

STRATEGY

charter

funding

RELEASE

vision

estimation

ITERATION

release
plan

retrospective

DAILY

review

acceptance
testing

CONTINUOUS

iteration
plan

standup

build integration

TDD

the
**SOFTWARE
DEVELOPMENT
LIFECYCLE**

refactoring

collaboration

Merchant Link Passport to Learning




An Introduction to Software Development
Processes and Best Practices

This page intentionally left blank

Course Agenda

View the Software Development Life Cycle at 50,000 ft.

- Get a bird's eye view of the Software Development Life cycle.
 - Learn about SDLC activities and the development models used to manage small to colossal projects
- 

Understand Iterative Design and Agile Principles

- Learn about Big Design Up Front (BDUF) as well as Iterative development perspectives
- Get introduced to the Agile Manifesto and the concepts behind adaptive iteration models

Take the SDLC Journey

- Learn more about the SDLC principles and techniques behind great software development.
- See more detail on the practices and activities in iterative SDLC:
 - Gathering Requirements
 - Project Planning
 - User Stories and Tasks
 - Version control, Building your code, and Testing
 - Continuous Integration
 - Test Driven Development concepts
 - Ending an iteration and planning for the next one
 - Maintenance practices and bug fixes

This page intentionally left blank

What is the Software Development Life Cycle?

Introduction

All software begins with great ideas and customer's needs. As a developer, or development team your role is to bring life to those ideas. Taking those vague and often changing ideas and turning that into code that fulfills the customer's vision is often much more difficult than it may appear.

Changing customer expectations, incomplete or vague requirements, even factors impacting the availability and productivity of your team can affect your ability to. It all comes down to:

Software Development is about delivering:

**What is needed
On time AND
On budget**

If you are a project manager or have studied project management fundamentals, you may be wondering how this can be accomplished. In fact you may even see this as a contradiction to the project management triangle. The concept that you have three competing goals: **Time, Cost, Features/Quality** and that you can only choose two of the three.

At first glance, there seems to be a conflict. The constraints of time, money, and quality work cannot be applied equally. How do we reconcile this? We employ **Software Development Activities and Models** to manage our project and to balance these competing interests.

As we cover the various topics of the Software Development Life Cycle (SDLC), you will start to notice a lot of techniques and concepts that are identical to standard project management procedures. In fact, SDLC's are partly a method of applying project management practices to the act of developing software in a way that is **predictable** and **repeatable**. There are many factors to developing software that seems to make this an elusive goal, but like any process, it is a continual journey and not a destination.

Above all other concepts, this practice is about creating technologies that fulfill the needs of our customers. That main focus will be seen throughout the concepts and practices discussed and should be core to your approach.

Before we get into the specifics, let's look at the activities and models at a high level so we have a roadmap to follow.....

Software Development Activities

All software development has a core set of activities regardless the process you use to perform them, the size of the project or of the project team.

Planning

Like any construction project, the final product is completely dependent on the initial planning. Without proper planning you risk complete failure and in some cases damages from such a failure.

Understanding **what** you are attempting to do, and **how** you will accomplish it is critical. Most customers have a general idea of what they want the end result to be, but may not know what the software should do to reach that result.

Gathering requirements is critical to the planning process. Often customer's requirements are incomplete, ambiguous, or may be contradictory. This requires skilled analysts and software engineers to refine these requirements.

Determining the scope of a project includes development tools and resources needed to deliver the final product. Certain functionality may be out of scope for the project in terms of cost or time.

Create estimates and assign priorities for each requirement. Estimates for implementing the requirements are set by the developers. In all cases, priorities are set by the customer.

Implementation, Testing, and Documentation

The next set of activities involves the actual design and coding of the product. Implementing or coding software requires a supportive set of development tools and environments. Effective use of these tools can make or break your project, and ultimately your business.

Employing version control and testing tools allows developers to document changes, create branches to implement new versions while still maintaining older versions of code, and even automate testing to expose bugs and prevent introducing new ones. These tools also help manage tasks like merging code in a collaborative environment, and to introduce third-party code in a less disruptive manner.

In some cases the tools to manage these development tasks also provide ways to effectively document the code and to maintain a history that any new developer can use to better understand the code and the decision making process by prior developers to handle a problem a certain way.

Deployment and Maintenance

After all the development and testing (and the final acceptance by the customer) comes deployment. This is the major milestone for the effort and is supported (well or not) by all that has come before it. Through all of the planning, coding, and testing, this is where the goal is met. So you're done..... right?

Nope. The final activity is the **maintenance of the product**. This deals with patching bugs and other flaws that may have survived all of the testing. With a proper process you can significantly reduce the impact and number of bugs your software will encounter.

Pretty straight forward right? On the surface these activities are understandable and seem pretty easy to cope with. The real question is how to perform those activities in a way that lets you deliver what is needed, on time, and on budget. A structure to manage how and when these activities take place needs to be used.



Software Development Models

There seem to be as many software development models as there are projects and developers. They all provide their own advantages and disadvantages. They have also been born of vastly different industries to accomplish the same goal: (Yes, AGAIN) Delivering what is needed, on time, and on budget.

Like most things in life these models have their religious followers and pundits. You should approach these models as a large toolbox and simply choose the model that fulfills your needs. You will need to understand the pros and cons of particular approaches.

SDLC Models all have the same activities. The general differences are:

- the number of iterations
- *when* a particular activity is performed
- *how often* a particular activity is performed
- how involved your customer is and the general level of communication
- which activities are emphasized and which are simplified
- project scopes and their impact to and by certain models

It is true that the list above is an oversimplification and somewhat subjective but the truth is that there is no silver bullet SDLC processes or model. The process or model should be a framework to **support** the development effort and **not BE the effort**. It is a set of practices to get you to your goal. You should always remember:



Good Developers Develop, Great Developers ship

Good Developers can usually overcome a bad process

A good process is one that lets YOUR team be successful

The Waterfall Model

The Waterfall Model approaches software development with a long term perspective and is highly plan-driven. Waterfall processes have historically been used for large scale, long term projects. This process tends to be prone to failure as it follows a strict cycle that typically does not deliver any functional products until the whole process has completed.

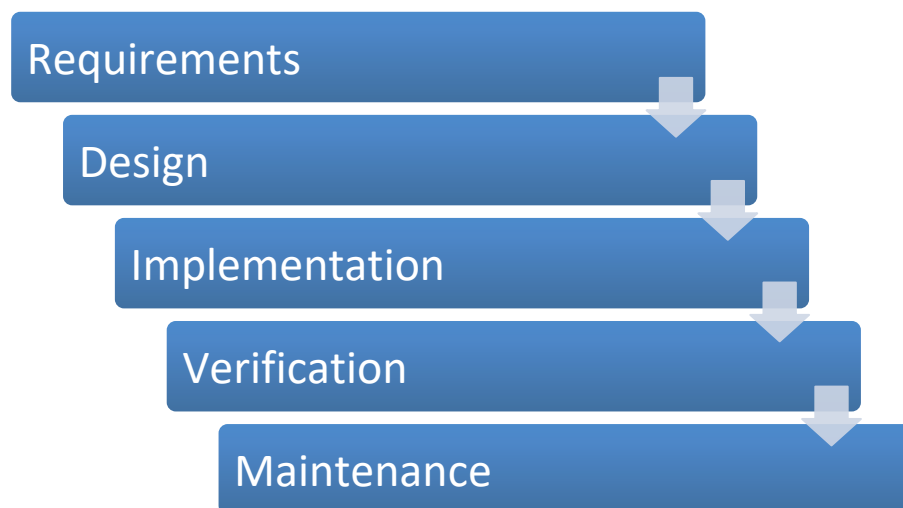
The Waterfall Model originated in the manufacturing and constructions industries and is geared towards highly structured efforts where after-the-fact changes are prohibitively costly. This process has been used frequently for government projects. For instance, in the 1990's California started to build a new statewide system for the Department of Motor Vehicles. This effort ultimately failed and the program canceled.

The central concept behind Big Design Up Front methods like Waterfall is that time is spent early on to ensure all requirements and design decisions are complete and accurate. The emphasis is to ensure that

each phase or activity in the model is 100% complete before moving on to the next phase of the program.

The phases below are followed in order and only move to the next one after the current phase is complete:

- Requirements Specifications
- Design
- Construction (or implementation of coding)
- Integration
- Testing and Debugging (validation)
- Installation
- Maintenance

