Agile

Andrew McNeil

Software Engineer

March 21, 2019

Introduction

I made this PowerPoint using only information that I have learned from reading books on Software Development. Some of these topics include

- Requirements
- Design
- Test Automation
- Continuous Integration
- Continuous Delivery
- Scrum
- Organizational Structure for Agile

How much upfront requirements should be complete at any point in time?

Requirements are fleshed out just in time for teams to start working on functionality to support the requirement[7,8,12,17]

Requirements are emergent, constantly evolving from client feedback and what you learn about as you work on the project[14,17]

“Clients don’t know what they want. They know what they don’t want once you make it for them.” –Jez Humble

Design

How much upfront design should be done before coding?

Some guesses

- High-level design and low-level design
- High-level design only
- No design before coding

Design

Only do high-level design before coding[1,3,4,5,6,7,8,11,14,16]

When doing low-level design prior to coding, your brain is full of little holes[1]

As you code up a solution, you learn more about the problem[1]

“You can’t create complete requirements or designs up front by simply working longer and harder. Some requirements and design will always emerge once product development is under way; no amount of comprehensive up-front work will prevent that.”[8]

Changes in design

Design is expected to change and evolve based on what we learn by trying to implement it[14]

Low level design is constantly evolving throughout a project with refactoring[1]

- Any time a developer is doing a coding task, they can refactor existing code if it will help their current task[1]
- Due to frequent changes in low-level design from refactoring, there should be no formal documentation of low-level design[5]
  - We should let our unit tests act as our low-level design documentation[3]

When should automated tests be written?

Test Type	When to write
Front-end UI VR Test	Prior to implementation if possible, immediately after implementation otherwise[5,10,13]
Back-end Functional Test	Prior to implementation[2,5,7,10,13]
Back-end Unit Test	Prior to implementation[2,5,7,10,13]

"Code that isn't tested doesn't work. If spending time creating acceptance tests before coding slows anything down, it is only slowing the rate at which we are creating code that doesn't work."[3]

The later that automated tests are written, the harder and less efficient it is to write automated tests[9]

How much of each test type should we write?

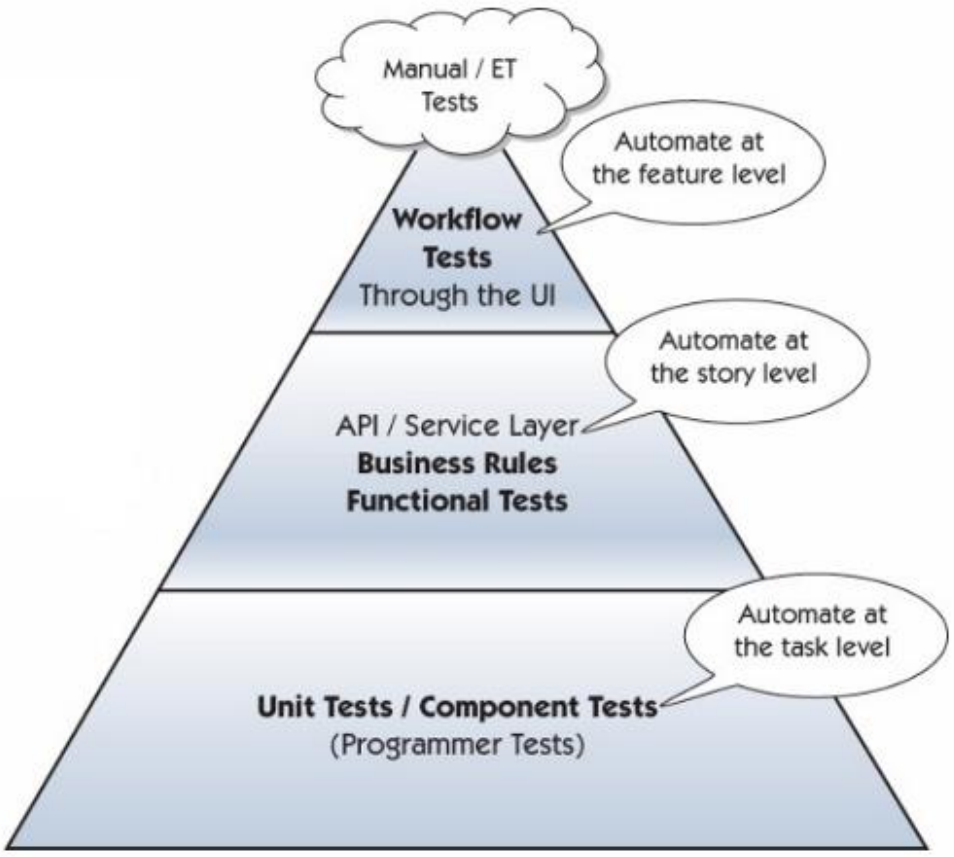

Figure 8-7 Automation pyramid

Maintain One Version of Code in Production

A complication with client installed software is the many different issues that clients may have[10]

- This creates a nightmare for support[10]

We should aim to keep clients on the same version-the latest version[6,10]

We should make the upgrade process as painless as possible. Some options:

1) Have the software check for new versions and prompt the user to upgrade the latest version[10]
2) Download in the back-ground and silently upgrade the next time the application is restarted[10]

Maintain One Version of Code in Production

In the case of upgrading, clients commonly defer the option of upgrading in fear that upgrading will break the application[10]

- If the upgrade process is flaky, then the client is correct to never upgrade[10]
- If the process is not flaky, then there is no point in providing a choice to the user[10]
- Giving the user a choice tells them that the developers have no confidence in the upgrade process[10]

The correct answer: Download in the back-ground and silently upgrade the next time the application is restarted[10]

Canary Releasing

Canary Releasing is a technique to roll out new features to production to a subset of users for testing prior to rolling the version out to everyone[10]

A team might think of accomplishing this by maintaining two versions of software-one version with the new feature and one version without the new feature.

