Certified Scrum Developer Course Workbook

Except where otherwise specified, all content © 2015-2016 Gargoyle Software Inc

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

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity--the art of maximizing the amount of work not done--is essential.

The best architectures, requirements, and designs emerge from self-organizing teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

Source: http://www.agilemanifesto.org

Except where otherwise specified, all content © 2015-2016 Gargoyle Software Inc

Introduction to the class

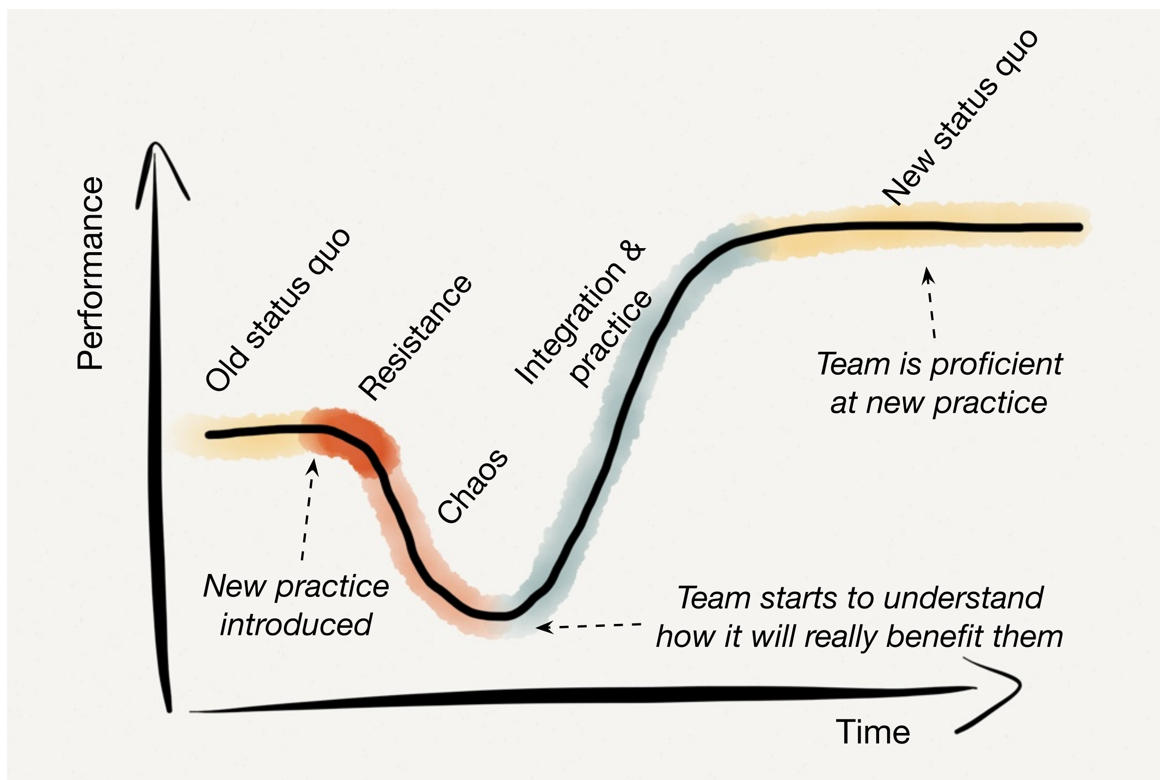
Why are we here?

Effective Scrum (or more generally, Agile) teams are able to deliver more value, sooner, to their clients. Additionally, they're able to continuously deliver this value over long periods of time.

To do that, we require effective teamwork and product management to ensure we're building the right thing at the right time. We also need technical excellence in order to ensure that we're building the thing "right".

This class is about the latter. How do we build technical excellence into everything we do? How do we get quality up and delivery faster? How do we change so that we're spending more time on building new features and less on fixing broken ones?

