actual results should be in a highly deterministic fashion. The mechanism that generates the expected result is sometimes called a "true oracle" and there are several ways the test results can be generated. Tests that have a true oracle are very useful when doing Acceptance Test Driven Development because there is a clear definition of "what done looks like."

When there is an existing system with similar functionality and we expect the new system to produce exactly the same results, it may be convenient to use the existing system as a Comparable System Test Oracle.

When we have a new system for which no comparable system exists, we often have to define the expected results manually. This is known as a Hand-crafted Test Oracle. Once the system is up and running we may also have the option of comparing subsequent releases of the software with previous versions, an approach we call Previous Result Test Oracle. This is the approach that most "record and playback" or "capture / replay" test tools use. In some cases it may be appropriate to forgo the Hand-Crafted Test Oracle and wait for the system to generate results which are then inspected by a person, a

Human Test Oracle, before being used as a Previous Result Test Oracle. This approach forgoes the benefits of Acceptance Test Driven Development.

Using Verifiers to Determine Test Results

When we cannot predetermine exactly what the expected result should be, we can instead examine the actual result for certain characteristics. Rather than have an oracle describe what the result should be, we ask the oracle to make a judgement as to whether the result is reasonable given the input(s). This approach has the advantage of not requiring that we predetermine the expected result for each potential input. The main disadvantage is that we may accept some results that satisfy the invariant but which are not actually correct.

For example, a Human Test Oracle could examine a generated graphic to determine whether they can recognize it as an acceptable rendering of an underlying model. Or a computer algorithm could verify that when the actual result is fed into another algorithm the original input value is recovered. The program that implements this algorithm is sometimes called a Heuristic Test Oracle. Heuristic Test Oracles may be able to verify some results are exactly correct while for other results it may only be able to verify they are approximately correct.

For example, we could write a test script that steps through all integers to verify that the square root function returns the right value. Rather than inspect the actual values returned by each function call and compare them to a hand-crafted test oracle or a comparable system oracle, we could instead multiply the result by itself and verify that we get back the original number within a specified tolerance, say, +/-.001. In this example, we should also test against another invariant to ensure that the actual result is not negative which would clearly be a test failure. A computerized heuristic oracle is sometimes called a verifier.

Logging Bugs

One of the key reasons for testing and reviews is to find differences between expectations and reality. When we do find such a difference it is important that it be logged so that it can be further investigated, prioritized and the appropriate action determined. The investigation could reveal it to be a bug, a misunderstanding by testers about how the system was intended to be used, missing documentation, or any of a myriad of other reasons. To avoid presuming the outcome, we prefer to call these *concerns.* To allow the investigation to be conducted efficiently, it is important to log the key characteristics of each concern. At a minimum, we need to log the following:

1. The exact steps required to reproduce the problem. This may require rerunning the test a number of times until we can confidently describe exactly what it takes to cause the problem to occur.
2. What actually occurred.
3. What we expected to happen. We should provide as much detail as would be necessary for the reader to understand. We should not just refer to a requirement but rather describe

exactly what we expected, what happened, and how what actually occurred was different from what we expected.

Every potential bug report should be given a clear title that describes specifically what was tested; this avoids confusion between bugs with very similar titles yet completely different expected and actual behaviours.

Refer to [Kaner et al, Chapter. Reporting and Analyzing Bugs.] on bug reporting guidance and http://blogs.msdn.com/productfeedback/archive/2004/09/27/235003.aspx for an additional example.

17.3.6 Test Maintenance Practices

Some tests are only intended to be run once while some are intended to be run many times over a long period of time. Some kinds of tests hold their value longer than others; some kinds of tests deteriorate very quickly because they are so tightly coupled to the SUT that even small changes to the SUT make them obsolete. Tests that are expected to be used more than once may warrant an upfront investment to ensure that they are repeatable and robust.

Useful techniques for making tests more robust include the following:

- Build maintainability into the tests. Write the tests at appropriate level of abstraction. Don't couple the test to any part of the system it isn't testing. Don't provide any unnecessary or irrelevant detail in any of the steps of the test. Strive to describe the test steps in business rather than technical terms.
- Design the system-under-test for testability. Designing for testability is common practice in computer hardware but is too often neglected in software design. Design the system to make it easy to put it into a specific state. Make it easy for test programs to interface with the system through application programming interfaces (API) rather than forcing programs to use interfaces intended for humans. Writing the tests before the system is designed is a good way to influence the design to support testability.
- Refactor the tests to improve maintainability. Tests should be assets, not liabilities. Tests that are hard to understand are hard to maintain when the system-under-test is modified to meet changing requirements. Automated tests in particular should be refactored to avoid unnecessary duplication and irrelevant information. See the Test Evolution, Refactoring and Maintenance thumbnail.

17.4 Summary

This chapter introduced the practices we use while defining, executing and maintaining individual tests. While some of these practices are specific to functional testing and others are specific to para-functional testing, the overall lifecycle of a test is consistent for both categories of tests. The practices used for

readiness assessment are more or less the same as those used for acceptance testing although the specific tests produced for each will depend on the overall test plan as described in Chapter 16 – Planning for Acceptance.

17.5 What's Next?

The next chapter describes practices related to managing the acceptance process, especially how it relates to monitoring and reporting on test progress and the results and managing the process of deciding which bugs to fix.

17.6 Resources

[ref1] A website dedicated to cataloging all the tools available for allparis testing. See http://www.pairwise.org/tools.asp

[ref3] Tabaka, Jean "Collaboration Explained" Addison Wesley. NJ Addison Wesley. NJ

[ACA]Cockburn, Alistair "Agile Software Development: The Cooperative Game" Addison Wesley Professional

[Kaner et al] Kaner, C. et al. Testing Computer Software, 2/e. Wiley, 1999.

Chapter 18 Managing the Acceptance Process

The previous chapter introduced the individual test lifecycle and the practices the assessors use for identifying test conditions, designing the tests, executing the tests and assessing the outcomes, and maintaining the tests. This chapter introduces the management practices we use while executing the tests and while making the acceptance decision.

The process of accepting software involves many activities that generate the data we use as input into the acceptance decision. Each of these activities takes time and consumes valuable human and other resources. A well managed acceptance process will use these resources wisely while a poor managed process has the potential to waste these resources, delay the acceptance decision and even compromise the quality of the decision made.

While we are executing our test plans, we want to know the answers to two important and inter-related questions:

- First, how is testing progressing? When will we have a high-confidence assessment of the product quality? Will it be in time to make our readiness or acceptance decision?
- Second, what is our current assessment of the product quality? Is it good enough to release whether to customers or to the acceptance testers? What actions do we need to take to make it good enough?

18.1 Test Scheduling Practices

The overall schedule that defines which tests will be run and when should have already been defined as part of the test plan. That includes the strategic decisions about whether all testing is done during a final test phase or incrementally throughout the project. In this section we focus on the techniques we use for detailed scheduling of test execution. As a reminder, under test execution we include both dynamic testing when test cases are run that involve executing software in the system-under-test and static testing that is performed in the form of reviews or inspections without actually running code in the system-under-test.

18.1.1 Plan-Driven Test Scheduling

The traditional approach to test scheduling involves defining a period of time, sometimes called a test cycle, in which one complete round of testing can be completed. The test plan defines how many test cycles are planned. The details of what happens in each test cycle often emerge as the project unfolds. One approach is to define a detailed project plan for the first test cycle, one which defines the set of

testing activities that will be done on a day by day basis. Project management tools such as Pert or Gantt charts may be used to document this detailed test execution plan. (See Plan-Driven Test Management.)

18.1.2 Session-Based Test Scheduling

The alternative to defining a detailed project plan of test activities is to schedule a series of test sessions and create a backlog (prioritized list) of test session charters that are executed in these test sessions. Each session would be of a standard duration, say 90 minutes, and most charters should be executable within one session. This Session-Based Test Management is most typically used for managing exploratory testing but could also be used for execution of scripted testing. When each test session is completed the tester marks the test charter as one of *completed* or *needs one or more additional sessions*, or suggests additional test charters for future test session. This provides a good indication of how testing is progressing as illustrated in figure K which shows the number of test charters dropping over time but with the occasional upwards spike as new test charters are identified.

<need burndown chart with added charters plotted in a different color or pattern>

Figure K

Test Charter Burndown

18.1.3 Self-Organized Test Scheduling

A third approach is often used by agile or self-organizing teams. All the testing tasks are posted on a wall chart, variously called a “big visible chart” [Ref, RJ] or an “information radiator”[ACA], and team members sign up to do specific testing tasks. As they finish one task, they mark it done and pick another task to perform. This is called Self-Organized Test Scheduling.

18.1.4 Event-Triggered Test Scheduling

Some forms of automated testing can be set up to run automatically at certain times (such as every night at 2am) or when certain events occur (such as when someone checks in a change to part of the code base.) The results can be posted on a web site, automatically e-mailed/IM’d to all team members, or communicated by a multi-colored light display or “lava lamps” in the team workspace or icon in everyone’s computer’s system tray or side gadget (for one such tool see Team Build Monitor - http://blogs.msdn.com/jimlamb/attachment/3467297.ashx). This practice is often called Continuous Integration [Ref. Martin Fowler’s CI article: http://martinfowler.com/articles/continuousIntegration.html]or CI for short. Teams that use continuous integration typically adopt a “stop the line” mentality to failing tests. The goal is to promote “code health” early. Whenever a test failure is reported by the automated test harness, fixing the failed test becomes the top priority for everyone on the team . This is the software equivalent of the “stop the line” practice used in lean manufacturing. Once the failure is fixed, everyone can go back to working on their respective tasks. This focus on keeping the product code ”healthy” improves quality and and reduces the duration of the acceptance test cycle.