However, we should be able to send the same version of software to all clients, with the functionality being hidden for some clients but not others[10]

- We can accomplish this with feature toggles[10]

Feature Toggles

Can be used to turn features on or off for particular users or facilities without requiring a production code deployment[10]

Typically done by using a configuration file to store properties that determine which features are visible[10]

For example, we may choose to have a new button visible for some clients but not others

Branching

- There is a non-linear relationship between lifespan of branches and integration problems[14]
- There is also a non-linear relationship between the number of branches and integration problems[14]

These issues are both solved by Continuous Integration and Pair Programming[14]

Continuous Integration

- Merging all working developer copies to a shared mainline at least once per day is known as Continuous Integration[5]
- Each developer should aim to merge code to master/trunk multiple times per day, once per day minimum[5,6,10,13,14,15]
- Development branches should be short-lived (merged in less than one day after creation)[10,13,15]
- In order to minimize context switching, developers should never have more than one active branch[10]
- Master should always be releaseable[10]
- Tasks developing incomplete features should still be merged to master. Feature toggles are used to hide all incomplete features in production while making it visible in development[10]

Continuous Review

- All code should be approved by at least one teammate before committing to master[14]
  - In addition to having one approval prior to committing to master, someone may optionally review after committing to master[14]
- Difficult tasks should be pair programmed[5,6,7]
- With pair programming, the partner counts as the second pair of eyes and the task is merged without a formal team-wide review[14]
- When not pair programming and your code is ready to be reviewed, call a teammate over to your desk to review your code with you before merging (Jez Humble on twitter… I tweeted him this question)

Characteristics of high delivery performance

Research was done on thousands of software organizations on delivery performance.[14] Teams with the highest delivery performance had the following characteristics[14]

- Fewer than three active branches at a time[14]
- Branches with short life times (less than a day)[14]
- No code freezes[14]
- No stabilization periods[14]

Continuous integration and pair programming have both been shown to increase code quality and decreases time to market[14]

“Ask a programmer to review ten lines of code, he'll find ten issues. Ask him to do 500 lines and he'll say it looks good.“[13]

Continuous Delivery

- Deploying software into development, test, or production should take two tasks: picking the version and the environment and pressing the “deploy” button[10]
- In continuous delivery, every code change we make is a release candidate. We may choose to opt in or opt out of releasing the code change[10]
- If something goes wrong during the release, we should be able to revert the release in seconds[10]

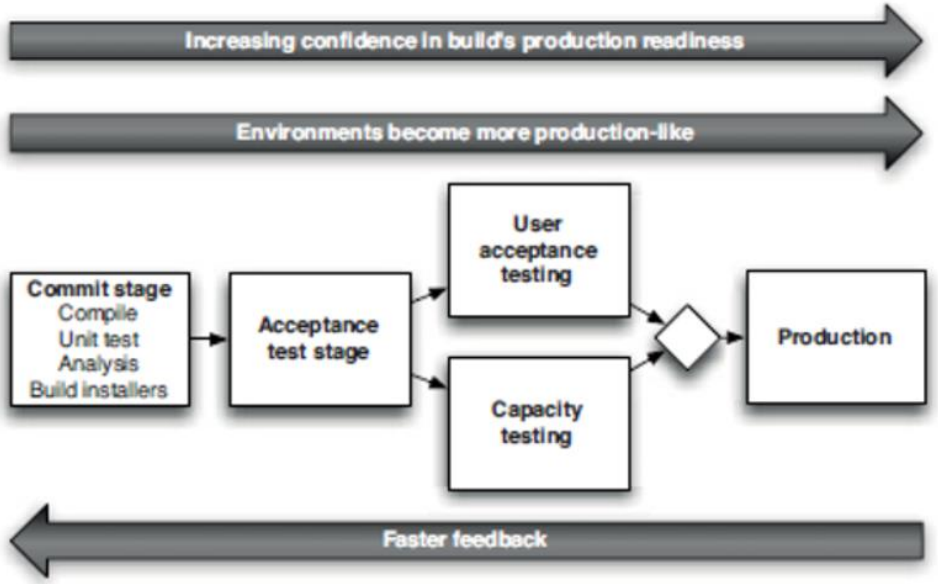

Figure 5.3 *Trade-offs in the deployment pipeline*

Manual Testing in a continuous delivery environment

Contrary to popular belief, manual testing is still necessary to be done prior to each release in a continuous delivery environment. This includes

- Exploratory testing[10]
- User Acceptance Testing[10]

The developer should not begin on the next story until the tester is done testing. This is done to prevent context switching under the event any issues are encountered during testing.[10]

- Instead, the developer should sit with the tester as he/she is testing in case issues are found[10]

Biggest contributors to continuous delivery

In 2017, a study was done on software organizations in an attempt to determine which factors were the biggest contributors to continuous delivery[14]

Surprisingly, the biggest contributor to continuous delivery was not test automation or deployment automation[14]

The biggest contributors to continuous delivery was the ability for teams to

- Complete their work without having to communicate with people on another team[14]
- Make changes to a system without permission of someone outside the team[14]
- Make changes to the a system without needing another team to make a change to their system[14]

The “ultimate configurability”

“It’s almost always better to focus on developing the high-value functionality with little configuration and then add configuration options later when necessary.”[10]

“The desire to achieve flexibility may lead to the common antipattern of ‘ultimate configurability’ which is, all too frequently, stated as a requirement for software projects. It is at best unhelpful, and at worst, this one requirement can kill a project.”[10]

What to do when our project is behind schedule?