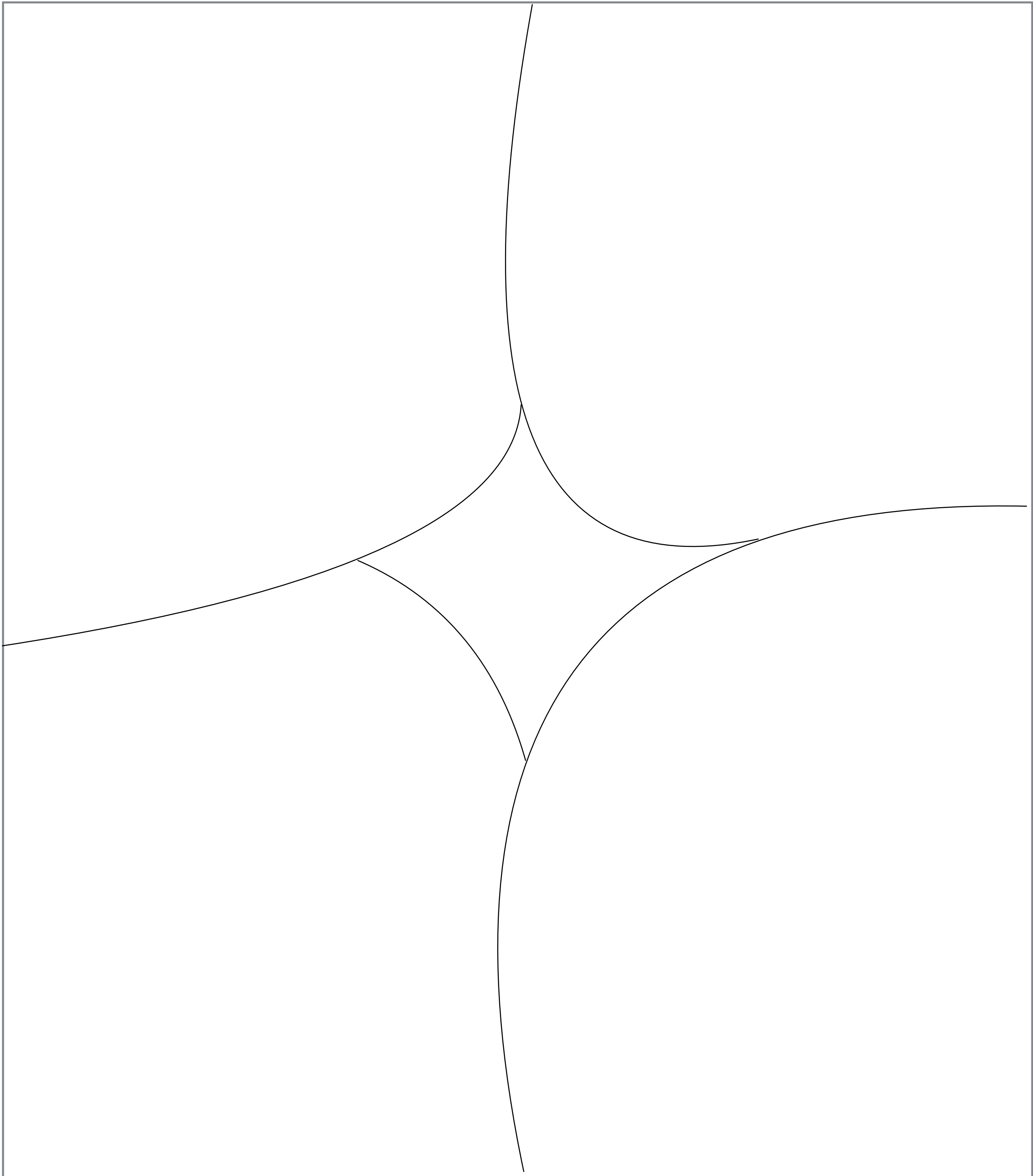
A note about getting better



SATIR CHANGE CURVE

Except where otherwise specified, all content © 2015-2016 Gargoyle Software Inc

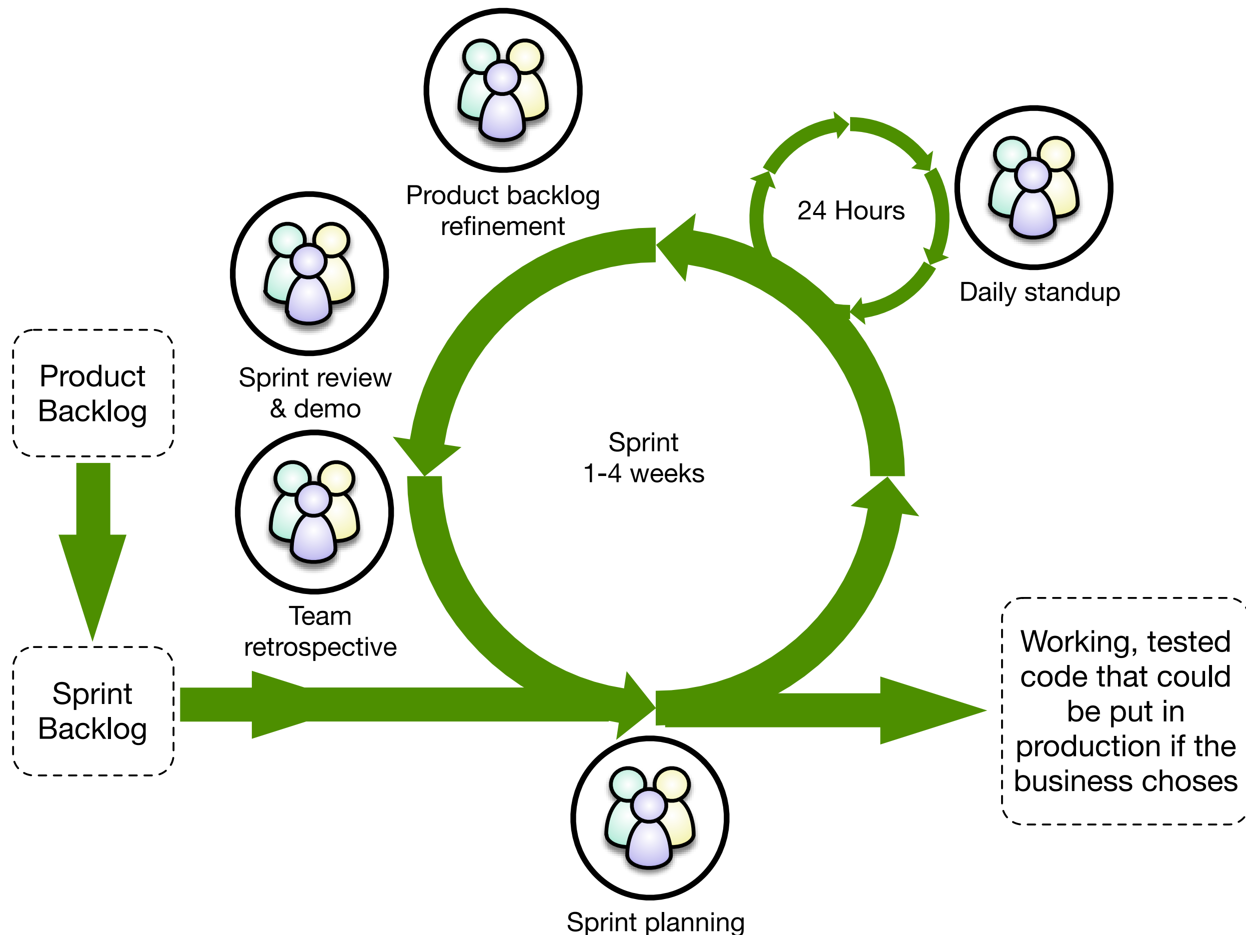
Cynefin



Source: http://cognitive-edge.com/

Except where otherwise specified, all content © 2015-2016 Gargoyle Software Inc

Scrum



The definitive documentation for Scrum is the Scrum Guide, which can be found at http://www.scrumguides.org/

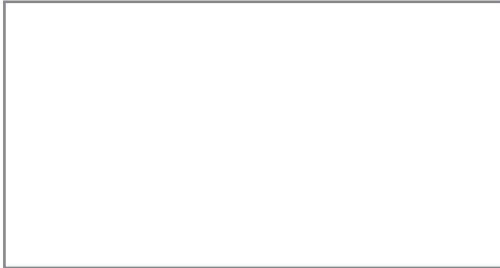
If something isn't in this document then it isn't part of canonical scrum. Having said that, there are many practices that today are commonplace for Scrum teams and yet aren't in this document at all. Stories, story points and technical practices to name just a few.