Another approach is to prevent changes that break the build from being checked-in. The Gated Check-in feature of the Team Foundation Server defers a developer’s check-in until it can be merged and

validated by an automated build. Passing automated tests or validating results of the static analysis could are examples of the validation policies. For more information on Team Foundation Build, see http://msdn.microsoft.com/en-us/library/bb668958.aspx

18.2 Test Progress Reporting Practices

When we first start executing the tests we don't know whether the quality is high or low but we do know that we don't have a high degree of confidence yet. As testing progresses, we should be getting a better idea of what the quality level is and how much longer it will take us to get to the required level of confidence. This is very similar to the cone of uncertainty concept that predicts the completion date and/or cost of developing the software. Figure x illustrates the cone of uncertainty for quality for a typical project. The initial guestimate was anywhere between 10 and 100 person-months. As the requirements were better understood, the range reduced somewhat but a sudden discovery of additional scope raised the estimates once again. An effort was made to descope the project to recover the original timelines. Work creep slowly raised both the upper and lower limit and the estimates finally converge when the product is accepted as-is.

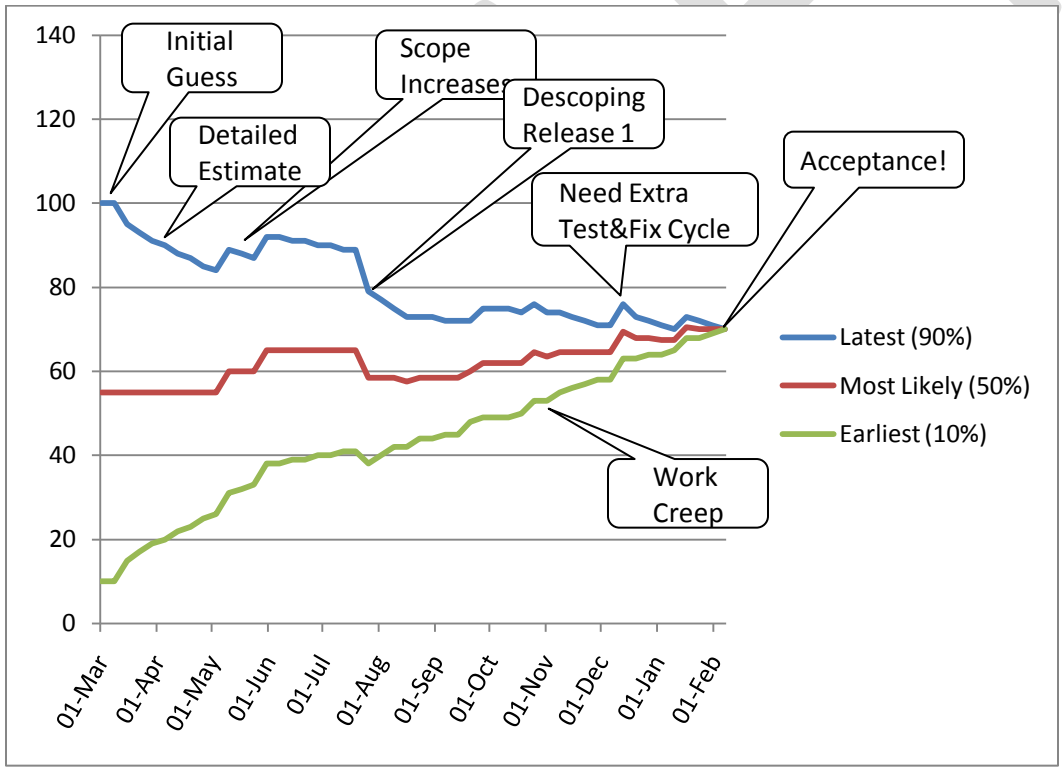


Figure x

Cone of Uncertainty for Quality

We can, of course, assume that we will have a reasonably accurate assessment of quality when we have finished all our planned testing. This assumption may or may not be correct because it depends on how effectively our planned test activities will find all the important bugs. This also implies a clear understanding of what is important to the customer. This is very difficult to assess ahead of time. We

certainly need to monitor how much testing work remains to be executed in the current test cycle and when we expect to have it all completed. This applies to each test cycle we execute.

Session-based testing introduces a feedback mechanism that explicitly allows us to adjust the test plans as we learn new information both about the product and about the customer and the customer's needs. As each test session is completed the testers add any newly identified areas of concern to the backlog of test charters yet to be executed. Developers may also suggest additional test charters to address the risk associated with the modifications they make to the software as a result of change requests and bug fixes identified by tests. We monitor the size of the backlog of test charters to get a sense of whether we are making headway. As the confidence of the testers and developers improves they will suggest fewer and fewer new test charters and therefore the size of the charter backlog would drop faster.

Predicting when the software will be of good enough quality to deliver is difficult because that involves predicting how many test cycles we will require and how much time the supplier will require between the test cycles to fix the bugs. (See Chapter 16 – Planning for Acceptance for a more detailed discussion on this topic.) This requires monitoring the bug backlog and the arrival rate of new bugs to assess and adjust the predicted delivery date.

18.3 Assessing Test Effectiveness

One of the challenges of testing is assessing how effective our tests really are so that we can know how confident we should be in our assessment of the quality we have. There are several ways to calculate the effectiveness of the tests including the use of coverage metrics and using the find rate of intentionally seeded bugs.

18.3.1 Coverage Metrics

We can use metrics like test coverage and code coverage to calculate the theoretical effectiveness of our tests. These metrics can tell us what percentage of the requirements has been tested and what percentage of the code has been executed by the tests but neither of these is a direct measure of what percentage of the bugs we have found. Of course, the primary issue is that we have no idea of how many bugs really exist so it is pretty difficult to say with any certainty what percentage we have found. For a cautious approach to using code coverage, see Brian Marick's essay "How to Misuse Code Coverage" [http://www.exampler.com/testing-com/writings/coverage.pdf]

18.3.2 Defect Seeding

One technique the can provide a more direct measure is defect seeding. It involves deliberately placing a known set of defects in the software. As bugs are found during testing we can estimate the percentage of defects found by dividing the number of seeded defects found by the total number of seeded defects as shown in Figure Y.

$$\text{PercentDefectsFound} = \left(\frac{\text{SeededDefectsFound}}{\text{TotalSeededDefects}}\right)$$

Figure Y

Percentage of Bugs Found

We can project the number of defects yet to be discovered using the formula in Figure Z.

$$\frac{\text{FoundDefects}}{\text{TotalDefects}} = \left(\frac{\text{SeededDefectsFound}}{\text{TotalSeededDefects}}\right)$$

Or,

$$\text{TotalDefects} = \text{FoundDefects}\left(\frac{\text{TotalSeededDefects}}{\text{SeededDefectsFound}}\right)$$

Figure Z

Total Bugs Calculation

These calculations are described in more detail in the Test Status Reporting thumbnail.

18.4 Bug Management and Concern Resolution

A key output of testing and reviews is a list of known gaps between what the customer expects of the product and how it actually behaves. While the progress of testing is usually reported in terms of which tests have been run and which haven't, the gap between expectations and reality is usually expressed as a list of known bugs or issues that may or may not have to be addressed before the product will be accepted. In Chapter 1 – Planning Afor 6cceptance, we introduced the Concern Resolution Model which describes how concerns, including software bugs, change requests, issues and documentation bugs, can be managed. Concerns that fall into any of these categories could be considered gating (also known as blocking or blockers), that is, would prevent the system from being accepted. Once we have finished our first full pass of testing, presumably at the end of our first test cycle, we can think of the set of gating bugs as being the outstanding work list for the supplier. Our goal is to drive the list of gating bugs down to zero so that we can consider accepting and releasing the product. (Note that just because it reaches zero does not imply that there are no gating bugs, just none we currently know about.) The customer can influence the gating count in two ways: the customer can classify newly found bugs as gating or they can reclassify existing gating bugs as non-gating if they decide that the bug can be tolerated because there is a low enough likelihood (reduced probability as described in Chapter 5 – Risk Model) it will be encountered or there are acceptable work-arounds in the event that it is encountered (reduced impact per the risk model.)

18.4.1 Bug Triage

It is common practice to classify the severity of bugs based on their business impact. Usually a Severity 1 (or Sev 1 as it is commonly abbreviated) bug is a complete show stopper while a Sev 5 bug is merely cosmetic and won't impact the ability of users to use the product effectively. Many customers insist that all Sev 1 & 2 bugs be fixed before they will accept the product. Some customers consider Sev 3 bugs

critical enough to insist they are resolved before the product can be accepted. Note that the interpretation of the severity scale is merely a vocabulary for communication of the importance of bugs between the customer and the supplier; it is entirely up to the customer to make the decision whether the bug needs to be fixed. The supplier may influence that decision by pointing out similarities or difference with other bugs that had the same severity rating or by pointing out the potential impact as part of an argument to increase or decrease the severity. They might also point out potential workarounds or partial fixes that they feel might justify reducing the severity. But ultimately, it is the customer decisions as to whether the product is acceptable with the bug still present.