Some ideas

- Less attention to quality
- Allocate more resources/add members to team
- Nothing. We will finish later
- Overtime! ☺ ☺ ☺
- Reduce scope

What to do when our project is behind schedule?

Worst answer: Less attention to quality[4,5,6]

Lowering quality lengthens development time[4,6]

For every dollar gained by cutting quality, it costs $4 to restore it[5]

Best answer: reduce scope[5]

- Promotes early feedback[5]
- Doesn't add technical debt[4]

How many projects to work on at a time?

You should aim to work on exactly one project at a time[2,5,8,11,13]

- Working on multiple projects leads to context switching and loss of focus[8,11,13]
- Working on multiple projects reduces the quality of the projects[8]

Should our scrum teams be long lived or short lived?

We should favor long lived scrum teams over short lived scrum teams[5,6,8,11,13]

Teammates generally work better together after they have been together for longer[5,8,13]

It is usually more optimal to keep teammates on a scrum team together through the completion of multiple projects as opposed to re-shuffling the teams after each project[5,13]

Self-Organizing scrum teams

Scrum teams should be self-organizing[8]

A self-organizing scrum team functions as follows:

- Most decisions are made by members of the scrum team[8]
- The scrum team chooses who will do which work[8]
  - Leader does not assign work
- Determine how the work will be done[8]
- People closest to the problem will typically know the most about it[13]

To what extent should each teammate generalize or specialize?

Teammates should posses T-Shaped skills[8,11,12,13]

- Teammates should have deep skills in their preferred functional area[8]
- Teammates should have broad skills in multiple other areas in order to reduce the number of hand-offs within the team and outside of the team[8]
  - The few remaining handoffs should primarily be handoffs to people within the team which are significantly faster and cheaper than handoffs to people outside of the team[11]

Additional Scrum team recommendations

- 5-7 members[8]
- 2-3 week sprints[8]
- Cross-functional[8]

Communication between teams

Teams should be assembled such that the number of communication paths between teams is minimized[5,13]

Teams should be able to independently develop and deploy code to production without requiring handoffs to other teams[13]

Teams should be able to quickly and independently deliver value to the customer[13]

Avoid attending recurring scrum meetings of other teams (retros, demos, planning, etc)[5]

Communication between teams shouldn’t revolve around specific details at the implementation level. It should revolve around discussing higher-level shared goals and how we can achieve those goals[14]

Ticketing Hell

When development and operations are split out to different teams, the process often results in ‘ticketing hell’, having to raise a series of tickets in order to get anything done[10,13]

Solutions

- Embed operations members onto the development team[13,15]
- Assign liaisons between operations and development members[13]
- Create self-service capabilities[13]
  - “It’s okay for people to be dependent on our tools, but it’s important that they don’t become dependent on us.”[13]

Potentially Shippable Product Increments

In scrum, teams create potentially shippable product increments each sprint[5,8]

This means that code is fully

- Tested[5,8]
- Documented[5,8]
- Integrated[5]

Vertical vs Horizontal division

Features generally involve multiple different layers[9]

Scrum asks that teams work on a small portion of each layer with the code fully integrated each sprint[9]

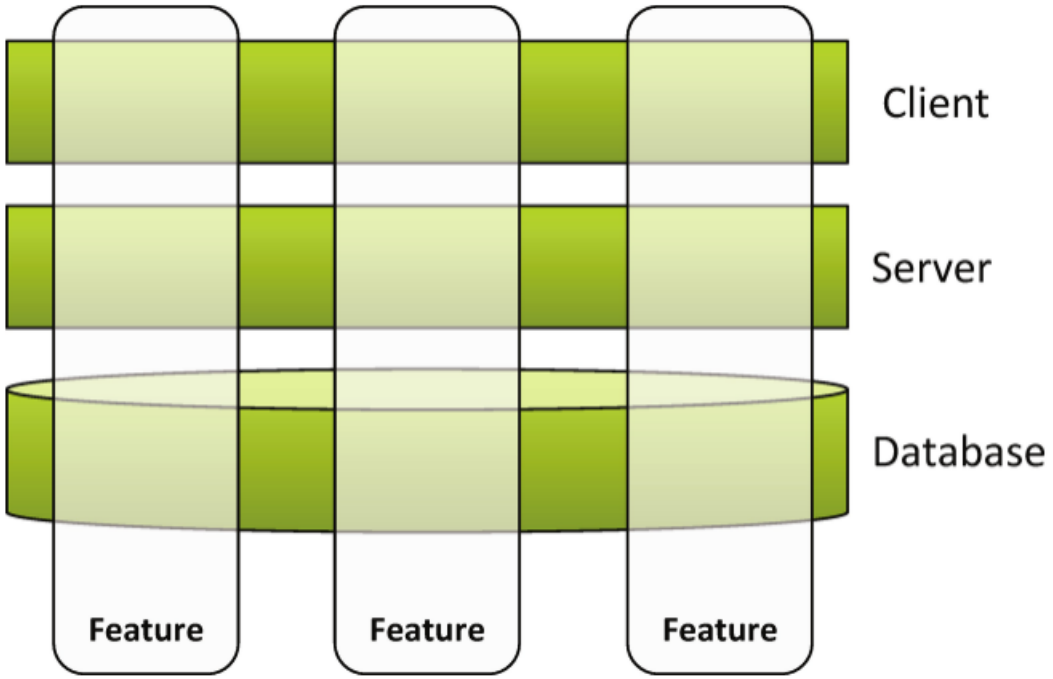

Figure 8-1 Vertical vs. Horizontal division

Feature Teams

Feature teams are cross-functional and teams that can work on end to end features with minimal hand-offs to other teams[8]

A feature team will typically work on multiple components in order to complete the feature[8]

Component Teams

Component teams

- own a particular component[8]
  - Components are re-usable assets providing common functionality needed to implement features
- Form on the belief that a team of experts who are trusted to make safe changes in an area should own that area[8]
- Results in people only working in areas that they are specialized in[8]

Feature teams or Component Teams?