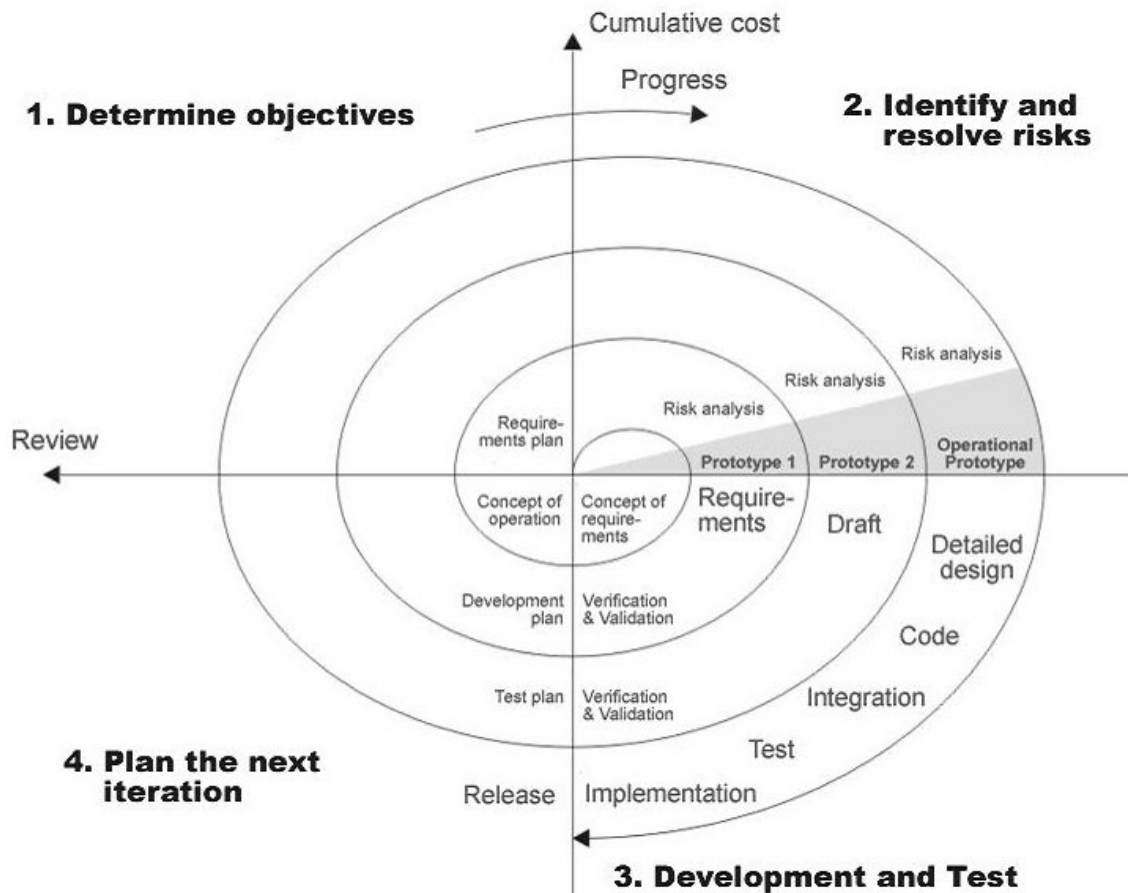


The Spiral Model

The Spiral Model is a process that combines the Waterfall Model with prototyping practices in stages. It was not the first to discuss iterative development but it was the first to explain why iteration matters. Originally spiral iterations were typically between 6 months to two years. Each starts with design goals and finishes with the client reviewing the process. Game development typically followed this model

The U.S. military adopted this process for its Future Combat Systems (FCS) program. After 6 years, the program was cancelled. The program employed a 2 year spiral cycle and should have resulted in 3 prototype stages. This suggested that the Spiral model is better suited to smaller projects (up to \$3 million in size as opposed to \$3 billion)

Spiral models work well where requirements are not well understood and risks are not known. As costs increase, typically the risk decreases. Conversely, it is not well suited for contract work as you do not know the outcome at the beginning of the project and the ability to mitigate risk is only as good as the engineers who are identifying those risks.



Agile Development

Agile software development refers to a group of software development methodologies based on iterative development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams. The term was coined in the year 2001 when the Agile Manifesto was formulated.

Agile methods generally promote a disciplined project management process that encourages frequent inspection and adaptation, a leadership philosophy that encourages teamwork, self-organization and accountability, a set of engineering best practices intended to allow for rapid delivery of high-quality software, and a business approach that aligns development with customer needs and company goals.

Agile methods are a family of development processes, not a single approach to software development.

The Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Agile Principles

- Customer satisfaction by rapid, continuous delivery of useful software
- Working software is delivered frequently (weeks rather than months)
- Working software is the principal measure of progress
- Even late changes in requirements are welcomed
- Close, daily cooperation between business people and developers
- Face-to-face conversation is the best form of communication (co-location)
- Projects are built around motivated individuals, who should be trusted
- Continuous attention to technical excellence and good design
- Simplicity
- Self-organizing teams
- Regular adaptation to changing circumstances

Agile methods

Some of the well-known agile software development methods:

- Agile Modeling
- Agile Unified Process (AUP)
- DSDM
- Essential Unified Process (EssUP)
- Extreme Programming (XP)
- Feature Driven Development (FDD)

- Open Unified Process (OpenUP)
- Scrum

Agile practices

- Test Driven Development (TDD)
- Behavior Driven Development (BDD)
- Code refactoring
- Continuous Integration
- Pair Programming
- Planning poker
- RITE Method

Note: Although these are often considered methodologies in and of themselves, they are simply practices used in different methodologies

Caveats to Agile Development

- Impeded in large scale development projects or teams
- Not well suited for distributed development teams
- Not the best option for mission critical systems where failure is not an option (surgical procedure software)

This page intentionally left blank

Introduction

We've seen several development models at a high level and discussed what differentiates them, and the types of projects they are best suited for. Ultimately the SDLC is a set of repeatable processes to help you deliver your product.

Each development model steps through the same activities, albeit in different ways and with varying results. They should be viewed as a toolset for accomplishing your tasks and delivering what is needed, on time and on budget. Now let's take a journey through a development process and see what it looks like on a closer level.

Your Customer

Yep, this is ~~where the money~~ is where we start our journey. It's all about fulfilling the customer's needs. Understanding those needs is not always easy. Many customers have a vague idea of what they want and you will need to make that happen.

Tom is the CEO for a small, but quickly growing airline called Acme Airways. They want to grow their regional flight coverage by offering a ticket booking system online before the summer travel season. It is March and the season opens in June. The customer wants to be able to take advantage of the lead up so they have a target of May 1st. So you have two months to deliver.

Your customer, like most, will really have only two major concerns besides their big idea:

How much will it cost?

Obviously. Customers will always need to budget their needs with their means. In this case Acme Airways has the cash need to see the project through.

How long will it take?

Revealing huh? Yes, this is another obvious concern. Your customer is almost always planning on the successful completion of your work and likely has many things that depend on your delivering on time.

Tom wants his customers to be able to book a flight, and to leave feedback on the service that they received. These features are pretty robust and will take the full two months to create. So your developer gets to work right away.

The developer spends two months coding frantically, working long nights on caffeine highs and finally delivers. The customer reviews the work and the first words out of his mouth are... "This is not at all what I wanted... you can't search for the latest deals, the design is confusing, and I can't make a payment with my MasterCard!! Now I can't launch in time for the summer travel season!"

Can you figure out how things went wrong? Look at the options below and select the one that most closely matches what Tom wanted the site to do.

The Customer should be able to book a flight

- The Customer enters a departure and arrival date, time and airport.
- The Customer should be able to book flights, choose seating, select meal options, and redeem frequent flier miles.
- The customer should be able to enter departure/arrival dates and times, and see a list of flights with the ability to sort by cost low to high, high to low, layover options, number of connecting flights.

The Customer should be able to leave feedback

- The customer should be able to leave a paragraph or two on their flight service experience.
- The customer should enter a ticket number to log in and be able to select 1 to 5 rankings on various aspects of their flight's service.
- The customer should be able to leave ratings as well as comments on their flight's service and should be randomly selected for a discount award.

If your customer isn't happy, you built the wrong software.

Big Design Up Front (BDUF) or Big Bang software means a lot of work, but very little customer interaction. You generally are not showing the customer much, if anything, until it is done. This approach increases the risk that you **think** you are programming what the customer needs with no real feedback until you **think** you are finished.

No matter how robust your software is, if it does not meet the customer's needs or expectations, it is not what the customer wanted or needed. What would have prevented much of this?



How the customer explained it



How the project leader understood it



How the analyst designed it



How the programmer wrote it



How patches were applied



What the customer really needed

Meeting your goals with ITERATION

Having a good understanding of what the customer needs is crucial to delivering, but employing an iterative development process would have revealed many of the problems early in the effort. This allows you to make adjustments to support the customer's requirement.

Iterative Development Produces Working Software

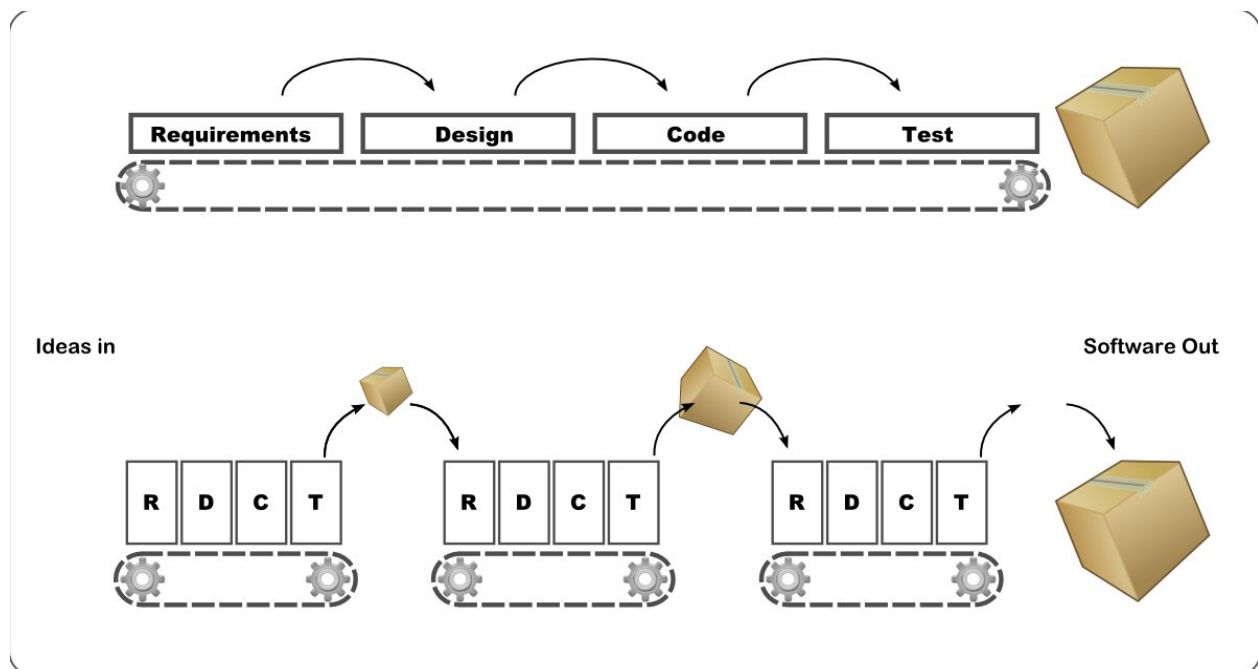
The goal of iterative development is to produce software that builds almost from day one. Developing in smaller cycles or phases means that you are (almost) always able to show your customer small, functioning pieces of code. You should rarely have periods where you have non-working code or builds that won't compile.

Each iteration is a small project

With iterative development you perform all of the same steps from a project **within each iteration**. Each iteration is a small project with its own requirements, design, coding, testing, and other steps. At the end of each iteration you have parts of the overall functionality in a working, buildable state to show the customer.

Each iteration produces QUALITY software

Think of iteration as more than just a way of producing large software. Imagine it as a series of cycles that produce working *quality* software maintaining your customer's vision and increasing your ability to adapt to changes.