The customer needs to be reasonable about what bugs should be classified as gating. If there are 1,000 bugs and the team can fix 20 bugs per week, it will take the team 50 weeks to fix all the existing bugs assuming that no new bugs are found and no regression bugs are introduced by the bug fixes. Both these assumptions are highly optimistic. Therefore, the customer needs to ask ”Which of these bugs truly need to be fixed before I can accept the product?” This is purely a business decision because every bug has a business impact. Some may have an infinitesimally small business impact while others may have a large business impact. The customer needs to be opinionated about this and to be prepared to live with the consequences of their decision whether that is to delay acceptance, deployment and usage of the product until a particular bug is fixed or to put the software into use with the known bug present. There is no point in insisting that all bugs must be fixed before acceptance simply to be able to say that there are no known bugs. Doing so will likely delay accruing any benefits of the system unnecessarily. The process of deciding the severity of each bug and whether or not it should ever be fixed is sometimes called Bug Triage.

18.4.2 Bug Backlog Analysis

This list of known bugs is a collection of useful knowledge about the state of the product. Most bug management tools include a set of standard reports that can be used to better understand the bug backlog. These include:

- Bug Fix Rate Report – Describes the rate at which bugs are being fixed.
- Bug Arrival Rate Report – Describes the rate at which new bugs are being reported. A decreasing arrival rate may be an indication that the return on investment of testing has reached the point of diminishing returns. Or it could just mean that less testing is being done or that the testing is repeating itself. Long-lived products that release on a regular cycle typically find that the shape of the curve is fairly constant from release to release and this can be used to predict the ready-to-deliver date quite accurately. See Figure D – Bug Arrival Rate and Figure E – Cumulative Bugs Found.
- Bug Debt or Bug Burn Down Report - Describes the rate at which the number of bugs is being reduced or is increasing. The burn down report aggregates the bug arrival rate and the bug fixing rate to allow us to predict when the number of gating bugs will be low enough to allow accepting the product and releasing it to users.

- Bug Aging Report – Classifies bugs by how long it has taken to fix them and how long it has been since unresolved bugs were first reported. The latter will give an indication of potential level of customer dissatisfaction if the average age of bugs is large.
- Bug Correlation Report – Classifies bugs based on their relationship with attributes of the system under-test. The customer is typically most interested in which features have the most bugs because this helps them understand the business impact of accepting the system without resolving them. The supplier, on the other hand, is typically interested in which components, subsystems, or development teams have the most bugs associated with them because this helps them understand where their own internal processes need to be improved most. Figure Z is an example of a Bugs by Area or Bugs by Feature Team report.

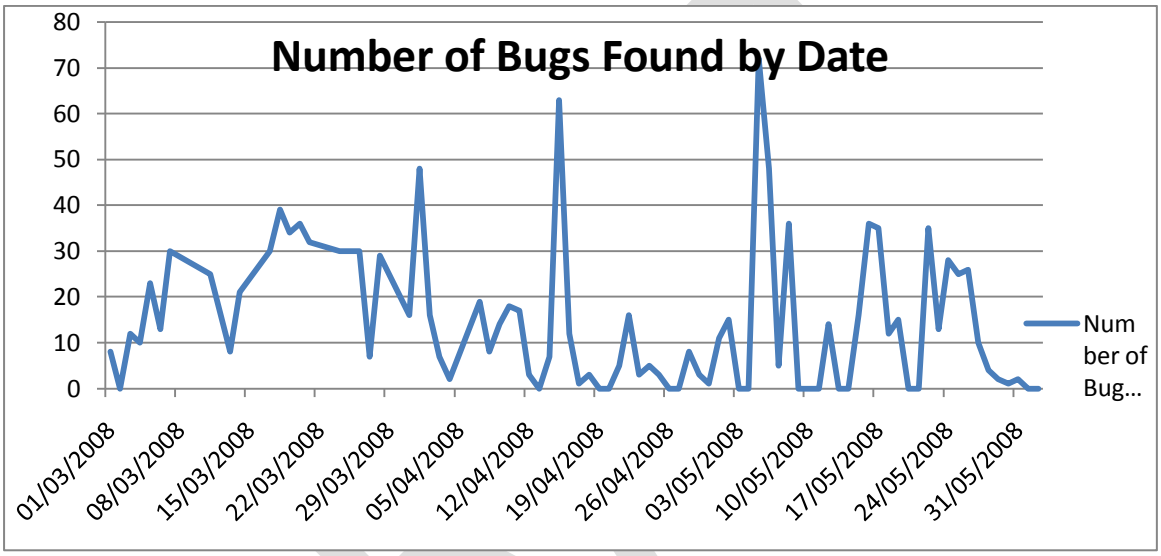


Figure D – Bug Arrival Rate

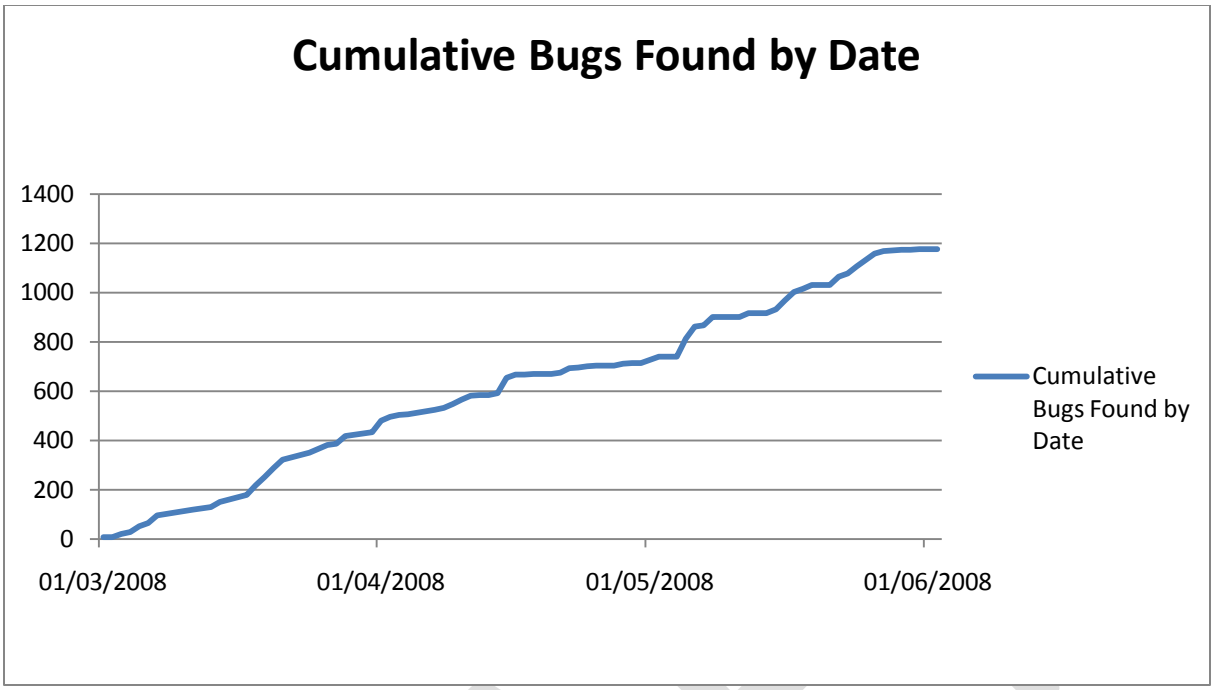


Figure E – Cumulative Bugs Found.