Component teams pose the following problems

- Most features result in code changes in multiple components[10]
- Multiple teams must communicate together in order to complete a feature[8]
- Component teams have to deal with competing requests from other teams while at the same time completing everything by the appropriate time[8]
- Commonly defers integration until the very end due to being on different teams
- Increases the number of failure points from one (feature team) to multiple (number of component teams)[8]
- Longer lead times[13]

Scrum favors feature teams over component teams[8,10,11,13]

Organizations should contain a large number of feature teams and a small number of component teams[8,10]

Summary

- Requirements and design are emergent and are constantly evolving
- Keep clients on the same version of code-the latest version
- Merge all working developer copies into a shared mainline several times per day
- Work on exactly one project
- Develop T Shaped skills
- Minimize handoffs
- Favor organizing around end to end features over organizing around components

Sources

[1] Fowler, Martin. Refactoring: Improving the Design of Existing Code. Boston, MA, USA: Addison-Wesley, 1999.

[2] The Clean Coder: A Code of Conduct for Professional Programmers. Prentice Hall, 2011.

[3] Beck, Kent. Test Driven Development By Example. 2002.

[4] Sterling, Chris. Managing Software Debt, Building for Inevitable Change. Addison-Wesley, 2010.

[5] Cohn, Mike. Succeeding With Agile: Software Development Using Scrum. Addison-Wesley, 2013.

[6] Beck, Kent. Extreme Programming Explained, Embrace Change . Addison-Wesley, 2012.

[7] Cohn, Mike. User Stories Applied For Agile Software Development. Addison-Wesley, 2004.

[8] Rubin, Kenneth. Essential Scrum: A Practical Guide To The Most Popular Agile Process. Addison-Wesley, 2013.

[9] Axelrod, Arnon. Complete Guide to Test Automation. Apress, 2018

[10] Humble, Jez and Farley, David. Continuous Delivery: Reliable Software Releases Through Build, Test And Deployment Automation. Addison-Wesley, 2010.

[11] Narayan, Sriram. Agile IT Organization Design. Pearson Education, 2015.

[12] Gregory Janet, and Cirspin, Lisa.More Agile Testing: Learning Journeys For The Whole Team. Addison-Wesley, 2014.

[13] Kim, Gene, et al. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution, 2016.

[14] Forsgren, Nicole, et al. Accelerate, The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations. IT Revolution, 2018.

[15] Wolff, Eberhard. A Practical Guide to Continuous Delivery. Addison-Wesley, 2017.

[16] Bain, Scott. Emergent Design: The Evolutionary Nature of Professional Software Development. Addison-Wesley, 2008.

[17] Goldstein, Ilan. Scrum Shortcuts without Cutting Corners, Agile Tactics, Tools, & Tips. Addison-Wesley, 2013.

Fowler on how much design up front before coding

Refactoring and Design

Refactoring has a special role as a complement to design. When I first learned to program, I just wrote the program and muddled my way through it. In time I learned that thinking about the design in advance helped me avoid costly rework. In time I got more into this style of *upfront design*. Many people consider design to be the key piece and programming just mechanics. The analogy is design is an engineering drawing and code is the construction work. But software is very different from physical machines. It is much more malleable, and it is all about thinking. As Alistair Cockburn puts it, "With design I can think very fast, but my thinking is full of little holes."

One argument is that refactoring can be an alternative to upfront design. In this scenario you don't do any design at all. You just code the first approach that comes into your head, get it working, and then refactor it into shape. Actually, this approach can work. I've seen people do this and come out with a very well-designed piece of software. Those who support Extreme Programming [Beck, XP] often are portrayed as advocating this approach.

Although doing only refactoring does work, it is not the most efficient way to work. Even the extreme programmers do some design first. They will try out various ideas with CRC cards or the like until they have a plausible first solution. Only after generating a plausible first shot will they code and then refactor. The point is that refactoring changes the role of upfront design. If you don't refactor, there is a lot of pressure in getting that upfront design right. The sense is that any changes to the design later are going to be expensive. Thus you put more time and effort into the upfront design to avoid the need for such changes.

With refactoring the emphasis changes. You still do upfront design, but now you don't try to find *the* solution. Instead all you want is a reasonable solution. You know that as you build the solution, as you understand more about the problem, you realize that the best solution is different from the one you originally came up with. With refactoring this is not a problem, for it no longer is expensive to make the changes.

Fowler, Martin. Refactoring: Improving the Design of Existing Code. Boston, MA, USA: Addison-Wesley, 1999.

Kent Beck on how much design up front before coding

As I write this edition in the mid-2000s, some software swamis are arguing for not doing any design at all. "Big Design Up Front is *BDUF*," they say. "BDUF is bad. You're better off not doing any design before you begin coding!"

In ten years the pendulum has swung from "design everything" to "design nothing." But the alternative to BDUF isn't no design up front, it's a Little Design Up Front (LDUF) or Enough Design Up Front—*ENUF*.