The length of iterations can vary with every project. Generally best practices suggest iterations be 20 working days (one calendar month) per iteration. You could have iterations that are longer or shorter but they can lose their effectiveness. If an iteration is too short you will not have the time to complete

larger requirements. If the iteration is too long you run the risk of communication gaps, and losing focus and momentum.

Gathering Requirements

Our previous example was oversimplified and the problem was quite obvious. Unfortunately the world is filled with many more subtle, but project killing variations on this theme.

If you are not sure what the customer wants (and even if you *think* you're sure) go back and ask. You need to keep your customer in the loop throughout the development process. Developing software is NOT guesswork.

Let's go back to Acme Airlines and their booking website. Acme Air is great as a travel provider, but they do not have a development staff to speak of. That's why they hire you. The CEO has provided you with a basic idea of what they want to accomplish:

“We want a website showing our customers our current deals, and to allow our customers to book flights and special packages, as well as paying online. We also want to provide premium shuttle service and hotel accommodations....”

Requirements Exercise

Using the CEO's initial statements above, generate requirements for the project. The first one has been added for you.

<p>Title ... Show Current Deals</p> <p>Description The website will show current deals to Acme Airways users</p>	<p>Title</p> <p>Description</p>
<p>Title</p> <p>Description</p>	<p>Title</p> <p>Description</p>
<p>Title</p> <p>Description</p>	<p>Title</p> <p>Description</p>

Gathering requirements is all about communicating with your customer to turn their vague idea into something more defined. There are always gaps in understanding what the software is supposed to do. When you find you have questions or that you are working on assumptions, you need to go back and talk to the customer until they are resolved.

After identifying the initial requirements you may have more questions. Some questions that might be possible include:

1. Should the software print out a receipts or monthly reports? (And what should be displayed on those reports)
2. Should the customer be able to change a booked flight or cancel altogether?
3. What administrative interfaces (if any) will the software have?
4. What payment types will be accpeted and how will they process?

Can you come up with any questions of your own?

5. _____
6. _____
7. _____
8. _____

Try to gather additional requirements

When getting more information on existing requirements from your customer, try to find out more about any additional requirements that may not have been brought up. You do not want to get to the end of a project only to find that some important detail was missed.

Blue-Sky, Observation, Role-play

When working with your customer in identifying their requirements, **think big**. Brainstorming with a group can produce a healthy set of requirements. Where two heads may be better than one, ten may be even better. Allow everyone to feel that they can contribute and don't dismiss any ideas out of hand as long as it's clear that the focus is fulfilling the customer's needs. This collaborative method of defining and exploring requirements is known as **blueskying**.

The key to developing robust requirements is to include as many stakeholders as possible. If meeting with everyone is getting in the way of producing something useful, have the stakeholders brainstorm individually and meet again to discuss.

Finding Out What People REALLY do

Sometimes the best approaches in gathering requirements are more hands on. Putting yourself in your customer's shoes by **role playing** can help in understanding the finer details of a customer's needs. In some instances, an on-site **observation** of the working conditions and processes can yield a wealth of knowledge in capturing requirements that can be easily overlooked. When using observation

Show Desktop.scf techniques, it is preferable that more than one observer is involved so you can ensure a more rounded set of impressions.

Developing User Stories

Great requirements are written **from your customer's perspective** describing what the software will do for the customer. Any requirement that the customer does not understand is a red flag of something that the customer likely did not ask for.

Requirements will read like a user story, that is, a story that describe how your customer will interact with the software you are creating. To determine if you have a good, effective requirement, use the following guidelines:

User stories SHOULD:

- ...describe **one thing** the software should do.
- ...be written using language the customer understands.
- ...be written by the customer
- ... be short. No more than three lines

User stories SHOULD NOT:

- ...be long or essay-like.
- ...contain technological terms the customer does not understand.
- ...mention specific technologies.

After you have completed your blueskying, gathered your requirements formed them into user stories you should have much more clear understanding of your customer's needs. So what if you have any remaining questions or doubts? **Ask the customer.** You are only ready to move on when you have no more questions and the *customer* is happy that your user stories capture what is needed.

Getting Estimates with Planning Poker

Now you have a set of clear, customer-focused requirements that addressed **what** is being built, the customer is going to want to know **when** it's going to be completed. Your project estimate is the sum total of the estimates for the user stories you developed.

All developers should participate in assigning time estimates in days per user story, even if they are not going to be working on a particular story. The goal is to end up with a set of estimates that **everyone believes in** and can be confident that can be delivered. One method of estimation is called **planning poker**. Let's see how this works.

1. Set a user story in the middle of the table. Choosing one user story helps to focus on initial estimates and any assumptions that might be made. We want a solid estimate so this should include time to design, code, test, and deliver the user story.

2. Everyone is given a set of cards with numbers of days. Cards marked “0” mean “It’s already done”, and a question mark is used when the developer doesn’t have enough information to offer an estimate.
3. Everyone picks a reasonable estimate and places the card face down on the table. This prevents influencing anyone else’s estimates.
4. When everyone is ready, they show their hand giving their honest estimate.
5. The dealer marks down the spread of estimates from the cards and then does a little analysis.

The **larger the spread** in estimates, the **less confident** you should be in those estimates, and the more assumptions you will need to root out. The estimate spread can sometimes be large. What could cause this? Vast spreads in estimates can indicate:

- ...vague, or poorly defined user stories
- ...a misunderstanding of the user story by the developer
- ...lack of confidence or understanding of how to implement a user story
- ...assumptions about the user story

With requirements, **no assumption is a good assumption**. Also be aware that having all of the developers come up with the same estimate *could* indicate that they are all making the **same assumptions**.

You are aiming for as few assumptions as possible. When assumptions appear, even if they are shared by all team members, **expect it to be wrong** until it is cleared by the customer. If any further assumptions remain after this point **they become risks**. Once you have your answers from the customer, play another round of planning poker to make adjustments to your initial estimates.

Big User Story Estimates are BAD User Story Estimates

Suppose your user estimate reaches 40 days for a user story. Remember that we are following the best practices for iterative development and that our iterations are 20 days (or a full calendar month). Now you have a single user story that lasts as long as two project iterations. As a general rule of thumb, estimates that approach or exceed 15 days are **less likely to be accurate**. Two ways to deal with a big user story estimate are to:

Break up the user story

The user story may *actually* contain more than one user story. Try to apply the “AND” rule. Whenever you encounter the word “and” in a user story is potentially a place where you can break it into smaller pieces.

Talk to the customer...again

Seek to clarify any potential assumptions that may be pushing out your estimates with the customer.

The Goal is Convergence

After your round(s) of planning poker, you should not only have estimates, but confidence in those estimates. The goal is to eliminate any remaining assumptions and to converge your estimate figures into one average. Run through the following cycle until you reach consensus:

1. Talk to the customer
2. Play planning poker
3. Eliminate assumptions.
4. Reach consensus

Your estimates are your PROMISE to your customer on how long it will take your team to DELIVER.

Project Planning

Customers want their software when they need it and not a moment later. Often your requirement and estimation efforts will reveal a whole lot of project and a small amount of time. The next step is to work with the customer to **prioritize** each of the features. Staying customer focused, you can offer expert help, but **deciding what's in and what's out is the customer's decision**.

Review the user stories with the customer to decide which ones will be delivered in the "Milestone 1.0". Milestone 1.0 is the **first major release** of the software to the customer and unlike the smaller iteration cycles, this will be the first time you **deliver your software** (and get paid).

Every user story should be given a priority rating from 10 to 50 in increments of 10. Remember, the **customer sets the priority** for what user stories are worked on in which iteration.

Next, check the total estimates for all of the user stories. If they still do not fit in the time to deliver then you should **reprioritize with your customer** for Milestone 1.0 functionality.

You should balance functionality with customer impatience. Help them understand what can be done with the time provided. You are not ignoring those user stories that don't make the first milestone, they are just postponed.

Don't worry about the estimated times when reviewing the user stories. The goal is to identify those features that are the most crucial ones to deliver. Avoid the "nice to have" stories and focus on **what is needed**.

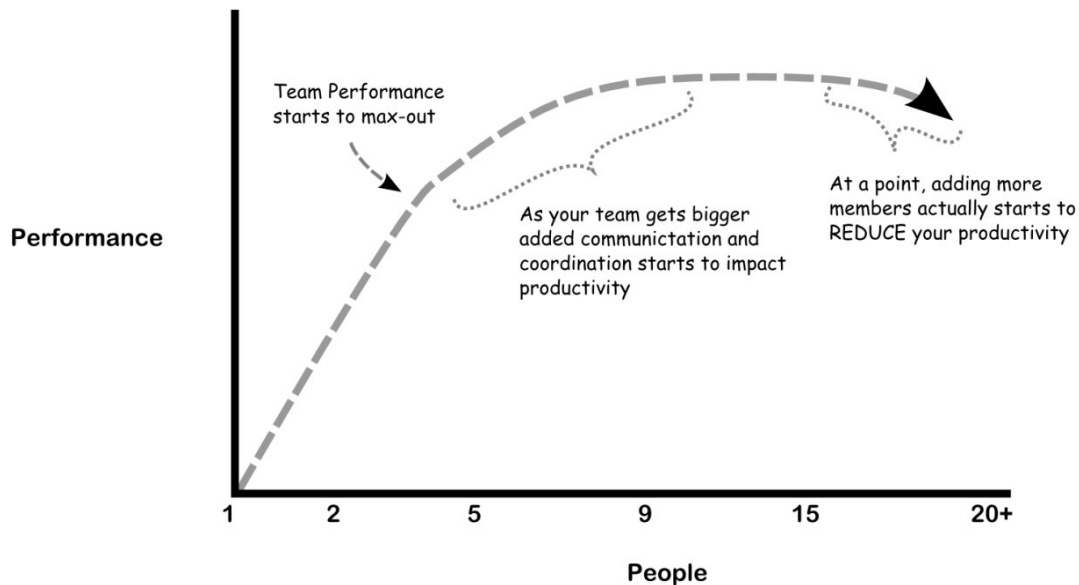
There is more to it than development time

An easy trap to get into is in thinking that doubling the number of developers will cut your estimates in half. This is not really the case. There are a number of things to consider besides productive development time. Other things to consider may include:

- Time for new team members to "get up to speed".
- Understanding the software or technical decisions made in the project.
- Understanding how "it all fits together".
- Time to setup tools and accounts, etc.