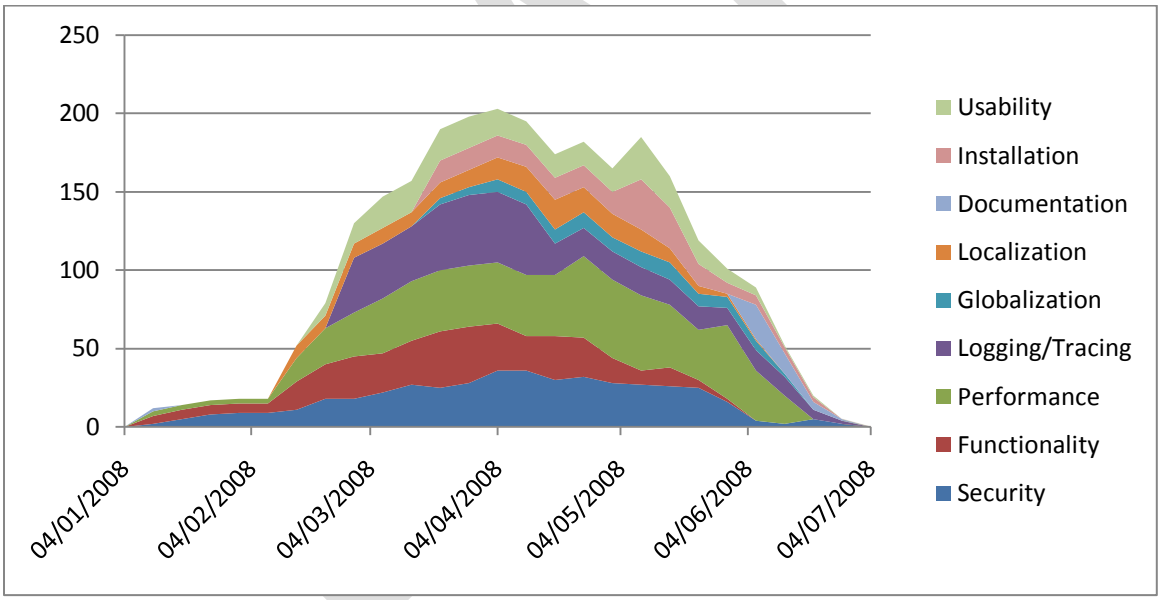


Figure Z

Bug Correlation Report

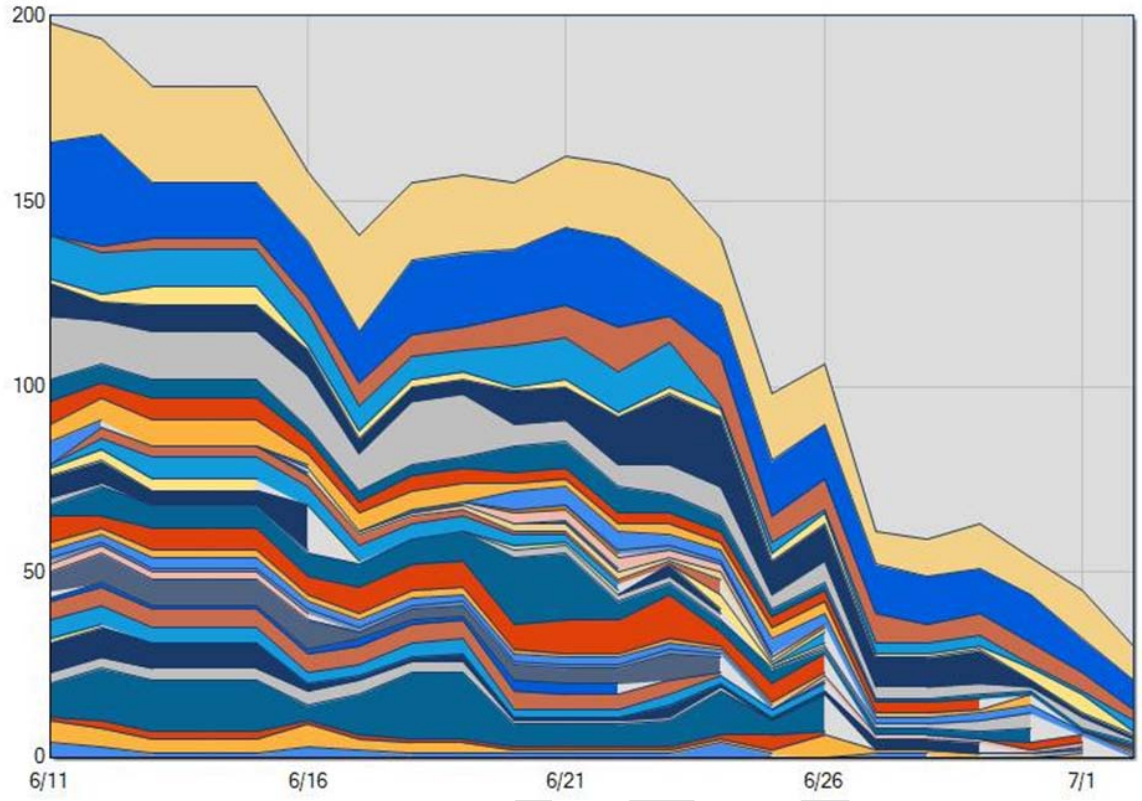


Figure W – Bug Debt/Burn Down Report

18.5 Test Asset Management Practices

The artefacts produced while planning and executing the assessment process are assets; these test assets need to be managed. If repeatability of testing is considered important, such as when the same tests need to be used for regression testing subsequent releases, test scripts and the corresponding test data sets need to be stored in a version-controlled test asset repository. This may be a document repository such as SharePoint or a code repository such as the Team Foundation Server repository, Subversion or CVS. This is described in more detail in the Test Asset Management thumbnail.

When test assets are expected to be long lived such as when the same tests will be used to regression test several releases of a product, it is important to have a strategy for the evolving the test assets to ensure maintainability. The Test Evolution, Refactoring and Maintenance thumbnail describes how we can keep our test assets from degrading over time as the product they verify evolves.

18.6 Acceptance Process Improvement

The acceptance process consumes a significant portion of a project's resources. Therefore, it is a good place to look when trying to reduce costs and improve the effectiveness of one's processes. Two good candidates for process improvement are improving the effectiveness of the test practices and

streamlining the acceptance process to reduce the elapsed time. The latter is the subject of the next chapter.

18.6.1 Improving Test Effectiveness

Multi-release projects and long-lived products will go through the acceptance process many times in their lifetime. Each release can benefit from lessons learned in the previous release, if we care to apply the lessons. It is worth conducting one or more retrospectives after each release to better understand which readiness assessment and acceptance testing activities had the most impact on product quality and which had the least. The highly effective activities should be continued in future releases and the ineffective ones should be replaced with something else. Some teams make it a point to try at least one new practice each release to see if it will detect bugs that previously slipped through the readiness assessment and acceptance testing safety nets. It is also worth analysing the list of defect found in the usage phase to identify any shortcomings in the safety net. Bug correlation reports can come in handy in this exercise. See Figure Q – Bugs by How Found Report.

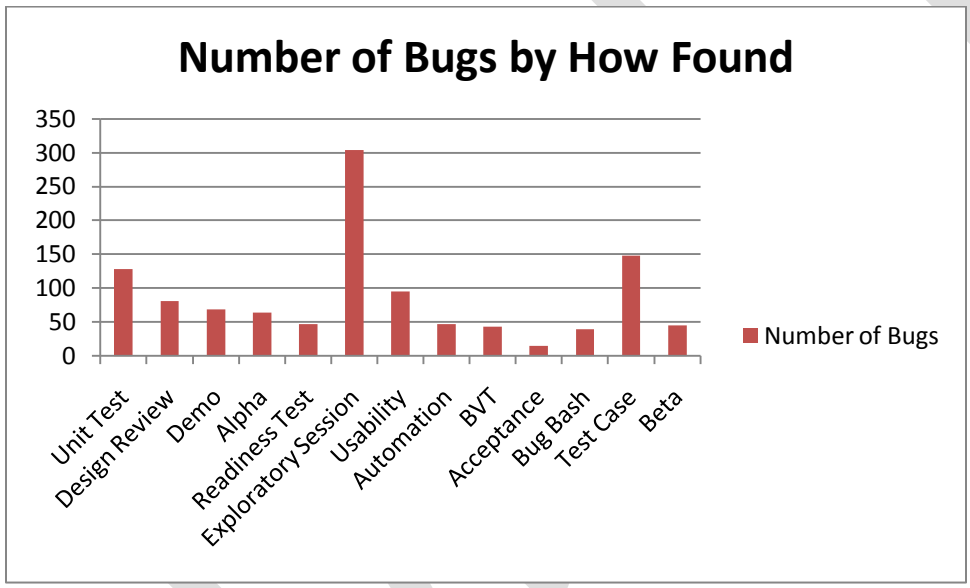


Figure Q – Bugs by How Found Report.

18.7 Summary

Like any process, execution of the acceptance process needs to be managed. This includes monitoring the execution of the planned testing and ensuring it results in data that allows for a high-confidence acceptance decision. A key aspect of managing acceptance is the managing the bug backlog to get the best possible quality product at the earliest possible time. These two goals are at odds with each other and deciding which takes precedence should be a business decision. When building product is an ongoing goal of the customer organization, continuous improvement of the acceptance process should also be managed to further reduce time to usage and improve product quality.

18.8 What's Next

The acceptance process typically takes up a significant portion of a project's resources and elapsed time. The next chapter looks at ways both elapsed time and resource usage can be reduced.

18.9 Resources

[ACA] See 70 – Assessing Software.docx.s

Chapter 19 Streamlining the Acceptance Process

The previous chapters of Part III introduced the practices involved in planning and executing the acceptance process. By reading those chapters the reader should be able to get a basic understanding of what is involved in accepting software. This chapter describes how we can organize the acceptance process to minimize the elapsed time and resources used while making the acceptance decision.

The acceptance process is a necessary but potentially wasteful exercise. We should strive to keep it as simple and value-adding as possible. The longer it takes, the more it costs in wasted effort and delayed delivery of business value. It can become a serious impediment to being responsive to our users (the product owner's customers.)

We can improve our understanding of an existing or proposed acceptance process through an exercise called Value Stream Mapping. This is a form of business process modeling that focuses our attention on the ratio of elapsed time and the amount of value added. Figure 1 is a value stream map of a hypothetical acceptance process.