How do you tell how much is enough? That's a judgment call, and no one can make that call perfectly. But while you can't know the exact right amount of design with any confidence, two amounts of design are guaranteed to be wrong every time: designing every last detail and not designing anything at all. The two positions advocated by extremists on both ends of the scale turn out to be the only two positions that are always wrong!

[3] Beck, Kent. Test Driven Development By Example. 2002

Chris Sterling on up front design

Get It "Right" the First Time

Getting it "right" the first time is antithetical to duplication. Because duplication in all of its incarnations is thought of as a negligent programming style, getting it "right" attempts to counteract this style with overabundant planning and design. This has been a problematic ideal that creates overly complex solutions with unused functionality that must also be maintained into the future. Designing a perfect solution is a destructive endeavor and is elusive to even the most capable software craftsman.

We're more likely to get it "right" the third time.

[4] Sterling, Chris. Managing Software Debt, Building for Inevitable Change. Addison-Wesley, 2010.

Martin Fowler on when to refactor

When Should You Refactor?

When I talk about refactoring, I'm often asked about how it should be scheduled. Should we allocate two weeks every couple of months to refactoring?

In almost all cases, I'm opposed to setting aside time for refactoring. In my view refactoring is not an activity you set aside time to do. Refactoring is something you do all the time in little bursts. You don't decide to refactor, you refactor because you want to do something else, and refactoring helps you do that other thing.

Refactor When You Add Function

The most common time to refactor is when I want to add a new feature to some software. Often the first reason to refactor here is to help me understand some code I need to modify. This code may have been written by someone else, or I may have written it. Whenever I have to think to understand what the code is doing, I ask myself if I can refactor the code to make that understanding more immediately apparent. Then I refactor it. This is partly for the next time I pass by here, but mostly it's because I can understand more things if I clarify the code as I'm going along.

The other driver of refactoring here is a design that does not help me add a feature easily. I look at the design and say to myself, "If only I'd designed the code this way, adding this feature would be easy." In this case I don't fret over my past misdeeds—I fix them by refactoring. I do this partly to make future enhancements easy, but mostly I do it because I've found it's the fastest way. Refactoring is a quick and smooth process. Once I've refactored, adding the feature can go much more quickly and smoothly.

[1] Fowler, Martin. Refactoring: Improving the Design of Existing Code. Boston, MA, USA: Addison-Wesley, 1999.

Mike Cohn on upfront design

One of the criticisms of agile processes is that there is no upfront design step, as there is in a waterfall process. While it's true there is no upfront design *phase*, agile processes are characterized by frequent short bursts of design. Disaggregating stories into tasks—which can only be done with at least a minimal design in mind—is one of these short bursts of just–in–time design that replace a waterfall's upfront design phase.

[7] Cohn, Mike. User Stories Applied For Agile Software Development. Addison-Wesley, 2004.

Kent Beck on TDD and Design

1. Quickly add a test
2. Run all tests and see the new one fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication

Kent Beck in regards to step 3. above

You probably aren't going to like the solution, but the goal right now is not to get the perfect answer, the goal is to pass the test. We'll make our sacrifice at the altar of truth and beauty later.

Now I'm worried. I've given you a license to abandon all the principles of good design. Off you go to your teams—"Kent says all that design stuff doesn't matter." Halt. The cycle is not complete. A four legged Aeron chair falls over. The first four steps of the cycle won't work without the fifth. Good design at good times. Make it run, make it right.

Uncle Bob on the number of projects to work on at a time

Now here's a rule: There is no such thing as half a person.

It makes no sense to tell a programer to devote half their time to project A and the rest of their time to project B, especially when the two projects have two different project managers, different business analysts, different programmers, and different testers. How in Hell's kitchen can you call a monstrosity like that a team? That's not a team, that's something that came out of a Waring blender.

The Clean Coder: A Code of Conduct for Professional Programmers. Prentice Hall, 2011.

Mike Cohn on working on multiple projects at a time

“Put people on one project. Individuals assigned to work on multiple projects inevitably get less done. Multitasking—attempting to work on two projects or two things at once— solved. Really, though, in many cases the problem has been made worse.”

[5] Cohn, Mike. Succeeding With Agile: Software Development Using Scrum. Addison-Wesley, 2013.

Mike Cohn on communication between teams

Does the structure minimize the number of communication paths between teams?

A poor team structure design will result in a seemingly infinite number of communication paths between teams. Teams will find themselves unable to complete any work without coordinating first with too many other teams. Some interteam coordination will always be required. But, if a team that wants to add a new field on a form is required to coordinate that effort with three other teams, as I've seen, then the communication overhead is too high.

[5] Cohn, Mike. Succeeding With Agile: Software Development Using Scrum. Addison-Wesley, 2013.

Mike Cohn on feature vs component teams

Are component teams used only in limited and easily justifiable cases?

Most teams should be created around the end-to-end delivery of working features. In some cases, it is acceptable to have a component team developing reusable user interface components, providing access to a database, or similar functionality. But these should be exceptions.

[5] Cohn, Mike. Succeeding With Agile: Software Development Using Scrum. Addison-Wesley, 2013.

Kent Beck on Overtime

Overtime is a symptom of a serious problem on the project. The XP rule is simple—you can't work a second week of overtime. For one week, fine, crank and put in some extra hours. If you come in on Monday and say "To meet our goals, we'll have to work late again," then you already have a problem that can't be solved by working more hours. (2004, 60)

[6] Beck, Kent. Extreme Programming Explained, Embrace Change . Addison-Wesley, 2012.