Ken and Jeff, the creators of Scrum and authors of the Scrum Guide, have stated repeatedly that anyone using Scrum for software development should extend the scrum practices with strong technical practices. Specifically, they refer to the set of technical practices from eXtreme Programming (XP) which is what we will cover in this class.

Except where otherwise specified, all content © 2015-2016 Gargoyle Software Inc

Personas

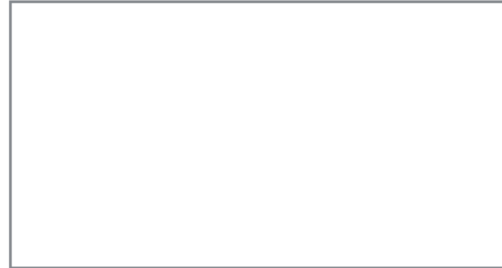
Who is trying to use the product?
What is this person wanting to do?
What need is this product solving for them?



Name:

Background:

Reason for using the product:



Name:

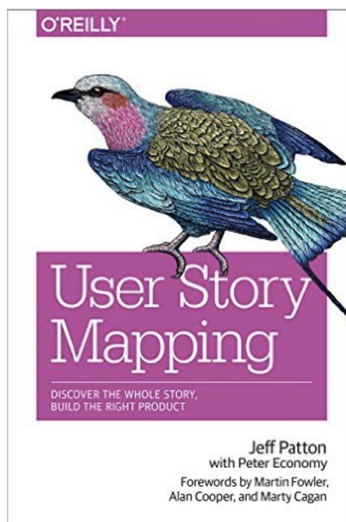
Background:

Reason for using the product:

Except where otherwise specified, all content © 2015-2016 Gargoyle Software Inc

Story Mapping

Source: http://winnipegagilist.blogspot.ca/2012/03/how-to-create-user-story-map.html



Except where otherwise specified, all content © 2015-2016 Gargoyle Software Inc

HOW TO SPLIT A USER STORY

1 PREPARE THE INPUT STORY

Does the big story satisfy INVEST* (except, perhaps, small)?

NO → Combine it with another story or otherwise reformulate it to get a good, if large, starting story.

YES → Is the story size 1/10 to 1/6 of your velocity?

- You're done.
- Continue. You need to split it.

2 APPLY THE SPLITTING PATTERNS

start here

WORKFLOW STEPS

Does the story describe a workflow?

- Can you split the story so you do the beginning and end of the workflow first and enhance with stories from the middle of the workflow?
- Can you take a thin slice through the workflow first and enhance it with more stories later?

OPERATIONS

Does the story include multiple operations? (e.g. is it about "managing" or "configuring" something?)

Can you split the operations into separate stories?

BUSINESS RULE VARIATIONS

Does the story have a variety of business rules? (e.g. is there a domain term in the story like "flexible dates" that suggests several variations?)

Can you split the story so you do a subset of the rules first and enhance with additional rules later?

VARIATIONS IN DATA

Does the story do the same thing to different kinds of data?

Can you split the story to process one kind of data first and enhance with the other kinds later?

INTERFACE VARIATIONS

Does the story have a complex interface?

Is there a simple version you could do first?

Does the story get the same kind of data via multiple interfaces?

Can you split the story to handle data from one interface first and enhance with the others later?

MAJOR EFFORT

When you apply the obvious split, is whichever story you do first the most difficult?

Could you group the later stories and defer the decision about which story comes first?

SIMPLE/COMPLEX

Does the story have a simple core that provides most of the value and/or learning?

Could you split the story to do that simple core first and enhance it with later stories?

DEFER PERFORMANCE

Does the story get much of its complexity from satisfying non-functional requirements like performance?

Could you split the story to just make it work first and then enhance it to satisfy the non-functional requirement?

BREAK OUT A SPIKE

last resort

Are you still baffled about how to split the story?

Can you find a small piece you understand well enough to start?