Employees cannot be 100% productive 100% of the time. Also common with developers is the "get in the zone" period that happens each day. It takes time for a developer to wrap their head around code and to do something productive with it. All this and we have not even factored in Holidays, vacations, sick leave, or other unforeseen issues.

Care should be taken with adding people to your team. In many circumstances there is a maximum number of team members you can have before you actually start seeing **less productivity**. In any case you should monitor your team to see if they get less productive as you add more people. At that time you should re-evaluate the amount of work you have to do, or the time you have to do it in.



Keep iterations short and simple

Shorter iterations allow your team to respond to change and unexpected details **as they happen**. Short iterations can increase your communication with the customer and provide greater *agility*.

Iterations should balance dealing with change, eliminating bugs, adding features, as well as actual performance of your team members. The other way that you benefit from shorter iterations is that it keeps your team **focused and motivated**.

Adding a dash of reality to your plan

Planning poker provided you with the estimates you need and you are confident in the numbers right? In a perfect world you have what you need. In the real world there is more to account for. Programmers estimate in **UTOPIAN days**, typically providing estimates of **uninterrupted time** to complete a task.

Planning your project needs to include adjustments for impacts to your team's productivity. We have been talking about 20 day iteration cycles since weekends need to be accounted for. Unfortunately (or fortunately if you are a developer) your team members cannot be 100% productive 100% of the time. Sick leave, vacation time, competing projects, and even time to reach full productivity ("getting in the zone") all impact your **true productivity**.

If only there was some way to account for impacts to our productivity.... This is where **velocity** comes in. Velocity is percentage: given a number of days (X), how much of that time is actually productive? There will always be uncertainty in your first iteration. You have to start somewhere and you should consider beginning with a rough estimate of about 0.70, or a productive state 70% of their available time. This helps to account for non-development work such as meetings, phone calls, paperwork, etc.

Why 70%? Well the truth is that you have to start somewhere. As you move through the iterations you will revisit your velocity calculations and make adjustments as you go forward. The goal is to provide a more **realistic estimate** of how long it will take to deliver. So how does it work?

This number will always be **BIGGER** than your days of work to account for non-development time

$$\frac{\text{Days of work}}{\text{Velocity (0 - 1.0)}} = \text{Days required to get the work done}$$

The following is an example of a first iteration calculation. When looking at our iterations we have a set of estimates from our user stories broken up by priority in each iteration. The velocity calculation results in the number of **days an iteration will take your team**.

Iteration 1

$$\frac{55 \text{ Days}}{0.70 \text{ Velocity}} = 79 \text{ Days of Development work}$$

Iteration 2

$$\frac{50 \text{ Days}}{0.70 \text{ Velocity}} = 72 \text{ Days of Development work} = 234 \text{ Days of work}$$

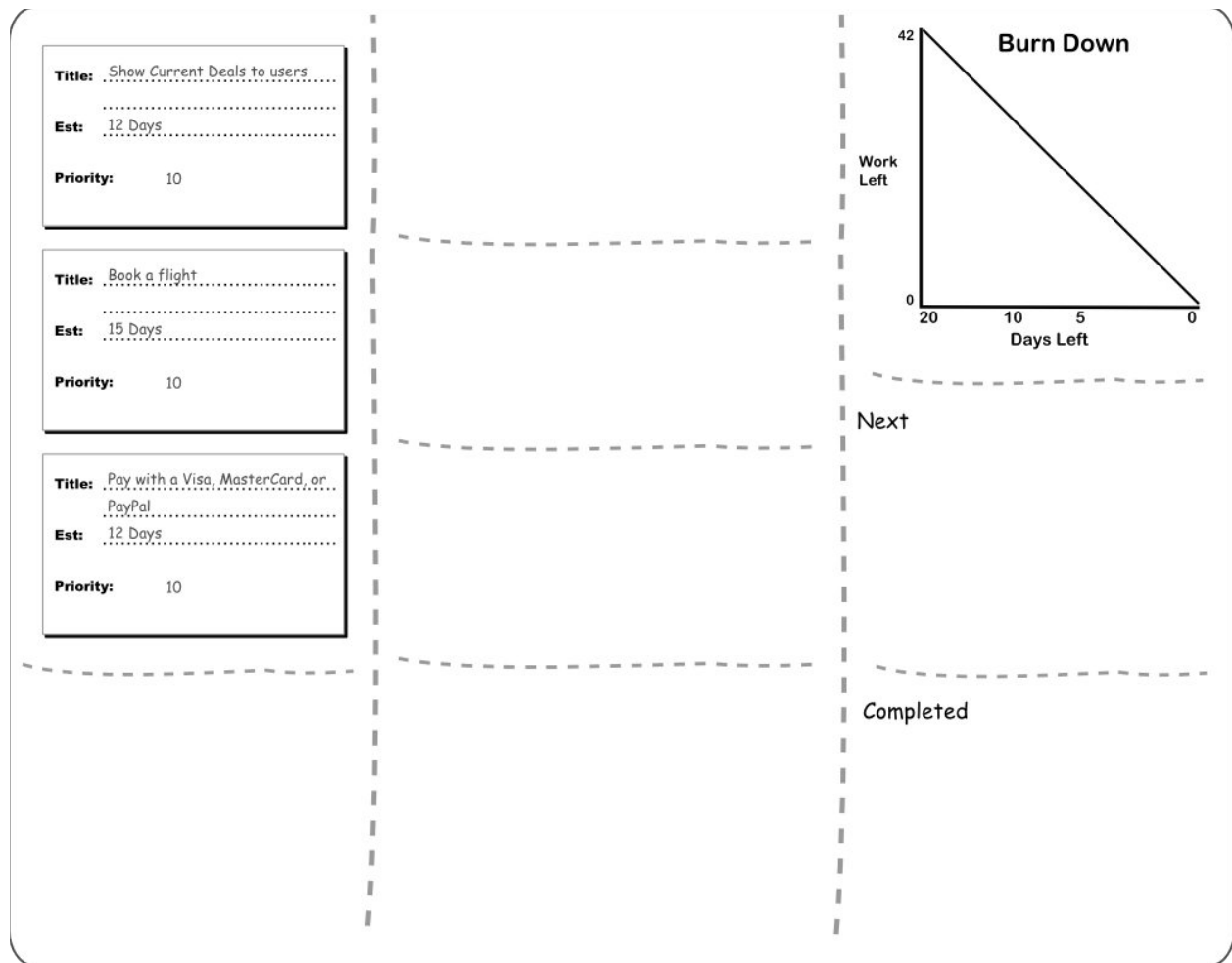
Iteration 3

$$\frac{58 \text{ Days}}{0.70 \text{ Velocity}} = 83 \text{ Days of Development work}$$

So... Three developers would have to work a total of 78 days in three months. But you only have 60 WORKING days.

What can be done to mitigate this? Calculating what your team is capable of **before** planning out your iteration will give you a better picture of what you can realistically accomplish in each iteration (and in the complete project). Then you can accurately determine what you can *really* deliver in Milestone 1.0.

First, apply your team's velocity to each iteration



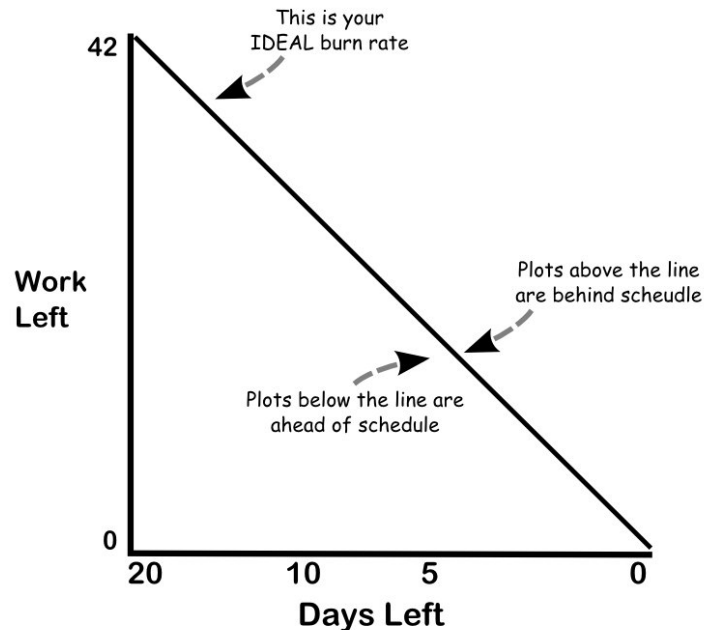
This board is a representation of one iteration cycle. User stories are placed in the left as the pool to be worked on. The below space the user stories is for any unplanned tasks such as a last minute call from your customer wanting a demo with his CFO.

Anytime a developer is working on a particular user story, the card is moved into a “swim lane” indicating that the task is actively being worked on. Each user story should be assigned to a developer by this point.

When tasks are completed they move to the **completed** box on the right. At the end of the iteration, any user stories that are not completed will be placed in the **next** box. (and we hope this never happens!!)

The upper-right corner is reserved for the “Burn Down” chart. This chart provides a visual indication of how the project is proceeding. Let’s look at this chart in a little more detail:

Burn Down



The chart is pretty straight-forward, the **Work Left** axis is measured in total “people-days” of development for all user stories in the iteration. The **Days Left** axis is for time left in the iteration. As the project progresses, the plot line is updated to show what remains in the cycle and how much time and work is left to do.

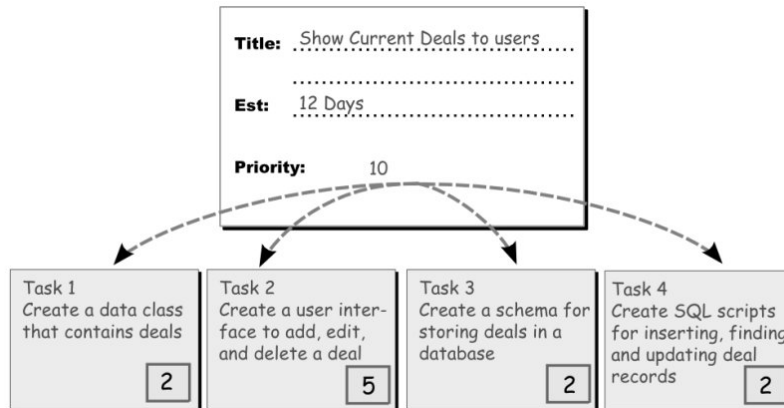
Burn down charts are an effective means of measuring and maintaining your development schedule but at the end of the day customers set user story priorities to be implemented in milestones, and what iteration they will be built.