Kent Beck on trying to “get it right” the first time

I remember an all-day presentation I gave in Aarhus, Denmark. One front-row attendee’s face got cloudier and cloudier as the day progressed. Finally he couldn’t stand it. “Wouldn’t it be easier just to do it right in the first place?” Of course it would, except for three things:

- We may not know how to do it “right”. If we are solving a novel problem there may be several solutions that might work or there may be no clear solution at all.
- What’s right for today may be wrong for tomorrow. Changes outside our control or ability to predict can easily invalidate yesterday’s decisions.
- Doing everything “right” today might take so long that changing circumstances tomorrow invalidate today’s solution before it is even finished.

Being satisfied with improvement rather than expecting instant perfection, we use feedback to get closer and closer to our goals. Feedback comes in many forms:

- Opinions about an idea, yours or your teammates’
- How the code looks when you implement the idea
- Whether the tests were easy to write
- Whether the tests run
- How the idea works once it has been deployed

[6] Beck, Kent. Extreme Programming Explained, Embrace Change . Addison-Wesley, 2012.

Kent Beck on maintaining one version of code

Single Code Base

There is only one code stream. You can develop in a temporary branch, but never let it live longer than a few hours.

Multiple code streams are an enormous source of waste in software development. I fix a defect in the currently deployed software. Then I have to retrofit the fix to all the other deployed versions and the active development branch. Then you find that my fix broke something you were working on and you interrupt me to fix my fix. And on and on.

There are legitimate reasons for having multiple versions of the source code active at one time. Sometimes, though, all that is at work is simple expedience, a micro-optimization taken without a view to the macro-consequences. If you have multiple code bases, put a plan in place for reducing them gradually. You can improve the build system to create several products from a single code base. You can move the variation into configuration files. Whatever you have to do, improve your process until you no longer need multiple versions of the code.

One of my clients had seven different code bases for seven different customers and it was costing them more than they could afford. Development was taking far longer than it used to. Programmers were creating far more defects than before. Programming just wasn't as fun as it had been initially. When I pointed out the costs of the multiple code bases and the impossibility of scaling such a practice, the client responded that they simply couldn't afford the work of reuniting the code. I couldn't convince the client to even try reducing from seven to six versions or adding the next customer as a variation of one of the existing versions.

Don't make more versions of your source code. Rather than add

[6] Beck, Kent. Extreme Programming Explained, Embrace Change . Addison-Wesley, 2012.

Kenneth Rubin on upfront requirements and upfront design

The fact is, when developing innovative products, you can't create complete requirements or designs up front by simply working longer and harder. Some requirements and design will always emerge once product development is under way; no amount of comprehensive up-front work will prevent that.

Thus, when using Scrum, we don't invest a great deal of time and money in fleshing out the details of a requirement up front. Because we expect the specifics to change as time passes and as we learn more about what we are building, we avoid overinvesting in requirements that we might later discard. Instead of compiling a large inventory of detailed requirements up front, we create placeholders for the requirements, called **product backlog items** (PBIs). Each product backlog item represents desirable business value (see Figure 5.1).

[8] Rubin, Kenneth. Essential Scrum: A Practical Guide To The Most Popular Agile Process. Addison-Wesley, 2013.

Kenneth Rubin on cross-functional teams

Cross-Functionally Diverse and Sufficient

Development team members should be cross-functionally diverse; collectively they should possess the necessary and sufficient set of skills to get the job done. A well-formed team can take an item off of the product backlog and produce a good-quality, working feature that meets the Scrum team's definition of done.

Teams composed solely of people with the same skills (traditional silo teams) can at most do part of the job. As a result, silo teams end up handing off work products to other silo teams. For example, the development team hands the code off to the testing team, or the UI team hands off screen designs to the business logic team. Handoffs represent an excellent opportunity for miscommunication and costly mistakes. Having diverse teams minimizes the number of handoffs. And creating diverse teams doesn't prevent us from having multiple team members who might be highly skilled in the same discipline such as Java or C++ development or testing.

Cross-functionally diverse teams also bring multiple perspectives, leading to better outcomes (see Figure 11.5).

[8] Rubin, Kenneth. Essential Scrum: A Practical Guide To The Most Popular Agile Process. Addison-Wesley, 2013.

Kenneth Rubin on working on multiple projects

If a person is working on only one product, it is far easier for that person to be focused and committed. When asked to work on multiple concurrent product development efforts, a person must split her time across those products, reducing her focus and commitment on all products.

Ask any person who works on multiple products about her focus and commitment and you will likely be told something like "I have so much to do that I just try to do the best job that I can on each product and then hop to the next product. I don't ever feel like I have time to focus on any one product and do it well. If there is an emergency situation on several products, I simply won't be able to help out on all of them."

It is harder for a team member to do a good-quality job when she is hopping from product to product. It's even harder to be truly committed to multiple products simultaneously. Instead of being in one boat with her team members, the multitasking team member is moving from boat to boat. If many of the boats spring a leak at the same time, how does this person choose which boat's crew to help? If a person isn't there to bail water, that team member is not *committed* to that team. At best she is *involved* with that team. To be fair to the other team members the involved team member should make it perfectly clear that she is only involved and therefore might not be available at critical times.

There is considerable data to support the widely held belief that being on multiple products (or projects) or multiple teams reduces productivity. Figure 11.8 shows a graph of such data (Wheelwright and Clark 1992).

This data indicates that nobody is 100% productive—there is overhead just to be a good corporate citizen. Productivity actually seems better with two projects than with one. This occurs because it is possible to get blocked on one project, so having a second one to switch to allows a person to be incrementally more productive.