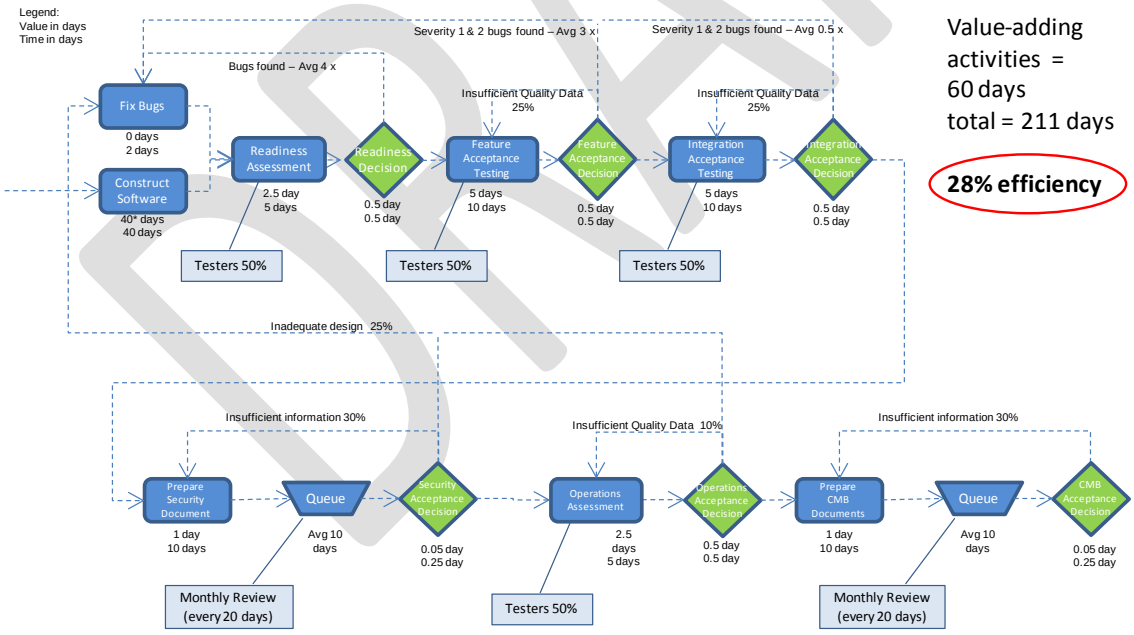


Figure 1

As-Is Acceptance Process

This process takes an average of 211 days to execute and provides only 60 days of actual value resulting in a cycle efficiency of only 28%. Some of the factors that make this process take so long to execute are:

- Sixty days of software development output is sent through the process in a single batch. Therefore, there is a large inventory of untested software which results in many bugs being found during the readiness assessment and acceptance testing activities. The fixing of these bugs is done entirely on the critical path of the project.
- There are several cycles in process each of which are typically executed several times. For example, Readiness Assessment sends the code back, on average 4 times for an additional 5 days elapsed time which adds zero days of value because it is pure rework. Each cycle takes 7.5 days (2.5 days fixing and 5 days readiness assessment) therefore this feedback loop adds 30 days of elapsed time.

Some tasks take longer than necessary because the resources are not dedicated. For example, the readiness assessors have other responsibilities and it takes them 5 days to do 2.5 days of testing.

- Customer acceptance is done in two separate phases and is followed by three other forms of acceptance decision making, two of which can send the software back all the way to bug fixing. The software needs to be retested each time the software is sent back to fix bugs.
- The security team is overworked resulting in an average wait time of 10 days for the security review. The preparation of the security documents takes an average of 10 days as developers discover what is required but only 1 day of that effort adds real value.
- The Change Management Board (CMB) meets monthly resulting in an average wait of 10 business days.

Note: The calculations assume that no value is provided by the bug fixing passes through the same activities as this is unplanned rework. Of course, this whole calculation is based on the assumption that we can add value by reviewing work; an argument can be made that the whole acceptance process is a form of waste caused by the customer's inability to accurately describe what they want built and the supplier's inability to follow the specification adequately.

This acceptance process can be streamlined by changing how the work passes through the various readiness and acceptance activities. Figure 2 illustrates the result of having applied a number transformations on the process. These transformations are inspired by Lean Thinking which focuses on eliminating waste. See the sidebar "Waste in Software Development" for a description of the seven common forms of waste. Which forms of waste we try to eliminate first depends on what is important to us. If time-to-market is the key consideration, we may want to tackle the queuing delays first and then look for ways to reduce the amount of processing. If cost/resource constraints are holding us back, we may want to look for ways to reduce the amount of processing first.

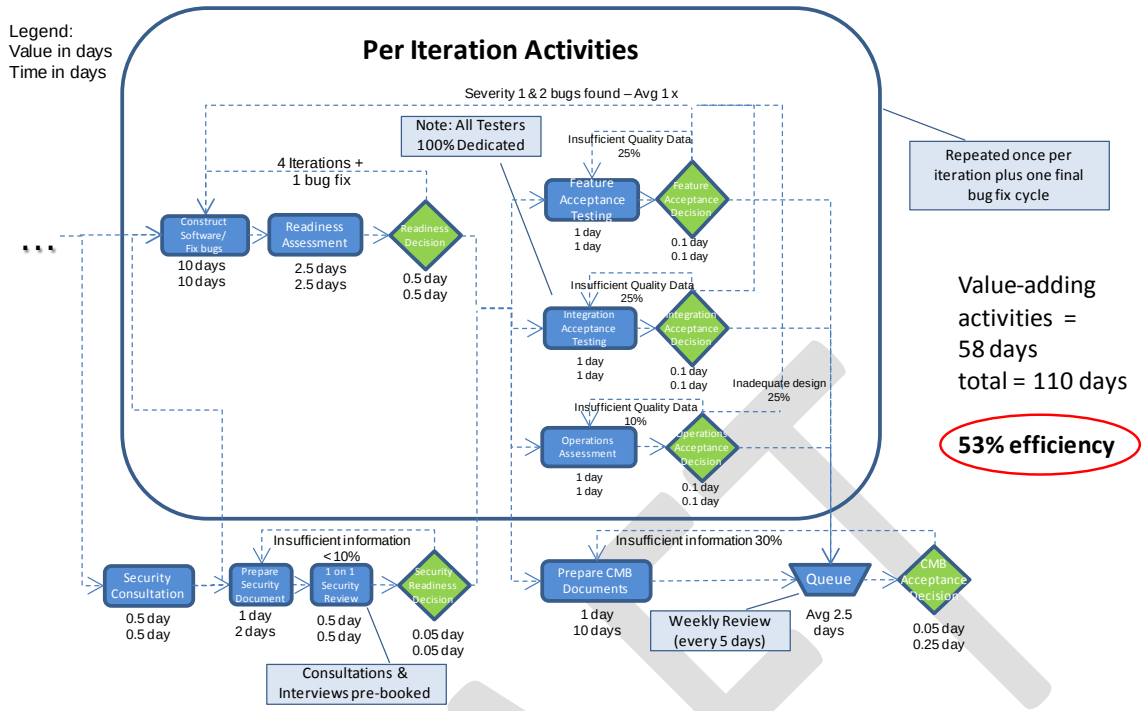


Figure 2

Streamlined Acceptance Process

This streamlined version of the process is the result of the following changes:

1. We have broken the project into 4 two week (10 business day) iterations. Each iteration is sent through readiness assessment, feature acceptance, integration acceptance and operational acceptance. Bugs found in the first three rounds of testing are fixed in the subsequent iteration. After the fourth iteration, the software is retested and the resulting bugs are fixed in a single bug fixing iteration. Contrast this with the 4 rounds of bug fix as a result of readiness assessment and three rounds due to feature acceptance testing. Each round of testing takes roughly 30% of the time it took in the As-Is process because the backlog of untested software is much smaller. This is an example of reducing waste associated with inventory.

2. The acceptance tests are provided to the development team along with the requirements. This allows the developers and the readiness assessment testers to verify the functionality is working correctly before handing it off to the acceptance testers thereby reducing the number of bugs found in acceptance testing. The readiness assessment shown as occurring after the iteration actually happens continuously (as soon as the developer says the feature is “ready”) therefore any bugs found in readiness assessment are fixed before the developer has shifted their focus on the next fixture. This reduces the cost of fixing the few bugs that are found in readiness assessment by a factor of 4. This is an example of reducing the waste associated with defects.

3. The acceptance review for security has been changed from an acceptance activity to a readiness activity. This allows it to be done in parallel with development. It has also been changed from

being a “push” model where the team prepares a document to be reviewed to a “pull” model. This starts with a consultation with the security specialist who helps the development team understand the appropriate security requirements and design and what needs to be in the security document. This provides input into the software construction process resulting in a security compliant design as well as more appropriate content in the security document. This reduces the turn-back rate from 30% to 10% and the total effort from 10 days to 3 days without any reduction in value. The effort to produce the document is reduced and the effort to read the document is also reduced; definitely a win-win solution. A final security review is held prior to the final iteration to review the finished document as well as any late-breaking security-related design considerations. This is an example of eliminating waste by reducing delays and by reducing extra processing.

4. The feature acceptance testing, integration acceptance testing and operations acceptance testing are done in parallel with an understanding that any showstopper bugs found in any of the parallel testing activities can result in the software being rejected. This reduces the elapsed time to the longest of the three instead of the sum of the three activities. This eliminates waste in the form of delays incurred while one group waits for another group to finish their work.
5. Because only 25% of the functionality is being tested anew in each iteration, the acceptance testing can be finished in 1 day. This is a short enough period that the testers can focus on the one project and finish it 1 day elapsed time. This isn’t a form of waste reduction as much as it is an example of improving “flow” by making the outputof the process less bursty via smaller batch sizes. Achieving flow is another of the key principles of lean development.
6. The Change Management Board documentation is prepared in parallel with the final bug-fixing iteration and can be submitted to the CMB as soon as the three parallel acceptance decisions are positive. To further reduce the delay, the CMB now meets weekly for 1 hour rather than monthly for a half day. This reduces the average wait from 10 business days to 2.5. This is an example of reducing waste associated with waiting.

The collected impact of these changes to the acceptance process has reduce the average elapsed time to 94 days to execute 110 days of value-added processing that delivers 58 days of value for a cycle efficiency of 53%.

Note: The calculations assume that value is provided by the planned development iterations but no value is provided by the final bug fixing iteration. The cycle efficiency was based on the cumulative duration including activities done in parallel even though they did not add any elapsed time.

Summary

A significant portion of the elapsed time between the finish of development of software and when the software can start providing value to the customer is consumed by the acceptance process. The elapsed time can be reduced significantly by building software incrementally and doing incremental readiness assessment and acceptance testing as each increment of software is finished. Different kinds of

inspection and testing activities can be done in parallel with development of the later increments reducing the amount of work that needs to be done on the critical path between completion of development and final acceptance.