User Stories, Tasks, and Designs

You have created your user stories to determine what you need to develop and setup an iteration plan. Now it's time to dish out the work. We need to take the user stories and create a set of tasks for bringing it to life.

User stories are simply descriptions of what the software needs to do from the user's perspective. Stories actually contain a number of tasks or small pieces of functionality that combine to make up the user story.

Tasks are treated as specific pieces of development to be created by **one developer**. We need to create task cards with a title, description, and estimate just like we did with the user stories. (Yep, you will likely need to play another round of planning poker to get your estimates)



Creating task estimates may reveal that the sum of those tasks actually take more time than it's user story. Task estimates are usually considered even more accurate and refined than the user stories. If time allows in the initial planning stages, you could mitigate this by performing task estimates when estimating the overall user stories. Breaking user stories into tasks **adds confidence** to your user story estimates. (As well as your project)

Conducting standup meetings

Development models that follow the Agile principles advocate standup meetings. These meetings are intended to be conducted from between 5-15 minutes and for participants to *literally* be standing. This encourages a shorter dialog. These meetings are to be held every morning and require that the following is discussed:

- Tracking of current progress
- Updated Burn Down rate stats
- Status and updates to the tasks
- Talk about yesterday's efforts and outline today's
- Bring up any issues

Stand up meetings are quite different than your typical corporate or project meeting. The focus is on open and organic communication. They are intended to **keep your team motivated** and to **reveal issues early**.

Successful software development is about always **knowing where you stand**. As you better understand your progress and any challenges, you are better able to keep your customer informed and deliver what is needed, on time.

Design Best Practices and Refactoring

Now that we have looked at some of the process techniques, we can start to understand how those help us manage the actual development effort. Good object oriented design follows the **single responsibility principle**. This principle dictates that objects in your system should have only one responsibility and that objects services should only carry out that single responsibility.

Ideally your design should employ these best practices, but often the code is evolving and it's difficult to build a system that is not easily impacted by change. During the coding effort programmers often have to perform a step called **refactoring**.

Refactoring is the process of **restructuring** your code without changing the **behavior** of the code. Refactoring makes the code cleaner, more flexible, more extensible, and is usually a result of improvements to your design. In the long run, these structural changes can save a developer a LOT of time when new functionality has to be introduced. Consider the following example:

```
public double getDisabilityAmount() {
    // Check for eligibility
    if (seniority < 2)
        return 0;
    if (monthDisabled > 12)
        return 0;
    if (isPartTime)
        return 0;
    // Calculate disability amount and return it
}
```

There is nothing really wrong with this code but it is inherently more difficult to maintain. This method is essentially performing two actions: checking the eligibility and then calculating the amount due.

As you may also notice, this breaks the Single Responsibility Principle as well. We really should refactor this code to separate the code that handles the two operations. The new code might look like this:

```
public double getDisabilityAmount() {
    // Check for eligibility
    if (isEligibleForDisability()) {
        // Calculate disability amount and return it
    } else {
        return 0;
    }
}
```

The code performs the same operations, but the calculation happens in a different area. Now if the eligibility requirements need to change but the method for calculating the amount does not, only the **isEligibleForDisability()** method needs to be updated.

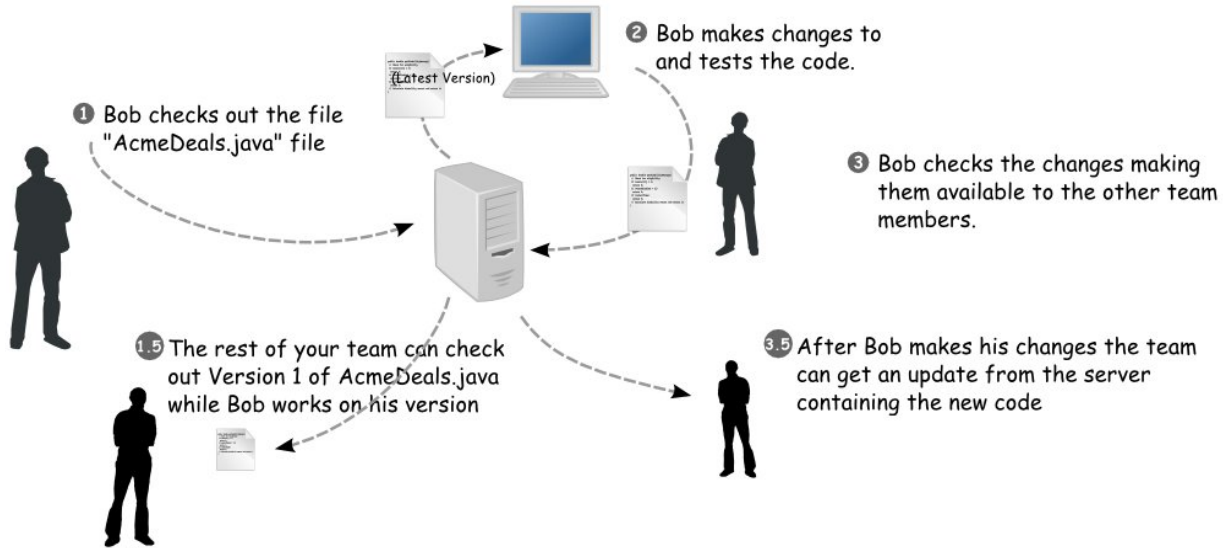
Defensive Development through Version Control

Collaborative coding is a complex thing. You have to make sure your code works and keeps working. A single typo or a bad design decision by a colleague can destroy your hard work. Version controls must be used to ensure your code is **always safe**, allows you to **undo mistakes**, and enables you to make updates for **bugs** on old *and* new versions of your code.

Version control systems are often referred to as **configuration management** tools or **CM** for short. These tools track all changes made by the developers and provide tools to allow developers to see when a change has occurred and what new code has been entered. These views allow the developer to merge

their code with the newly changed code. Some tools will lock files, only allowing one developer to make changes at a time, or will provide the merge analysis.

Interaction with a version control system may look something like this:



Again, some systems only allow one developer to edit a file at a time. The example above is illustrates a more collaborative environment as seen in systems like **Subversion** or **CVN**.

If a developer is attempting to check in his code changes, and another developer has uploaded their own changes, the conflict will be reported and the version control system will show the two versions. It is not able to automatically merge the changes, it is only able to detect them. One such report might look like:

```

public class RemoteReader implements Runnable {
    boolean[] checkboxState = null;
    String nameToShow = null;
    Object obj = null;

    public void run() {
        try {
            while ((obj = in.readObject() != null) {
                System.out.println("got an object from server");
                System.out.println(obj.getClass());
                String nameToShow = (String) obj;
                checkboxState = (boolean[]) in.readObject();
                if (nameToShow.equals(PICTURE_START_SEQUENCE)) {
                    receiveJPEG();
                }
                else {
                    otherSeqsMap.put(nameToShow, checkboxState);
                    listVector.add(nameToShow);
                    incomingList.setListData(listVector);
                    // now reset the sequence to be this
                }
            } // close while
        } catch (Exception ex) {
            ex.printStackTrace();
        } // close run
    } // close inner class
}
    
```

Bob's AcmeDeals.java

```

public class RemoteReader implements Runnable {
    boolean[] checkboxState = null;
    String nameToShow = null;
    Object obj = null;

    public void run() {
        try {
            while ((obj = in.readObject() != null) {
                System.out.println("got an object from server");
                System.out.println(obj.getClass());
                String nameToShow = (String) obj;
                checkboxState = (boolean[]) in.readObject();
                if (nameToShow.equals(POKE_START_SEQUENCE)) {
                    playPoke();
                }
                otherSeqsMap.put(nameToShow, checkboxState);
                listVector.add(nameToShow);
                incomingList.setListData(listVector);
            } // close while
        } catch (Exception ex) {ex.printStackTrace();}
    } // close run

    private void playPoke() {
        Toolkit.getDefaultToolkit().beep();
    }
} // close inner class
    
```

AcmeDeals.java

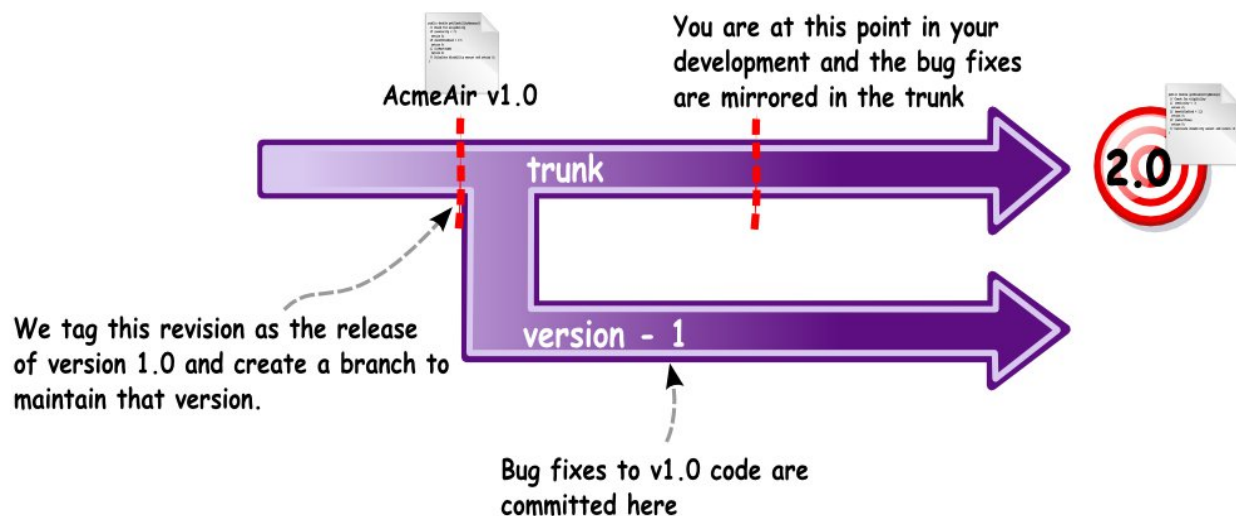
The version control system will attempt to merge any code that does not conflict. Code that does conflict will be shown in a report allowing the developer to resolve the conflicts before the code can be committed back to the system.