Based on this data, working on three or more concurrent projects is a bad economic choice because more time is spent on coordinating, remembering, and tracking down information and less time is spent doing value-adding work. So, how many projects/products (and probably different teams) should a person be on simultaneously? Probably not more than two. I have a strong preference for one, because in today's highly connected, information-rich world with email, instant messaging, Twitter, Facebook, and other technologies, being a good corporate citizen is probably the equivalent of being on one project!

[8] Rubin, Kenneth. Essential Scrum: A Practical Guide To The Most Popular Agile Process. Addison-Wesley, 2013.

Kenneth Rubin on feature teams vs component teams

In my experience, most organizations using component teams recognize that there's a problem when things begin to fall on the floor (the baton drops, causing a break in value-delivery flow). It usually goes something like this. A senior manager asks a feature-level product owner, "How come the customer feature isn't ready?" The response: "Well, all but one of the component teams finished the pieces we assigned to them. Because that last team didn't finish, the feature isn't done." The manager might then say, "Why didn't that team finish the piece you gave them?" The response might be "I asked, and I was told that they had 15 other competing requests for changes in their component area, and for technical reasons they felt it made more sense to work on the requests from other products before ours. But they still promise to finish our piece—perhaps in the next sprint."

This is no way to operate a business. We can never be certain when (or even if) we can deliver a feature—because the responsibility for delivery has been distributed among two or more component teams, each of which might have very different priorities. Using component teams this way multiplicatively increases the probability that a feature won't get finished, because there are now multiple points of failure (each component team) instead of one (a single feature team).

Is there a solution to this problem? Well, a very good solution would be to create cross-functional feature teams that have all of the skills necessary to work on multiple end-customer features and get them done—without having to farm out pieces to component teams. But what about the principal reason that most organizations create component teams—having a single trusted team to work in a component area? Won't feature teams lead to chaotic development and maintenance of reusable components with large amounts of technical debt? Not if we have well-formed feature teams that, over time, share code ownership and collectively become trusted custodians of the code.

[8] Rubin, Kenneth. Essential Scrum: A Practical Guide To The Most Popular Agile Process. Addison-Wesley, 2013.

Kenneth Rubin on T Shaped skills

Who Does the Work?

Who should work on each task? An obvious answer is the person best able to quickly and correctly get it done. What if that person is unavailable? Perhaps she is already working on another, more important task, or maybe she is out sick and the task needs to get done immediately.

There are a number of factors that can and should influence who will work on a task; it's the collective responsibility of the team members to consider those factors and make a good choice.

When team members have T-shaped skills, several people on the team have the ability to work on each task. When some skills overlap among team members, the team can swarm people to the tasks that are inhibiting the flow of a product backlog item through sprint execution, making the team more efficient.

[8] Rubin, Kenneth. Essential Scrum: A Practical Guide To The Most Popular Agile Process. Addison-Wesley, 2013.

Jez Humble on branching

A solution that some people use to resolve this dilemma is to create a separate branch within the version control system for new functionality. At some point, when the changes are deemed satisfactory, they will be merged into the main development branch. This is a bit like a two-stage check-in; in fact, some version control systems work naturally in this way.

However, we are opposed to this practice (with three exceptions, discussed in Chapter 14). This is a controversial viewpoint, especially to users of tools like ClearCase. There are a few problems with this approach.

- It is antithetical to continuous integration, since the creation of a branch defers the integration of new functionality, and integration problems are only found when the branch is merged.
- If several developers create branches, the problem increases exponentially, and the merge process can become absurdly complex.
- Although there are some great tools for automated merging, these don't solve semantic conflicts, such as somebody renaming a method in one branch while somebody else adds a new call to that method in another branch.
- It becomes very hard to refactor the codebase, since branches tend to touch many files which makes merging even more difficult.

A much better answer is to develop new features incrementally and to commit them to the trunk in version control on a regular and frequent basis. This keeps the software working and integrated at all times. It means that your software is always tested because your automated tests are run on trunk by the continuous integration (CI) server every time you check in. It reduces the possibility of large merge conflicts caused by refactoring, ensures that integration problems are caught immediately when they are cheap to fix, and results in higher-quality software. We discuss techniques to avoid branching in more detail in Chapter 13, "Managing Components and Dependencies."

[10] Humble, Jez and Farley, David. Continuous Delivery: Reliable Software Releases Through Build, Test And Deployment Automation. Addison-Wesley, 2010.

Jez Humble on how frequent to merge to master

The second is to introduce changes incrementally. We recommend that you aim to commit changes to the version control system at the conclusion of each separate incremental change or refactoring. If you use this technique correctly, you should be checking in at the very minimum once a day, and more usually several times a day. This may sound unrealistic if you are not used to doing it, but we assure you, it leads to a far more efficient software delivery process.

[10] Humble, Jez and Farley, David. Continuous Delivery: Reliable Software Releases Through Build, Test And Deployment Automation. Addison-Wesley, 2010.

Jez Humble on component teams

We do *not* recommend making teams responsible for individual components. This is because in most cases, requirements don't divide along component boundaries. In our experience, cross-functional teams in which people develop features end-to-end are much more effective. Although one team per component may seem more efficient, this is not in fact the case.

First, it is often hard to write and test requirements for a single component in isolation, since usually implementing a piece of functionality will touch more than one component. If you group teams by component, you thus require two or more teams to collaborate to complete a feature, automatically adding a large and unnecessary communication cost. Furthermore, people in component-centered teams tend to form silos and optimize locally, losing their ability to judge what is in the best interest of the project as a whole.

It is better to split teams up so that each team takes on one stream of stories (perhaps all with a common theme), and touches whatever components they need to in order to get their work done. Teams with a mandate to implement a business-level feature, and the freedom to change any component that they need to, are much more efficient. Organize teams by functional area rather than by component, ensure that everybody has the right to change any part of the codebase, rotate people between teams regularly, and ensure that there is good communication between teams.