What's Next?

Volume I has introduced a number of tools you can use for reasoning about how you accept software. It has described when to use a large number of acceptance-related planning, requirements, inspection, review and testing practices. You may want to research some of these practices in more detail. Volume 2 in this series describes many of these practices in more detail while Volume 3 provides examples of the artefacts that might have been produced on a fictional project.

Sidebar: Forms of Waste in Software Development

The seven common wastes of manufacturing can be remembered using the acronym TIM WOOD. In software, there is an equivalent of each form waste as exemplified by the list provided by Mary & Tom Poppendieck in their book Implementing Lean Software Development [ILSD]. Unfortunately, the software-specific names don't form a nice acronym so we've included both names here. The software-specific names have a * next to them.

T = The waste of Transport (Handoffs*)

Transportation is waste because it doesn't add any value to the end product but it increases the cost and the elapsed time.

In software, transport corresponds to handoffs between parties usually via documents. The requirements document handed by specifiers to software developers is one example, the design document handed by architects to developers is another. The preparation of these documents takes large amounts of effort – often much more than communicating the same information verbally. Handoffs usually result in loss of information and this is typically worse when the handoff is asynchronous (e.g. documents.)

I = Inventory (Partially Done Work*)

Inventory is bad because it costs money to produce the inventory and often costs money to store or manage the inventory. Inventory also masks issues by delaying when defective parts are discovered. Just-in-time manufacturing is all about reducing inventory to the lowest levels possible.

In software, inventory is any artifact that has taken effort to produce but which is not yet providing the customer with the value expected to be provided by having the software in use. Some common forms of inventory include:

Untested software – Software that has been written but not yet tested

Requirements documents – Documents that provide detailed descriptions of functionality that won't be built right away.

M = Movement (Task Switching*)

Movement is bad because while a worker is moving they can't be producing. This adds no value to the product but reduces productivity of the worker thereby increasing cost.

In software development, the equivalent of movement is task switching. This is caused by asking people to work on several things at the same time. Every time they switch between one task and another task, time is wasted while they re-establish their working context.

W = Waiting (Delays* or Queuing)

Whenever work stops while waiting for something to happen, it is a form of waste.

Common causes of waiting in software include waiting for approvals, waiting for clarification of requirements and waiting for slow computers or tools to finish their processing.

O = Overprocessing (Lost Knowledge* or Process Inefficiency)

Overprocessing is waste caused by doing unnecessary steps or doing a step longer than necessary. This adds no additional value but it does add cost.

In software, overprocessing is any steps in the production process that are required to produce high quality software. The most common example is the production of any documentation that no one will ever read. Another common form is having to rediscover information that was lost somewhere before it could be used.

O = Overproduction (Extra Features*)

Overproduction is producing too much. It is like inventory except that it is finished product while inventory is work-in-progress.

In software, overproduction is the development of unnecessary features. It has been reported [REF] that on average 80% of features developed are rarely or never used. This is clearly overproduction!

D = Defects

Defects, bugs, problem reports, usability issues all require extra work to analyse, understand and address. This extra work is pure waste. It is exactly the same in software.

Resources

[ILSD] Poppendieck, M., Poppendieck, T. "Implementing Lean Software Development – From Concept to Cash", Addison-Wesley, …. (See ref in another chapter.)

Appendix A - Reader Personas

This appendix describes two stereotypical organizations and the people involved in making or providing data for the readiness and acceptance decisions. The people are described as personas.

Note to production

Reviewers: These two stereotypes are included to act as a business context for the personas. The personas are included for two reasons:

7. To remind the authors and reviewers of our primary target audience.
8. To give readers a figure to empathize or associate themselves with. This should help them pick the persona that most closely resembles their own job responsibilities.

	Contents	Primary personas	Secondary personas
Volume I	Thinking Models	– Lisa Andrews – Eric Andersen	– Joseph Krawczak Yvette Kirwan Chatty Cathy – Fred Spook
Volume II	Thumbnails	– Joseph Krawczak Yvette Kirwan Chatty Cathy – Fred Spook	– Lisa Andrews – Eric Andersen
Volume III	Examples	– Joseph Krawczak Yvette Kirwan Chatty Cathy – Fred Spook	– Lisa Andrews – Eric Andersen

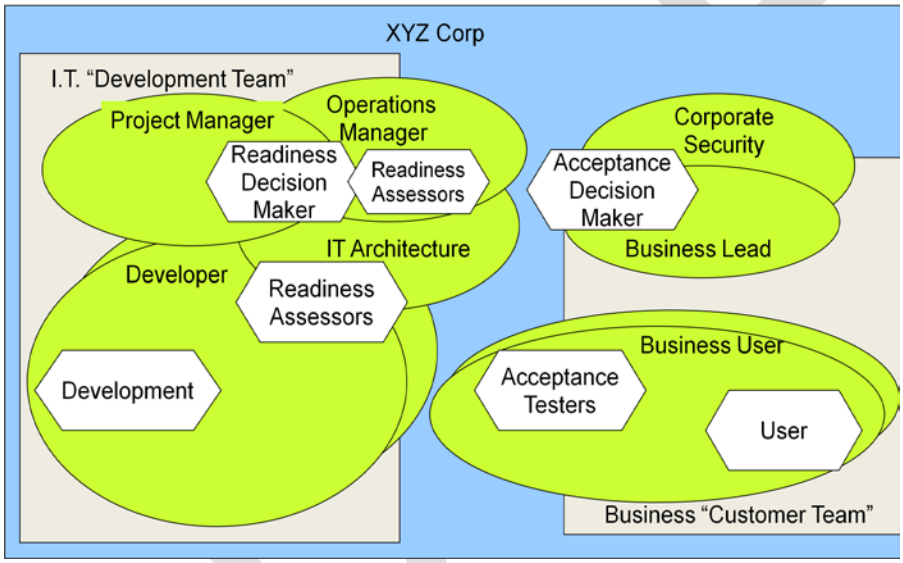
Software Built by an IT Department for the Business

XYZ Corp is a large transportation company. Computer software has been integral to the company's business for several decades. The company has several thousand "applications". At the high end are systems built by the IT department ranging from large mainframe systems written in Cobol and PL-1 to Web-based applications written in ASP.NET. At the low end there are end user applications built in Visual Basic and Access and spreadsheet applications built by end users scattered throughout the business areas.

The IT department will build the software based on the specifications provided by the business and, after doing its own system testing, expects the business to conduct user acceptance testing. There is no

separate QA department but each development team has at least one testing specialist who helps the team test their own software. The IT department's quality gate process also requires signoffs from the IT Architecture group and the Operational support group before software can be deployed into the Acceptance environment. As a result of recently introduced corporate governance initiatives (such as SOX), company policy requires approval by the Corporate Security department before any new application can be deployed into production.

The IT Architecture department includes a team of data architects whose primary responsibility is to own the corporate data model. The company views the data in the applications as a corporate asset and the data architects have been working hard to manage the duplication of data across them. The IT department also manages the lifecycles of the various IT technologies used in the company. New technologies have to be approved by the IT Architecture group before they can be used and old technologies may be marked for replacement. The Corporate Security department takes an active interest in all new applications and must approve the security measures before any new application can be deployed.



Business Person in the Marketing Division

Lisa Andrews is a business person in the marketing division of XYZ Corp. Lisa is in her late twenties (relatively young for her position in the somewhat conservative company) so she has good computer usage skills but is not by any stretch of the imagination considered "technical". This is Lisa's first IT project. Lisa has been asked to be the business lead of the project to replace several existing systems used by the staff in the marketing, sales and contracts departments with a single integrated system using the latest technologies. In this role she will have the final say on whether the system is acceptable to be deployed. She will also be responsible for ensuring that the business benefits used to justify the

project are actually realized. She will be assisted on this project by several users of the existing applications as well as a business analyst and a test professional. She will need to oversee this team as it defines the detailed requirements for the new system.

Key questions Lisa needs to answer:

What kinds of testing will my team need to do to help me decide whether or not to accept the software?

How can the release strategy (multiple releases, alphas, betas, community technology previews, etc.) help me?

E.g. Improve usability, fitness for purpose, user buy-in

Get cost savings earlier

When should I plan on doing AT? How much time should I allow for AT? How much effort will be involved? How much of staff do I need to do it?

Test last or incremental?

Am I involved in AT or do I delegate it to somebody else? From my org ? a third party lab? Who else can help me with AT if I don't have the skill/confidence

What kinds of testing do I expect the IT department to do before asking my team to do acceptance testing?

What information do I need IT to provide to me about the kinds of testing they have done and how will I use that information in deciding whether or not to accept the software (or even start acceptance testing?)

What information do I need to provide to IT prior to me doing AT?