Multiple Versions of your code.

Consider the complexities of supporting multiple version of your code. You have successfully delivered your Milestone 1.0 code to the customer. Now your customer has you working towards version 2.0. Your code base contains the 1.0 code mingled with your updated to your 2.0 target and now a bug has been found in version 1.0.

Unless provisions are made in your tools, any fixes you put in code cause you to either release a 1.1 bug fix that has 2.0 code mingled with it, or you have to pull out all of your changes to get your development code back to 1.0, fix the bug, and hope it still builds.

Thankfully version control systems provide the ability to **branch** off your code into definable versions or snapshots. Your code base is typically referred to as your **trunk code**. When you deliver a Milestone, you tag that release, creating a branch as shown below:



It is plain to see that we now have **two codebases** to maintain and work with: the version 1.x **branch**, and the main code base or **trunk**. As you release fixes to version 1.0, you will be releasing version 1.1, 1.2, and so on. At each of those points you are tagging the code and creating a branch so you can get back to that version's starting point in case you need to back out changes.

You may be thinking... "I can see the benefits of tagging and branching, but that could grow very quickly. So how do I consider when to tag and branch so everything is still manageable?" Here are some rules of thumb for deciding when to branch:

Branch when...

- You are releasing a **version of your software** to be maintained **outside of the main development cycle**.

- When you need to **introduce radical changes** to code that may need to be abandoned and you **don't want to affect** the rest of the team.

Do not branch when....

- You can **split code** into different files or libraries that can be built as needed on different platforms.
- You have developers that can't keep their code in a buildable state in the trunk and you want to **sandbox** their efforts.

What version control can do...

- Allows you to **create a repository** keeping your code in a single place for backup and recovery.
- Allows multiple team members **check copies of code out** and work as a team.
- Allows multiple team members **check in changes to the code** and distribute them to other team members.
- Keep track of **who changed what, when, and why**.
- Provides **branches** and **tags** so you can find change version of code from the past.
- Allows you to **roll back changes** that should not have happened.

What version control cannot do...

- Ensure your code compiles.
- Test your code.
- Think for you.
- Ensure your code is readable and designed well.

Automation tools

Developers often rely on **build tools** to automate the process of compiling and performing basic tests on code. These scripts can **create documentation** on your code, and **compile your code**. The build can interact with the version control system reporting on whether code being checked in can successfully build.

These tools provide a uniform way to automate repetitive tasks that developers would have to do manually, reducing the potential for errors. **Build scripts are code**, and as such should also be committed to a version system and treated like any other code.

Testing and Continuous Integration

Anyone who has ever developed code will know what it's like. You have been working hard on that new, cool feature, you have consumed a case of Jolt cola and you're nearly done for the day. You don't know it but you have mistyped a comparison operator (= instead of ==) and when showing your customer the latest build's new feature.... **If fails!!**

The hard truth is that no matter how good a programmer you are, **something will ALWAYS go wrong**. It's not pessimism but rather a reality of life. There are ways to mitigate these sore spots though and that is through testing.

White, Gray and Black box testing

There are essentially three perspectives in testing. Different people will view the software in different ways. Each view is valid and can expose issues that are beyond simple bugs to include intuitive use of the system.

Your users see the system from outside...

Users are not exposed to your elegant code. They do not see the power and flexibility of the database engine or the network management code. They only see a **Black box**, and it either works or it doesn't. User's perspectives are basic and all about **functionality**.

Black Box Testing is about input and output:

- **Functionality**
In other words, does the system do what the user stories say it should? As a black box viewer, you don't worry about where and how the data is being stored, rather you care that the data goes in the way it's supposed to.
- **User input validation**
Does the system explode when you enter -1 as your phone number? How about when you enter letters? Does the search tool allow you to enter an SQL statement and hack the system? The system has to anticipate input that is not valid and had better have a way to gracefully handle it.
- **Output results**
Check that what the system returns is what is expected for each input by the user. This should include the expected response when invalid input is provided by the user. Do you show an error message that the user didn't enter the minimum number of password characters and then allow them to retry?
- **Transitions in State**
While this is similar to checking output results, it is useful to have a table of states and how you expect the system to move and behave from state to state. Does the PeopleSoft ticket allow you to make changes after you close it?

- **Boundary cases and off-by-one errors**

What does your system do when you have 12 options to choose from and you enter 12 and it fails? Did the developer use a zero based array? (0-11 for 12 items rather than 1-12)

Testers are able to get a view under the hood...

Testers also are looking at functionality but they are also looking to **verify what is happening** underneath the bells and whistles. They generally are looking at whether certain values are in the database, that network connections and ports are working, etc. To the tester, your system is more of a **gray box**.

Gray Box Testing is like Black Box but you get to peek:

- **Verifying auditing and logging**

If you are dealing with money or very important information there are usually audits and logs involved. This information is not generally available to the end user and you may need to work with the database directly to be sure the data is right.

- **Data destined for other systems**

Systems that pass data to external systems need to be verified as sending the proper data, sending in across the proper connections or in the right format.

- **System-added information**

Many times a system relies on generated checksums and hashes to ensure data is not corrupted, or that timestamps are in the proper time zones, etc.

- **Scraps left laying around**

This is simple house cleaning. Is the system dumping data it is no longer using? Has it cleaned up memory, or temporary files. In many cases these could be resource destroying or security issues.

Developers have the keys to the kingdom...

Developers can see the design patterns, the classes, duplicate code, or inconsistencies of a system. They see it all... it is a white box to them. Sometimes the level of detail is so high that it's possible for developers to be blinded to broken or ineffective functionality and even come with their own assumptions about the system that users and tester may not.

White box testing uses insider knowledge:

- **Testing all of the various branches of code**

You're seeing *all* of the code. You see the conditional statements, the evaluations of variables, the calculations and operators.

- **Proper error handling**

When you use invalid data with a class's method, do you get the appropriate error? Do you provide the proper feedback to the user when an error is encountered so troubleshooting can

proceed?

- **Working as documented**

If your documentation states that passing a null value to a database field is allowed, does it continue to work and not crash? If the process is said to be multi-threaded, is it?

- **Proper resource handling and constraints**

If your code is supposed to connect to a database and it is not available what does the code do?

Automated testing principles and tools

Black box testing is often performed code on code. The use of **automated testing tools** means that you can test **more of your code, more often, and in a consistent way**. It frees up team members and allows for a repeatable set of tests.

The benefits are plenty, so what's the catch? Well YOU have to build the **library of tests**. There are frameworks that can automatically build tests but they are VERY specific in what they test and make a number of assumptions about how your software works. While helpful, it does not eliminate the need for you to build a library of tests.

Regression testing

The initial build of your **test suite** can be a huge effort, especially if you are creating a new suite for existing code. Your completed test suite can then be run with a single command and can be used by the entire team. You also get **regression testing for free**. Changes to your code can introduce conflicts with old code creating new bugs. Because you took the time to build your library of tests, you can immediately see if an old feature that passed before, fails after adding new code.

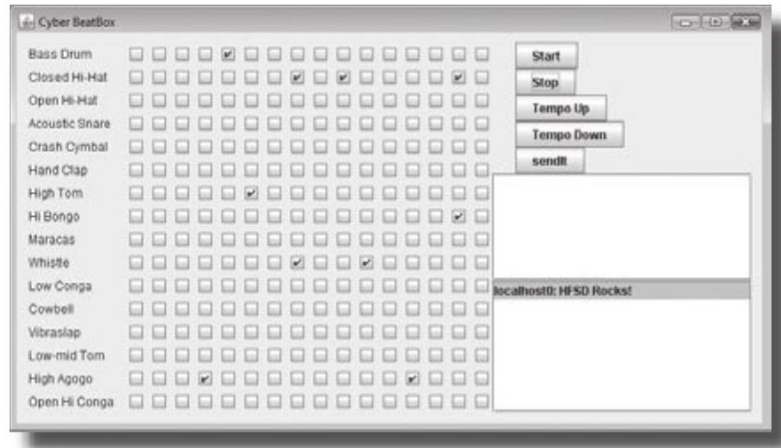
Segment your test suite

Large test suites typically take a long time to run, and can be difficult to maintain. Breaking up your library into smaller test suites increases the likelihood that they will be run more often and allows you to run only the tests you need if you have a specific area to test.

Testing Frameworks

There are numerous testing frameworks for every language out there. There's a whole family referred to as **xUnit**. One flavor of **xUnit** is **JUnit**, a testing framework for Java. Others include PHPUnit, ASPUnit, etc.

Let's look at a test script that is intended to be used under the JUnit framework. The following code is for a program called **BeatBox**. BeatBox is a multi-player drum machine that lets you send messages and drum loops to other users over a network.