This approach also has the benefit that making all the components work together is everybody's responsibility, not just that of the integration team. One of the more serious dangers of having a team per component is that the application as a whole won't work until the end of the project because nobody has the incentive to integrate the components.

[10] Humble, Jez and Farley, David. Continuous Delivery: Reliable Software Releases Through Build, Test And Deployment Automation. Addison-Wesley, 2010.

Jez Humble on Conway's Law

In other words, how we organize our teams has a powerful effect on the software we produce, as well as our resulting architectural and production outcomes. In order to get fast flow of work from Development into Operations, with high quality and great customer outcomes, we must organize our teams and our work so that Conway's Law works to our advantage. Done poorly, Conway's Law will prevent teams from working safely and independently; instead, they will be tightly-coupled together, all waiting on each other for work to be done, with even small changes creating potentially global, catastrophic consequences.

[13] Kim, Gene, et al. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution, 2016.

Jez Humble on overly functional orientation

PROBLEMS OFTEN CAUSED BY OVERLY FUNCTIONAL ORIENTATION (“OPTIMIZING FOR COST”)

In traditional IT Operations organizations, we often use functional orientation to organize our teams by their specialties. We put the database administrators in one group, the network administrators in another, the server administrators in a third, and so forth. One of the most visible consequences of this is long lead times, especially for complex activities like large deployments where we must open up tickets with multiple groups and coordinate work handoffs, resulting in our work waiting in long queues at every step.

Compounding the issue, the person performing the work often has little visibility or understanding of how their work relates to any value stream goals (e.g., “I’m just configuring servers because someone told me to.”). This places workers in a creativity and motivation vacuum.

The problem is exacerbated when each Operations functional area has to serve multiple value streams (i.e., multiple Development teams) who all compete for their scarce cycles. In order for Development teams to get their work done in a timely manner, we often have to escalate issues to a manager or director, and eventually to someone (usually an executive) who can finally prioritize the work against the global organizational goals instead of the functional silo goals. This decision must then get cascaded down into each of the functional areas to change the local priorities, and this, in turn, slows down other teams. When every team expedites their work, the net result is that every project ends up moving at the same slow crawl.

[13] Kim, Gene, et al. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution, 2016.

Jez Humble on cross-team communication

Ideally, our software architecture should enable small teams to be independently productive, sufficiently decoupled from each other so that work can be done without excessive or unnecessary communication and coordination.

Our goal is to enable market-oriented outcomes where many small teams can quickly and independently deliver value to the customer. This can be a challenge to achieve when Operations is centralized and functionally-oriented, having to serve the needs of many different development teams with potentially wildly different needs. The result can often be long lead times for needed Ops work, constant reprioritization and escalation, and poor deployment outcomes.

[13] Kim, Gene, et al. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution, 2016.

Jez Humble on Trunk Based Development

However, the longer developers are allowed to work in their branches in isolation, the more difficult it becomes to integrate and merge everyone's changes back into trunk. In fact, integrating those changes becomes exponentially more difficult as we increase the number of branches and the number of changes in each code branch.

ADOPT TRUNK-BASED DEVELOPMENT PRACTICES

Our countermeasure to large batch size merges is to institute continuous integration and trunk-based development practices, where all developers check in their code to trunk at least once per day. Checking code in this frequently reduces our batch size to the work performed by our entire developer team in a single day. The more frequently developers check in their code to trunk, the smaller the batch size and the closer we are to the theoretical ideal of single-piece flow.

[13] Kim, Gene, et al. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution, 2016.

Jez Humble on making decisions where the knowledge is

One of the core beliefs in the Toyota Production System is that "people closest to a problem typically know the most about it." This becomes more pronounced as the work being performed and the system the work occurs in become more complex and dynamic, as is typical in DevOps value streams. In these cases, creating approval steps from people who are located further and further away from the work may actually reduce the likelihood of success. As has been proven time and again, the further the distance between the person doing the work (i.e., the change implementer) and the person deciding to do the work (i.e., the change authorizer), the worse the outcome.

[13] Kim, Gene, et al. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution, 2016.

Jez Humble on architectural capabilities

In teams which scored highly on architectural capabilities, little communication is required between delivery teams to get their work done, and the architecture of the system is designed to enable teams to test, deploy, and change their systems without dependencies on other teams. In other words, architecture and teams are loosely coupled. To enable this, we must also ensure delivery teams are cross-functional, with all the skills necessary to design, develop, test, deploy, and operate the system on the same team.

[14] Forsgren, Nicole, et al. Accelerate, The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations. IT Revolution, 2018.

Jez Humble on the impact of change approval boards on software delivery performance

We wanted to investigate the impact of change approval processes on software delivery performance. Thus, we asked about four possible scenarios:

1. All production changes must be approved by an external body (such as a manager or CAB).
2. Only high-risk changes, such as database changes, require approval.
3. We rely on peer review to manage changes.
4. We have no change approval process.

The results were surprising. We found that approval only for high-risk changes was not correlated with software delivery performance. Teams that reported no approval process or used peer review achieved higher software delivery performance. Finally, teams that required approval by an external body achieved lower performance.

We investigated further the case of approval by an external body to see if this practice correlated with stability. We found that external approvals were negatively correlated with lead time, deployment frequency, and restore time, and had no correlation with change fail rate. In short, approval by an external body (such as a manager or CAB) simply doesn't work to increase the stability of production systems, measured by the time to restore service and change fail rate. However, it certainly slows things down. It is, in fact, worse than having no change approval process at all.