Operations Department Manager

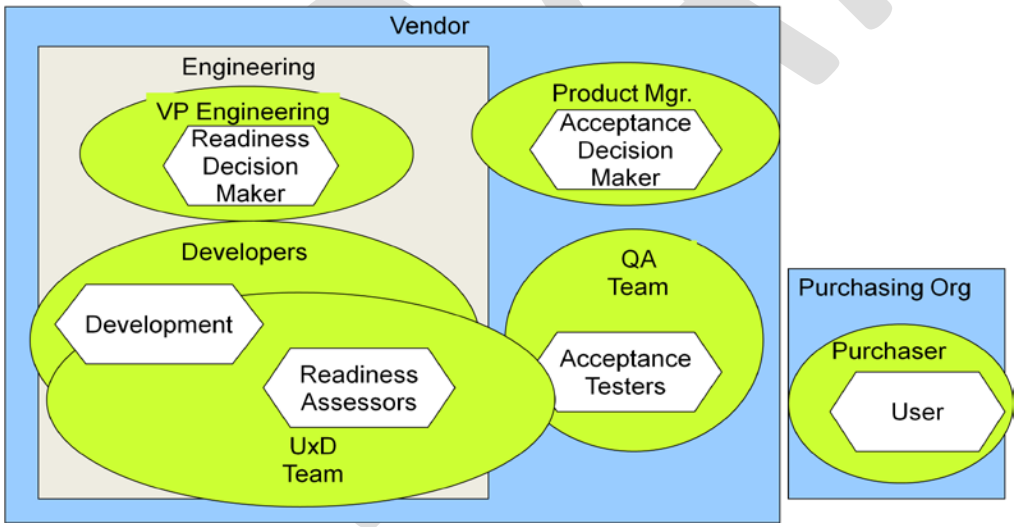
Chatty Cathy is the manager of the operations department. She is responsible for deploying all server-based applications and keeping them running according to each application's service level agreement (SLA). She's been with the company 30 years having worked her way up from being an administrative assistant in one of the business units. Her formal training was secretarial school and she's learned whatever she needed to know on the job. When she was first moved into I.T. it was as a tester for desktop applications but she quickly moved to up manage the desktop support team. When that function was outsourced, she was asked to take over the operations group. That was two years ago and she has only recently become comfortable with her understanding of how the group functions. She has yet to make any changes to the group's processes.

IT Security Specialist

Fred Spook is the IT Security Specialist in the Corporate Security department. He's a former spy ("If I told you which agency I worked for I would have to kill you," he says with a wink whenever he's asked.) who moved into the private sector when the first major security breeches were publicized. His background has made him rather paranoid and he delights in skewering the development teams with his arsenal of nasty security scenarios. As a result, most applications fail their first security review and require significant architectural refactoring before passing. The better development teams have learned to approach him for an early "off the record" security audit as soon as their design has stabilized.

A Product Company

Adventure Works is a product company that sells business productivity improvement products that include both hardware and software. Some of the products are sold as hardware with embedded software and some as software packages that leverage existing hardware products. The company has a separate UxD and QA departments. The UxD department is involved in part of the requirements definition activities associated with designing the product as well as ensuring that the products have a consistent look& fell. The QA department tests the software built by the Product Development group and advises the Product Manager on whether the software can be released to the market.



Product Manager or Product Owner

Eric Andersen is a product manager who has been asked to work with an agile development team as the Product Owner for one of the software-only products (a new software release that works with existing hardware supplied by the company and third parties.) Because the company's products are considered

high-technology, Eric is fairly tech-savvy. He has a good understanding of the customer's requirements but does get a bit lost when the discussion descends to the level of networking protocols.

Key questions Eric needs to find answers to:

What kinds of information do I need from the QA department to decide whether or not to accept the software? What quality criteria must be met? How do I know they have been met?

Who is responsible for verifying these criteria are met? Who is responsible for ensuring these criteria are met?

How does the UxD team participate in acceptance testing of the software?

How can the release strategy (multiple releases, alphas, betas, etc.) help me?

E.g. Improve usability, fitness for purpose

Get revenue earlier

Viral marketing

What is the process for addressing deficiencies prior to release?

Test Manager

Joseph Krawczak is the manager of the product verification group at Adventure Works. His team of professional testers verifies that the products built by the product development group work as specified by the product management team. His background is in testing telecom systems where he managed a 30 person testing department. When the telecom industry downsized he was given a sizable package and was hired by ABC Corp to head up it's newly created Independent Test department. His mandate was to improve the quality of the products produced by Adventure Works. His team of professional testers is expected to provide him with the data required to determine whether the product is acceptable. He provides this data, along with a accept/reject recommendation to the Product Manager who makes the final decision. He is always prepared to back up his recommendation with a clear description of the business impact of the bugs that he feels must be fixed before he would be comfortable advising the product be accepted.

User Experiensce Specialist

Yvette Kirwan is a Fine Art major who got a job as a graphic artist in a media company. When conventional advertising work took a downturn, she was moved into the product design group where she did graphic design for web-based software applications. She quickly realized that the graphics were mainly eye-candy and that the real value was provided by the interaction design so she went back to

school part time to get trained as an interaction designer. With this training under her belt she quickly became a key player in the design of software-intensive products at the company. She was hired by Adventure Works. specifically to work on the new product. Her role is to define the interaction design based on market requirements, user models and feedback from the product manager and the development. The designs need to be implementable in a cost efficient manner using the technology stack used by the product development team. While she is technically part of the UxD Team, she sits in the development area with the development team as she has found this to be a more effective way to transition design knowledge to the developers and their manager. She has also found it to be useful to enlist friends and people off the street to do informal usability testing of paper prototypes as well as the actual software. This gives her real data she can use to help influence the test department's understanding the users.

Appendix B – Key Points

The chapters in Part II described the process leading up to the acceptance decision from the perspectives of the Business Lead, Product Manager, Test Manager, Development Manager, User Experience Specialist, Solutions Architect, Enterprise Architect, Operations Manager, and Legal. For convenience, we summarized the key points here. There are Key Points common to all roles and others specific to each individual role.

The following table lists Key Points that are role specific and what roles they are specific to.

<The table will be updated and available on testingguidance.codeplex.com>

Key Points	Business Lead	Product Manager	Test Manager	Development Manager	User Experience Specialist	Operations Manager	Legal	Notes
Recognize there may be many decision makers.	x	x						
Recognize the choice of process impacts everything	x	x						
Recognize that complex products and organizations have complex decision making processes.		x						
Ensure all suppliers understand users and usage model.		x						
Agree on a definition of what done is			x					
Determine your test mandate.			x					
Get agreement from the product owner that the test cases reflect the requirements.			x					
Plan a multi-pronged test strategy.			x					
Collaborate with the development team.			x					
Do test automation when feasible and in a way it supports meeting your test objectives.			x					
Perform readiness assessments.				x				
Use acceptance tests to define success.				x				
Define tests first.				x				
Minimize untested code.				x				
Use testing to prevent bugs.				x				
Define the usability component of the acceptance criteria as early as possible.					x			
Address the usability risks as early as possible.					x			
Define acceptance from the operational perspective as early as possible.						x		
Identify ways to reduce the number of test & fix cycles						x		
Include clear stipulations of any contractual implications.							x	
Clearly define how much time is allowed for acceptance tests.							x	
Include a clear definition of acceptance tests.							x	
Know the explicit and implicit contractual conditions.							x	

Figure 1

Key Points specific to individual roles

Key Points Applicable to All Roles

- **Treat acceptance as a process not an event:** Appreciate the fact that acceptance testing will not be a single discrete event, but a process with a distinct start and end.

- **Define the scope of testing:** Have a clear understanding of what will be tested by the supplier and what you are expected to test. Typically, supplier is expected to do thorough (e.g.) testing of all capabilities and quality characteristics (such as performance, reliability, usability etc.) before handing off system for acceptance testing. Does the supplier have the environments for integration testing without in-house systems?
- **Delegate decisions:** If you don't have enough time or understanding to take part in the project, assign a proxy to represent you and have a clear understanding of the kinds of decisions they can make with/without consulting with you.
- **Delegate work/testing:** Have a clear understanding of your (organization's) abilities w.r.t. testing skills. Contract other parties to help you test if necessary. But don't underestimate the effort involved in managing the testing outsourcer as they will need to learn your business domain and requirements before they can be effective.
- **Understand the test environments:** Understand where you will test. Be clear on how you'll get the new release candidates delivered
- **View tests as requirements:** Acceptance criteria/tests should be articulated and prepared before the software is built and shared with vendor as a way to flesh out the details of the requirements.
- **Estimate acceptance effort & duration while realizing it is difficult to estimate:** Estimating the effort & duration of acceptance testing is hard; time-boxing the final acceptance testing cycle is common but requires past experience to come up with reasonable duration/effort. If in doubt, guess high and allow for several cycles of acceptance testing to give the vendor time to fix bugs.
- **Use incremental acceptance testing to reduce uncertainty of estimating:** Can avoid some of the uncertainty and most of the risk by doing incremental acceptance testing. Bugs found earlier identify ambiguity in the spec while there is still time to address it and can be fixed long before the final acceptance test phase
- **Understand the decision-making process:** Make sure you have a clear understanding of the decision making process. Other potential decision makers include Test Managers, Security Manager, User Experience/Usability Manager, Architect, Legal Department, etc. Ensure you have agreement on whether they make a decision or provide data or a recommendation to you.
- **Communicate information flow clearly:** Ensure anyone who you expect to provide you with data on which to base your decision understands what data they are providing you and when. E.g. The Test Manager provides test results to you and you make the acceptance decision.
- **Understand the operational requirements: Make sure you understand the operational requirements for your system. Whether you are selling a software package to customers or selling Software-as-a-Service (SaaS), be aware that whoever operates the software will have operational requirements over and above the functional requirements of the end users.**

- **Clarify decision making vs decision supporting:** Make sure you have a clear understanding of whether you are the gate-keeper (acceptance decision maker) or a supplier of information to the Product Manager for her to make the decision (preferred)
- **Maximize learning:** Work with the product owner to ensure that everyone on the team has access to the definition of success in the form of acceptance tests before the software is built