The following jUnit testing code simulates a user checking a couple of boxes and sending that updated sequence to users:

```
import java.io.*;
import java.net.Socket;
import org.junit.*;

public class TestRemoteReader {
    private Socket mTestSocket;
    private ObjectOutputStream mOutputStream
    private ObjectInputStream mInStream

    public static final boolean[] EMPTY_CHECKBOXES = new boolean[256];

    @Before
    public void setUp() throws IOException {
        mTestSocket = new Socket("127.0.0.1",4242);
        mOutputStream =
            new ObjectOutputStream(mTestSocket.getOutputStream());
        mInStream =
            new ObjectInputStream(mTestSocket.getInputStream());
    }

    @After
    public void tearDown() throws IOException {
        mTestSocket.close();
        mOutputStream = null;
        mInStream = null;
        mTestSocket = null;
    }

    @Test
    public void testNormalMessage() throws IOException {
        boolean[] checkboxStat = new boolean[256];
        checkboxStat[0] = true;
        checkboxStat[5] = true;
        checkboxStat[19] = true;
    }
}
```

```

    mOutputStream.writeObject("This is a test message!");
    mOutputStream.writeObject(checkboxState);
}
}

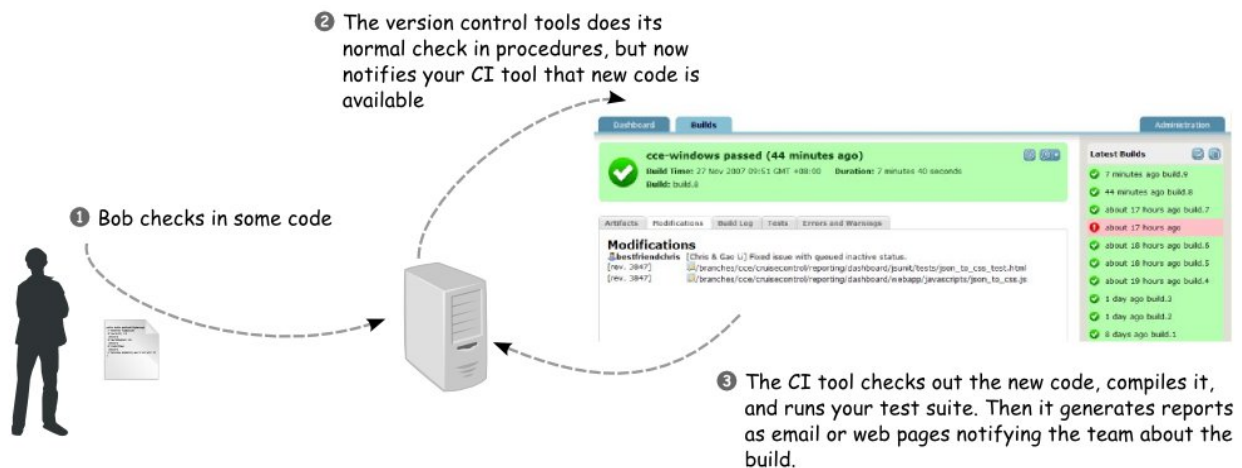
```

The sections marked **@Before** and **@After** setup are run, well...., before and after the tests! They setup any conditions or dependencies such as network connectivity, database interaction or other tools, then those tools are shut down to release the resources in the system.

The actual testing starts at the **@Test** section. This area is where all of your tests are written. In this case, an array of variables is defined for all of the checkboxes. Then three of those check boxes are marked as checked. A message is then sent to the IM window, followed by the sending of the checkbox states.

Continuous Integration concepts and tools

We now have a version control system that keeps track of our code, and we have a set of automated tests. We can now tie the systems together in a way that lets our version control system compile our code and run our automated test. We can even have it email our team with reports when code is committed to the system. This is all part of **continuous integration** or **CI**. Continuous integration wraps version control, compilation and testing into a single, repeatable process.



Code Coverage

When building your tests you need to think about the **percentage of code** you are testing and **not the number of tests**. A hundred tests that all test the same 50-line method is a 100,000 line codebase isn't going to give you confidence in your code.

Good testing takes a lot of time. In general, it is not practical to hit 100% code coverage. There are diminishing returns after a certain point. Most development projects more realistically aim for between 85% - 90%.

You should monitor your coverage levels while tracking your bug reporting. If you are seeing a higher number of bugs than you are comfortable with then you should increase your coverage expectations by 5% or so.

The easiest code to miss in testing has lots of branches, or several conditions that are happening. Your tests need to test every branch. Consider the following login code:

```
public class ComplexCode {
    public class UserCredentials {
        private String mToken;

        UserCredentials() {
            mToken = token;
        }
        public String getUserToken() { return mToken; }
    }
}

public UserCredentials login(String userId, String password) {
    if (userId == null) {
        throw new IllegalArgumentException("userId cannot be null");
    }
    if (password == null) {
        throw new IllegalArgumentException("password cannot be null");
    }
    User user = findUserByIdAndPassword(userId, password);
    if (user != null) {
        return new UserCredentials(generateToken(userId, password,
            Calendar.getInstance().getTimeInMillis()));
    }
    throw new RuntimeException("Can't find user: " + userId);
}

private User findUserByIdAndPassword(String userId, String password) {
    // coe here only used by class internals
}

private String generateToken(String userId, String password, long nonce) {
    // utility method used only by this class
}
}
```

You only need one test for the UserCredentials class code

You will need several tests for this method.

One with a valid userID and password.

One with a null userID

One with a null password

One with a valid userID but invlaid password

One with a valid password but invalid userID

And you have these to consider as well

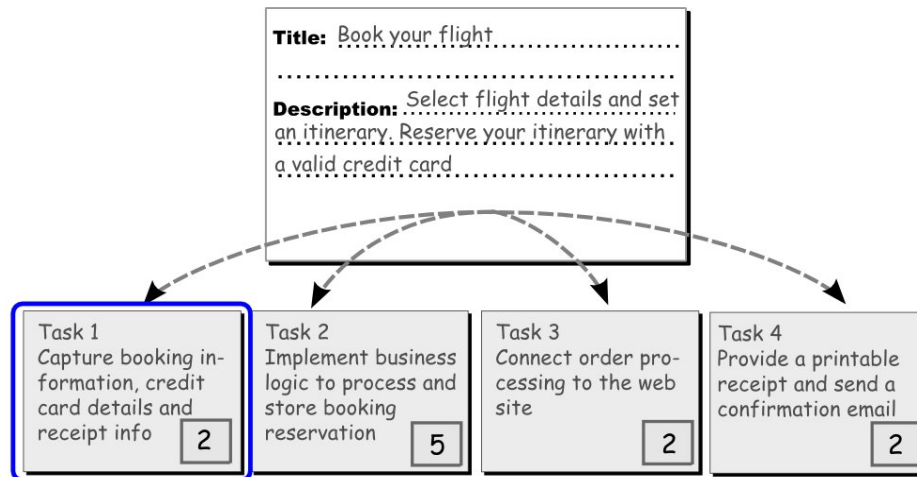
Getting good coverage is not always easy. There are some things that are just inherently hard to test. User interfaces are not impossible to test in automation, but there may be aspects of the user interface that are not tangible to the code. Things like graphical programs or audio require that you have a user test them.

Test Driven Development (TDD)

Even with Unit testing, you will have code that goes untested. But what if your tests were a basic part of your software development to begin with? If you approach testing *while* you are developing, you can increase the coverage of your code, improve your refactoring, and feel more confident in fixing bugs or even re-implementing parts of your software.

It is very difficult to apply comprehensive testing suites to software projects retroactively. When approaching a new project, you are in the prime time to consider the Test Driven Development practice. The concept is pretty straight-forward. You **write your tests first**, and then **code to pass those tests**. So **initially, your test always fail**.

The next step is to refactor, or restructure your code so that you **get to green**. Getting to green is a phrase that refers to the state where you tests show as passing in your unit tests. First we need to analyze the task from our user story:



Let's break down Task 1. For this task you will need to

- **Represent the order information.** Capture the customer's personal details, the flight number, the departing and arrival dates, times, and airports.
- **Represent the credit card information.** You will need to capture the credit card number, expiration date and other processing data.
- **Represent the receipt information.** You need to capture the approval information, reservation confirmation details, etc.

Write the test BEFORE any other code

Now you write your tests and load them into your testing framework. Obviously your initial test fails and this is the point. Welcome to Test Driven Development. With your tests in place you then **write the simplest code possible to pass** the test. Ideally you:

- Write a test
- Run the test and fail
- Write the simplest code to pass that test
- Run the test and pass (get to green)
- Write the next test and start over

You have to resist the urge to add anything that might be needed in the future. Your focus is on **small bits of code** and you should approach all other urges with a YAGNI attitude (You Ain't Gonna Need It). This is the heart and soul of test driven development.

In TDD, your tests drive your implementation. Each test should verify ONE THING ONLY (remember the SRP or Single Responsibility Principle?) You also try to avoid duplicating your test code just like you avoid duplicated production code.

When developing in a TDD world you are always trying to implement the simplest code possible to make your tests pass. In the real world all code has **dependencies**. Your system may rely upon network connections, or databases.

When testing our booking code you don't want to implement an entire database system just to test a couple lines of code. You need to find a way to **make your code independent** for your tests. When this happens you could use **mockup test objects** to simulate needed functionality. These tools allow you to **simulate the dependency**.

Ending an Iteration

Your iteration is almost complete you have your user stories stacked up in the complete section of your Big Board, and everyone is ready for the break in the cycle. So what do we have so far?

- Customer driven functionality
- Code that will build / compile
- Builds that are monitored by our CI tools
- Continually tested code
- Comprehensive coverage of our code
- A reliable way to track our progress
- A pace that adapts to our team

That is a pretty complete list. If we have actually completed our iteration's goals a little early should we just start working on user stories that are due in the next iteration? No. Best practices dictate that we keep the work **within** their iterations. You risk introducing new problems and adversely effecting your ability to effectively manage your iterations.

Still the extra time you may have gained is not lost or non-productive. There are a number of other activities and considerations like:

- Improvements to our process
- System Testing (no this is NOT unit testing)
- Refactoring of code as we look at what we have learned in this iteration
- Code cleaning for readability and documentation updates
- Updates to our tools and development environment
- R&D on new technologies
- Personal development/education time

At the end of your iteration cycle you need to be looking to the future as well as the past. Ending an iteration is a time of discovery. So what happens during this period?

1. Prepare ahead of time

Iteration reviews are chances to have your team provide input on how things went during the cycle. It is not a time to lecture, but to actively converse about what when right, wrong, or could have been done differently to make life easier.

2. Approach with a forward-looking attitude

There may have been some bumps and people do need to vent, this is a time to work for proactive changes and to build confidence rather than foster a demoralizing attitude to challenges encountered.

3. Calculate iteration metrics

Review your velocity and coverage from the iteration. Recalculate the task estimates and apply your adjusted velocity and coverage numbers. You may not want to share the specific numbers so you can focus on morale building, but you can talk about whether productivity was up or down.

4. Have a standard set of questions to review

Have recurring discussion topics so there are expectations on what will be reviewed. Questions can be augmented throughout the project, but the goal is to have consistency in the discussion. Some potential questions may be:

- Is everyone happy with the quality of: Work, Documentation, Testing?
- How was the pace of the iteration?
- Was everyone comfortable with the area(s) of the system you worked on?
- Are there any tools or processes that are helping or hurting our efforts?
- Was the process effective?
- Were there any performance problems?
- Were there bugs identified that must be discussed before prioritization?
- Was the testing effective?
- Is the deployment under control and is it repeatable?

Plan for next Iteration

When you get to the end of an iteration you should have a buildable, working piece of software. But complete software is more than code. It is completing iteration work, passing your testing suite, and pleasing the customer.