- Write that story first, build it, and start again at the top of this process.
- Can you define the 1-3 questions most holding you back?
  - Write a spike with those questions, do the minimum to answer them, and start again at the top of this process
  - Take a break and try again.

3 EVALUATE THE SPLIT

Are the new stories roughly equal in size?

NO → Try another pattern on the original story or the larger post-split stories.

YES → Is each story about 1/10 to 1/6 of your velocity? → Try another pattern on the original story or the larger post-split stories.

Do each of the stories satisfy INVEST? → Try another pattern.

Are there stories you can deprioritize or delete? → Try another pattern. You probably have waste in each of your stories.

Is there an obvious story to start with that gets you early value, learning, risk mitigation, etc.? → Try another pattern to see if you can get this.

You're done, though you could try another pattern to see if it works better.

* INVEST - Stories should be:
Independent
Negotiable
Valuable
Estimable
Small
Testable

www.agileforall.com

Visit http://www.richardlawrence.info/splitting-user-stories/ for more info on the story splitting patterns

Copyright © 2011-2013 Agile For All. All rights reserved.

Last updated 3/26/2013

Except where otherwise specified, all content © 2015-2016 Gargoyle Software Inc

Qualities of a good story

I	Stories are easiest to work with if they are **independent**. That is, we'd like them to not overlap in concept, and we'd like to be able to schedule and implement them in any order.
N	A good story is **negotiable**. It is not an explicit contract for features; rather, details will be co-created by the customer and programmer during development. A good story captures the essence, not the details. Over time, the card may acquire notes, test ideas, and so on, but we don't need these to prioritize or schedule stories.
V	A story needs to be **valuable**. We don't care about value to just anybody; it needs to be valuable to the customer. Developers may have (legitimate) concerns, but these framed in a way that makes the customer perceive them as important. This is especially an issue when splitting stories. Think of a whole story as a multi-layer cake, e.g., a network layer, a persistence layer, a logic layer, and a presentation layer. When we split a story, we're serving up only part of that cake. We want to give the customer the essence of the whole cake, and the best way is to slice vertically through the layers. Developers often have an inclination to work on only one layer at a time (and get it "right"); but a full database layer (for example) has little value to the customer if there's no presentation layer.
E	A good story can be **estimated**. We don't need an exact estimate, but just enough to help the customer rank and schedule the story's implementation. Being estimable is partly a function of being negotiated, as it's hard to estimate a story we don't understand. It is also a function of size: bigger stories are harder to estimate. Finally, it's a function of the team: what's easy to estimate will vary depending on the team's experience. (Sometimes a team may have to split a story into a (time-boxed) "spike" that will give the team enough information to make a decent estimate, and the rest of the story that will actually implement the desired feature.)
S	Good stories tend to be **small**. Stories typically represent at most a few person-weeks worth of work. (Some teams restrict them to a few person-days of work.) Above this size, and it seems to be too hard to know what's in the story's scope. Saying, "it would take me more than a month" often implicitly adds, "as I don't understand what-all it would entail." Smaller stories tend to get more accurate estimates. Story descriptions can be small too (and putting them on an index card helps make that happen). Alistair Cockburn described the cards as tokens promising a future conversation. Remember, the details can be elaborated through conversations with the customer.
T	A good story is **testable**. Writing a story card carries an implicit promise: "I understand what I want well enough that I could write a test for it." Several teams have reported that by requiring customer tests before implementing a story, the team is more productive. "Testability" has always been a characteristic of good requirements; actually writing the tests early helps us know whether this goal is met. If a customer doesn't know how to test something, this may indicate that the story isn't clear enough, or that it doesn't reflect something valuable to them, or that the customer just needs help in testing.