Key Points for Business lead

9. **Treat acceptance as a process not an event:** Appreciate the fact that acceptance testing will not be a single discrete event, but a process with a distinct start and end.
10. **Recognize there may be many decision makers:** The process may require decisions by many parties each supported by data collection activities. Each party likely has very different requirements and interests; you need to be aware of them and communicate them to the supplier(s) to ensure they address them.
11. **Define the scope of testing:** Have a clear understanding of what will be tested by the supplier and what you are expected to test. Typically, supplier is expected to do thorough (e.g.) testing of all capabilities and quality characteristics (such as performance, reliability, usability etc.) before handing off system for AT. Does the supplier have the environments for integration testing without in-house systems?
12. **Delegate decisions:** If you don't have enough time or understanding to take part in the project, assign a proxy to represent you and have a clear understanding of the kinds of decisions they can make with/without consulting with you.
13. **Delegate work/testing:** Have a clear understanding of your (organization's) abilities w.r.t. testing skills. Contract other parties to help you test if necessary. But don't underestimate the effort involved in managing the testing outsourcer as they will need to learn your business domain and requirements before they can be effective.
14. **Understand the test environments:** Understand where you will test. Be clear on how you'll get the new release candidates delivered
15. **View tests as requirements:** Acceptance criteria/tests should be articulated/prepared before the software is built and shared with vendor as a way to flesh out the details of the requirements.
16. **Recognize the choice of process impacts everything:** Two key styles of AT: Final Test Phase and Incremental AT. The former concentrates most acceptance activities at the end of the project. Incremental acceptance testing spreads out the work and reduces risk but requires all parties to work in different ways.
17. **Estimate acceptance effort & duration while realizing it is difficult to estimate:** Estimating the effort & duration of acceptance testing is hard; time-boxing the final acceptance testing

cycle is common but requires past experience to come up with reasonable duration/effort. If in doubt, guess high and allow for several cycles of acceptance testing to give the vendor time to fix bugs.

18. **Use incremental acceptance testing to reduce uncertainty of estimating:** Can avoid some of the uncertainty and most of the risk by doing incremental acceptance testing. Bugs found earlier identify ambiguity in the spec while there is still time to address it and can be fixed long before the final acceptance test phase.

Key Points for Product Manager

- *All Business Lead's Key Points plus:*
- **Understand the decision-making process:** Make sure you have a clear understanding of the decision making process. Other potential decision makers include Test Managers, Security Manager, User Experience/Usability Manager, Architect, Legal Department, etc. Ensure you have agreement on whether they make a decision or provide data or a recommendation to you.
- **Recognize that complex products and organizations have complex decision making processes:** There are several different ways to organize the teams to deal with very large products or products that are the amalgamation of several products/product lines (e.g SharePoint=Server+Office+IE). The choice should consider the resulting decision-making model (e.g. component teams, feature teams.)
- **Communicate information flow clearly:** Ensure anyone who you expect to provide you with data on which to base your decision understands what data they are providing you and when. E.g. The Test Manager provides test results to you and you make the acceptance decision.
- **Ensure all suppliers understand users and usage model:** Make sure you understand the users and communicate this to both supplier and test organizations.
- **Understand the operational requirements: Make sure you understand the operational requirements for your system. Whether you are selling a software package to customers or selling Software-as-a-Service (SaaS), be aware that whoever operates the software will have operational requirements over and above the functional requirements of the end users.**

Key Points for Test Manager

- **Determine your test mandate:** Make sure you understand whether you are doing readiness assessment or acceptance testing or both. For acceptance testing be sure the testers have realistic tests for the users they proxy. Consider using personas. For readiness assessment it

is important to have a good relationship with the development team to help them understand the quality of the product and what will improve it.

- **Clarify decision making vs decision supporting:** Make sure you have a clear understanding of whether you are the gate-keeper (acceptance decision maker) or a supplier of information to the Product Manager for her to make the decision (preferred)
- **Get agreement from the product owner that the test cases reflect the requirements:** Aim to write the acceptance test cases first and provide them to the development teams before the software is built to be used as aides to assist in understanding how the system should behave. This is known as Acceptance Test Driven Development.
- **Plan a multi-pronged test strategy:** Plan for a multi-pronged test strategy that includes incremental testing and both scripted and exploratory tests.
- **Collaborate with the development team:** Aim for an incremental delivery of functionality to support incremental acceptance testing. Work with the development team to design for testability to facilitate automated testing and enabling them to be able to run tests on demand.
- **Do test automation** when feasible and in the way it supports (not dictates!) meeting your test objectives. Consider having a dedicated test automation engineer.
- **Agree on a definition of what done is:** Get the various parties involved to agree on a definition of what done is for software before it goes through each of the quality gates of the gating model and include a checklist of requirements.

Key Points for Development Manager

- **Perform readiness assessments:** All software should be adequately tested by the development team before presenting it to the customer(s).
- **Agree on the minimum credible requirement (MCR) ahead of time:** The product owner, test organization and the supplier should all agree on the MCR ahead of time, that is, before the software is built.
- **Use acceptance tests to define success:** Use acceptance tests based on the MQR to define success before the software is built and ensure that everyone on the development team has access to them and runs the tests as they build the software.
- **Run the tests as part of readiness acceptance:** Run all the tests that the product owner will run and some you've defined.
- **Minimize untested code:** Test as you go, ideally incremental testing.

- **Use testing to prevent bugs:** Tests can help prevent bugs being introduced during the development cycle. Include functional testing, component/unit tests, and para-function testing as early as possible.

Key Points for User Experience Specialist

- **Understand your role:** Have a clear understanding with the product owner, product manager or business lead what your role shall be in making the acceptance decision.
- **Maximize learning:** Work with the product owner to ensure that everyone on the team has access to the definition of success in the form of acceptance tests before the software is built.
- **Define the usability component of the acceptance criteria as early as possible:** Work with the business lead or product manager to define the usability component of the acceptance criteria as early as possible.
- **Address the usability risks as early as possible:** The areas with highest usability impact should be built and tested first so that any usability-related issues can be addressed as early as possible in the development cycle.

Key Points for Operations Manager

- **Understand role in acceptance decision:** You and everyone concerned should be clear on your role in the making of the acceptance decision. Are you a data provider to the acceptance decision maker or an acceptance decision maker?
- **Identify ways to reduce the number of test&fix cycles:** Work with the development manager to identify ways to reduce the number of test&fix cycles through early delivery of testable software and incremental acceptance testing.
- **Define acceptance from the operational perspective as early as possible:** Ideally you have clearly defined acceptance criteria before the software is built.

Key Points for Solution Architect

- **Describe the whole solution to everyone**: Communicate the solution in its entirety to the whole supplier team to avoid the “My part works just fine (even if the solution doesn’t!)” mentality.
- **Ensure availability of acceptance tests:** Work with the requirements analysts and testers to ensure that the acceptance tests are available to the supplier team before the software is built.

- **Educate about tradeoffs**: Work with the product owner to help them understand what kinds of functionality is easy to build and what is hard. Help them get the most bang for their money.
- **Unbundle functionality**: Work with the product owner to help them identify the lower-value functionality that could potentially be deferred in case there is a need to trade off functionality for quality or delivery date.
- **Define SLA metrics**
- **Engage enterprise architects**: Engage the enterprise-wide architects (responsible for infrastructure, standards, technology decisions, long-term vision, security, data architecture, etc.) early so they can help you understand what requires their approval before it becomes part of the critical path.

Key Points for Enterprise Architect

- **Publish architectural acceptance criteria**: Publish general acceptance criteria so that all supplier teams understand the rules of the game and what kinds of approvals they need to get from you.
- **Articulate cross system integration requirements and SLAs**: Identify applicable integration requirements and service level agreements. Make sure the supplier team is aware of them and that they are acceptance criteria.
- **Ensure standards compliance**: Verify that products comply with applicable standards.
- **Encourage reuse**: Verify that product teams are aware of and capitalize on opportunities to reuse components and data.
- **Engage supplier teams early**: Engage supplier teams early to ensure that inspections prevent defects rather than find them.
- **Delegate non-critical decisions**: If it is not important to be involved in a decision then bow out and let the project team make the decision.
- **Keep approval process lightweight**: Make the process for seeking guidance and approval as lightweight as possible. Emphasize effectiveness of communication over comprehensive documentation. Avoid offloading work (e.g. more documentation) to the project teams to save you effort if that results in increasing the total work.
- **Be pragmatic**: Be pragmatic as to what progress individual project or product teams are expected to make towards long-term architectural objectives. They are responsible to the product owner, not to you.

Key Points for Legal

- **Include clear stipulations of any contractual implications**: Acceptance typically is a condition for rendering a payment and affects the application of any warranty provisions and potentially any remedies available to the customer.
- **Discourage one-sided contracts**: The contract should encourage both parties to get to a mutually agreeable acceptance as quickly as possible. Contracts that penalize one party often lead to dysfunctional behavior that results in lose-lose outcomes.
- **Clearly define how much time is allowed for acceptance tests:** Acceptance testing is always time-boxed. There is a limited amount of time for acceptance and reporting deficiencies.
- **Include a clear definition of acceptance criteria:** Besides testing performed at the location of supplier, and functional and operational tests performed when the software is deployed to the customer, software system often are required to successfully run with current data in a live operating environment for a period of time, with minimum bugs, before the system is formally accepted.
- **Know the explicit and implicit contractual conditions:** Acceptance can be viewed in terms of explicit provisions specified in the software development contract and the terms implied into the contract by law (which also depends on the geographic jurisdiction).