Before you can start the next iteration you need to go back to the planning phase. Your first step is to revise your user story estimates, task estimates, and your team's velocity. This process is similar to your initial effort in many ways. The difference here is that you have a better understanding of the real challenges to your planning. You may find that your initial velocity of 70% is actually high, and that real productivity is lower.

Lower initial productivity may be a shock to you but those numbers may be impacted by the ramp up time to get to a productive state. Additionally, you should have a clearer picture of your user stories now and anything that may impact their implementation. Your velocity numbers will fluctuate between iterations but its normal and part of the process of being an **adaptive** or **agile** development team.

The start of a new iteration is the one time in the process that you can change your processes. You should NEVER change a process while **in** an iteration. Before starting the next iteration, you should also

communicate with your customer. This is the time to **demo** new functionality, to **communicate** any user stories that may have had to be pushed to the new iteration, and to provide a point to **reprioritize** the user stories for the next iteration.

The ultimate goal is to deliver **working** software at the end of each iteration, and eventually at the end of a Milestone.

Software Maintenance

It's your code, it's your responsibility. Even if you were required to merge code from a third party, you are ultimately responsible for all of it. There are no exceptions. Third party code may be a huge unknown, be laden with bugs and problems, and have little to no documentation, but your customer expects you to deliver.

When bugs show up, your first priority is to contact your customer. **Communication**, even when it's not good news, is paramount. It also builds confidence when your customer sees you as **transparent** as well as **effective** in fixing issues.

Your next top priority is to get your code to a buildable state (if it's not, or if you are merging third party code). You must ensure that your code is **controlled** (version control, CI, unit tests and test libraries) **and buildable** before you make **any** code changes for bugs.

Spike test to estimate

Spike tests are a way to sample a set of bugs and to extrapolate an estimate on the time to repair all of the remaining bugs. For instance, if you have 30% of your tests failing you may not have any idea whether those bugs can be fixed with a couple lines of code or if it will take new class objects and hundreds of lines of code. Spike testing can help with those estimates.

1. Take a week to run your spike test
2. Choose a random sample of bugs to work on.
3. Calculate your bug fix rate. (number of bugs / days = rate)

It's true we cannot be certain how long it will actually take to fix everything, but this method provides a more accurate number than purely guessing. It may not be exact but you are at least attempting to provide an educated estimate.

Spike testing provides you with **quantitative data** to base your estimates on. You know how many bugs you fixed and it was a random sample so we can be fairly confident of the rate. Conversely a spike test does not provide you with **qualitative data**. You only know how fast you can fix the types you just did. We have no way of knowing if it's far worse with other bugs (or better if you are a glass half full type).

Spike tests help, but you can (and should) also factor in your team's confidence levels in providing fixes. Once you have your estimates, you need to **contact the customer**. All bugs are worked on in the same fashion as a user story. In fact, the **bug becomes a user story** of a fix to the functionality. It will travel on the Big Board like any other story.

Priorities for the bug fixes are always set by the customer. It may be necessary to reprioritize the user stories and some may even get bumped to the next iteration, that is normal. You simply need to consider your burn rate when adjusting for bugs.

Tracking bugs

All bugs should be entered into a formal tracking system like Bugzilla, TestTrackPro, or ClearQuest. These tools allow a collaborative way to track, prioritize and document bugs. Regardless of what tool you use, make sure you:

1. **Record and communicate priorities**

You can work these into your Big Board by grouping all bugs of a certain priority level and turning them into user stories to be worked on. These user stories will need their own priority level just like any other user story.

2. **Keep track of EVERYTHING**

Bug trackers are good at maintaining histories of conversations, tests, code change reports, and other information about a particular bug.

3. **Generate Metrics**

Trackers can give you insight on things like rate of bug submission, if bug reports are increasing or decreasing, how many bugs remain and if they are being actively worked on, and more.

Bug reports should contain:

- **A summary** describing the bug in detail, including any error messages or behaviors that the system presented. You should be able to read it and fully understand what the problem is.
- **Steps to reproduce** should describe how you got this bug to happen. You may not remember every little step, but there should be a clear set of motions listed that allow the reader to attempt the same steps to verify the bug.
- **What you expected to happen and what really happened.** This is particularly helpful in revealing problems with user stories or requirements problems where the user expected something that the developer didn't know about. It can also reveal problems with the intuitive nature of the software's usage.
- **Version, platform, and location information.** Is your application web based? What was the URL used? Are you running the latest build? Are you a Mac or PC?
- **Severity and priority.** How bad did this impact the usage of the system? Is there data corruption? Is it just annoying, or will someone actually die as a result? Does it happen only on the seasons of the moon?

Pinning down a software development process

A software development process is a structure imposed on the development of a software product.

- Wikipedia

That is a pretty vague description considering the model we just walked through. It does not rigidly set out a definition involving iterations, or best practices. Ultimately, there is no silver bullet process in software development. There are only tools to move you through your process. Those tools can even change to match the latest project you are working on. A great software process is one that lets YOUR team be successful. You should always consider the following:

- **Develop iteratively.** Time has shown that large, rigid development processes are risky and prone to failure. Breaking into iterations allows for more communication and adaptability.
- **Always evaluate and assess.** No process is perfect and even when everything is working really well, your project will change. You need to incorporate ways to evaluate your effectiveness and be able to adapt.
- **Incorporate best practices.** Following the pack, or jumping on the latest process bandwagon is not the way to succeed. Be critical but fair about other approaches to problems and adapt to other processes when it's clear that you can benefit from them. Maintain a healthy level of **process skepticism**.

This page intentionally left blank

After all is said and done.....

Development Techniques and Principles

The following techniques and principles appear in the book *Head First Software Development*, by Dan Pilone & Russ Miles, 2008 O'Reilly Media, Inc. ISBN: 978-0-596-52735-8. Much of this course is based on this book.

These techniques and principles (principles are italicized) provide a quick snapshot of the salient points discussed in this course. They should prove as a useful reminder on the topic of Software development Life Cycles.

In reality, this course serves as a very brief introduction to the SDLC. There is an abundance of resources available online and in formal education for those who wish to study this topic further.

Great Software Development

- Iteration helps you stay on course
- Plan out and balance your iterations when (not if) change occurs
- Every iteration results in working software and gathers feedback from your customer every step of the way
- *Deliver software that's needed.*
- *Deliver software on time.*
- *Deliver software on budget.*

Gathering requirements

- Blue-sky, Observation, and Roleplay to figure out how your system should behave.
- Use user stories to keep that focus on functionality
- Play "planning poker" for estimation
- *The customer knows what they want, but sometimes you need to help them nail it down.*
- *Keep requirements customer-oriented.*
- *Develop and refine your requirements iteratively with the customer.*

Project Planning

- Iterations should ideally be no longer than a month. That means you have 20 working calendar days per iteration.
- Applying velocity to your plan lets you feel more confident in your ability to keep your development promises to your customer.
- User (literally) a big board on your wall to plan and monitor your current iteration's work.
- Get your customer's buy-in when choosing what user stories can be completed for Milestone 1.0, and when choosing what iteration a user story will be built in.
- *Keep iterations short and manageable.*
- *Ultimately, the customer decides what is in and what is out for Milestone 1.0.*
- *Promise and deliver.*
- *ALWAYS be honest with the customer.*

User stories and tasks

- User stories describe one thing and one thing only
- User stories are prioritized by the customer
- User stories are broken down into tasks. Both have estimates for delivery effort

Version Control

- Use a version control tool to track and distribute changes in your software to your team.
- Use tags to keep track of major milestones in your project (ends of iterations, releases, bug fixes, etc).
- User branches to maintain a separate copy of your code, but only branch if absolutely necessary.
- *Always know where changes should (and should not) go.*
- *Know what code went into a given release and be able to get to it again.*
- *Control code change and distribution.*

Building you code

- Use a build tool to script building, packaging, testing and deploying your system
- Most IDEs are already using a build tool, and you can build on what the IDE already does.
- Treat your build script like code and check it into version control.
- *Building a project should be repeatable and automated.*
- *Build scripts set the stage for other automation tools.*
- *Build scripts go beyond just step-by-step automation and can capture compilation and deployment logic decisions.*

Testing and continuous integration

- There are different views of your system, and you need to test them all.
- Testing as to account for success cases as well as failure cases.
- Automate testing whenever possible.
- Use a continuous integration tool to automate building and testing your code on each commit.
- *Testing is a tool to let you know where your project is at all times.*
- *Continuous integration gives you confidence that the code in your repository is correct and builds properly.*
- *Code coverage is a much better metric of testing effectiveness than test count.*

Test-driven development

- Write tests first, then code to make those tests pass.
- Your tests should fail initially; then after they pass you can refactor.
- Use mock objects to provide variations on objects that you need for testing.
- *TDD forces you to focus on functionality.*
- *Automated tests make refactoring safer. You'll know immediately if you've broken something.*
- *Good code coverage is much more achievable in a TDD approach.*

Ending and iteration

- Pay attention to your burn-down rate. Especially after the iteration ends.
- Iteration pacing is important. Drop stories if you need to keep it going.

- Don't punish people for getting done early. If their stuff works, let them use the extra time to get ahead or learn something new.
- *Iterations are a way to impose intermediate deadlines and stick to them.*
- *Always estimate for the ideal day for the average team member.*
- *Keep the big picture in mind when planning iterations. That may include external testing of the system.*
- *Improve your process iteratively through iteration reviews.*

Bugs

- Before you change a single line of code, make sure it is controlled and buildable.
- When bugs hit code you don't know, use a spike test to estimate how long it will take to fix them.
- Factor in your team's confidence when estimating the work remaining to fix bugs.
- User test to tell you when a bug is fixed.
- *Be honest with your customer, especially when the news is bad.*
- *Working software is your top priority.*
- *Readable and understandable code comes a close second.*
- *If you haven't tested a piece of code assume that it doesn't work.*
- *Fix functionality.*
- *Be proud of your code.*
- *All the code in your software, even the bits that you didn't write, are your responsibility.*

The real world

- Critically evaluate any changes to your process with real metrics
- Formalize your deliverables if you need to, but always know how it's providing value.
- Try hard to only change your process between iterations
- *Good developer develop – Great developers ship.*
- *Good developers can usually overcome a bad process.*
- *A good process is one that lets YOUR team be successful.*