Source: http://xp123.com/articles/invest-in-good-stories-and-smart-tasks/ by Bill Wake

Except where otherwise specified, all content © 2015-2016 Gargoyle Software Inc

Stories

User Story Template

As **<who>**, I want to **<what>** so that **<why>**

As **a bank customer**,
I want to **report fraud activity**
so that **I am not liable for false charges**

As **a bank customer**
I want to **pay my bills online**
so that **I don't have to go into a branch location**

With personas, can becomes cleaner:

<who>, wants to **<what>** so that **<why>**

Amy wants to **report fraud activity**
so that **she isn't liable for false charges**

Is it…

- ☐ **I**ndependent
- ☐ **N**egotiable
- ☐ **V**aluable
- ☐ **E**stimable
- ☐ **S**mall
- ☐ **T**estable

Do we know…

- ☐ **Who** it's for
- ☐ **What** is wanted
- ☐ **Why** it's important

Do we have…

- ☐ Specific **examples**

Job Story Template

When **<situation>**, I want to **<what>**, so that **<why>**

When **a payments file arrives on our FTP server**,
I want to **load it into the payments system**
So that **customers are properly credited**

See also: http://alanklement.blogspot.com/2013/09/replacing-user-story-with-job-story.html

Acceptance Criteria (specific examples)

Given **<precondition>**, when **<action>** then **<result>**

Given Amy has fraud activity to report
When she fills out the online fraud form
Then a fraud report is sent to loss prevention

Given an incoming payments file is waiting on the FTP server with Amy's payment details
When the batch process runs
Then Amy's account is credited with the payment

Except where otherwise specified, all content © 2015-2016 Gargoyle Software Inc

Cucumber features

```
Feature: Book a hotel
    In this feature, we'll try different combinations of booking a hotel

    Scenario: View upcoming reservations
        Given I am on the space hotel site
        And I am signed in as "Amy"
        When I make a reservation for next month
        Then I see that reservation on the upcoming reservations page
```

```
Feature: Book a hotel
    In this feature, we'll try different combinations of booking a hotel

    Background:
        Given I am on the space hotel site

    Scenario: View upcoming reservations
        And I am signed in as "Amy"
        When I make a reservation for next month
        Then I see that reservation on the upcoming reservations page
```

```
Feature: Book a hotel
    In this feature, we'll try different combinations of booking a hotel

    Scenario Outline: View upcoming reservations
        Given I am on the space hotel site
        And I am signed in as "<user>"
        When I make a reservation for <when>
        Then I see that reservation on the upcoming reservations page

    Examples:
        | user | when      |
        | Amy  | next week |
        | Bob  | tomorrow  |
```

```
Feature: Book a hotel
    In this feature, we'll try different combinations of booking a hotel

    Scenario Outline: View upcoming reservations
        Given I am on the space hotel site with reservations:
            | confirmation_id | from       | to         | who |
            | 123             | 2024-01-03 | 2024-01-03 | Amy |
        And I am signed in as "Amy"
        When I cancel a reservation 123
        Then that reservation is no longer on the upcoming reservations page
```


Except where otherwise specified, all content © 2015-2016 Gargoyle Software Inc

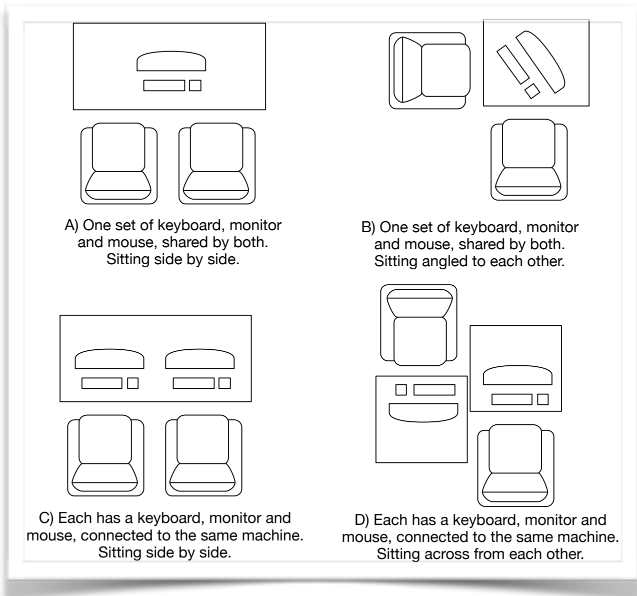
Importance of Common Language

- Brush
- Tooth
- Paste
- Bristles
- Water

Except where otherwise specified, all content © 2015-2016 Gargoyle Software Inc

Pair Programming

Styles



Driver-Navigator
The driver is at the keyboard, and takes a tactical view of the problem. The navigator observes and takes a strategic view of the problem, offering suggestions and asking questions. The roles may switch back and forth as needed, depending on how the work flows and how the interaction between the two people works.

Useful for general development and for knowledge/skill transfer. When used for knowledge/skills transfer, the junior member takes the driver role and the senior member takes the navigator role.

Ping-Pong
One member of the pair writes a failing unit test. The second member writes production code to make the test pass, and then writes the next unit test. The first member writes production code to make that test pass, and then writes the next unit test. The pair continues in the same way to solve the problem. The ping-pong style is usually applied when two peers are working on a problem together.

Silent Running
Typically as a variation on the ping-pong style, the developers do not speak aloud, but communicate only through the test code and production code they write.
Silent running is sometimes used as a way to alleviate the monotony of a long pairing session.

Anti-Patterns

- **Superman.** He refuses to be impaired by a slow partner. He will hog the keyboard and save the world all on his own.
- **Absent-Mind Ed.** He's distracted, sleepy, or preoccupied with other things besides the task at hand.
- **The Back-Seat Driver.** As navigator, he breaks the driver's train of thought with a non-stop stream of trivial comments.
- **The King of Shortcuts.** As navigator, he mentions every keyboard shortcut the driver fails to use, but doesn't make practical or meaningful observations.
- **The Anti-Mentor.** As navigator, this senior developer leaves his junior partner without guidance.
- **Fearful Freddie.** He refuses to refactor code he didn't personally write.
- **The Defactorator.** He reverses refactorings others have done to make the code consistent with his personal preferences.
- **The Soloist.** He works solo as much as possible, and has a large repertoire of excuses not to pair.

Source: "Does Pair Programming Work?" - presentation by Mike Bowler & Dave Nicolette

Except where otherwise specified, all content © 2015-2016 Gargoyle Software Inc

TDD: Test Driven Development

START
Identify the smallest piece of functionality that we want to write. Typically this will be a single path through a single method. It might be a "happy path" where the code does as expected or it could be an error condition.

RED - Write a failing test
Write an automated test that fails when you run it.

In order to even get the code to be executable at all, you'll need to write a tiny bit of the production code. Write only the bare minimum that allows the code to execute without having it actually perform any logic.

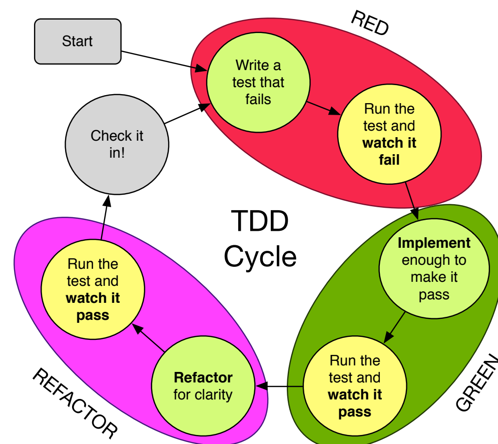
Run the test and watch it fail. Sometimes you'll run the test and it won't fail and this indicates that there's a problem somewhere. Most often the test is wrong but sometimes the production code isn't doing what we expect.

Ensure that the code is failing for the right reasons. If we expect it to return a 0 and instead it throws an exception, that is certainly a failure but it's failing for the wrong reasons.

GREEN - Make it pass
Now change the production code to make the pass. Do this in the simplest way you can possibly think to do it.

The goal here is just to make the test pass, in the easiest way possible. We don't care how pretty or how fast the code is, if it doesn't work. Make it work before doing anything else.

Run the test to verify that it really is working. Run all the tests, if you can, to ensure you didn't break anything else as you were making this one work.



REFACTOR - Make it beautiful
Now that we have working code, let's make it beautiful. Refactor it to keep the code clean. Remove duplication. Make the code easy to read. Make it something you can be proud of.

Now run all the tests again to ensure that nothing got broken as we were cleaning the code. If something did get broken then fix it now, before we move on.

CHECK IT IN!
Yes, really check it in to version control. You now have a tiny sliver of production code that is working and tested. By definition, we wouldn't be at this point if anything were broken. Check it in.

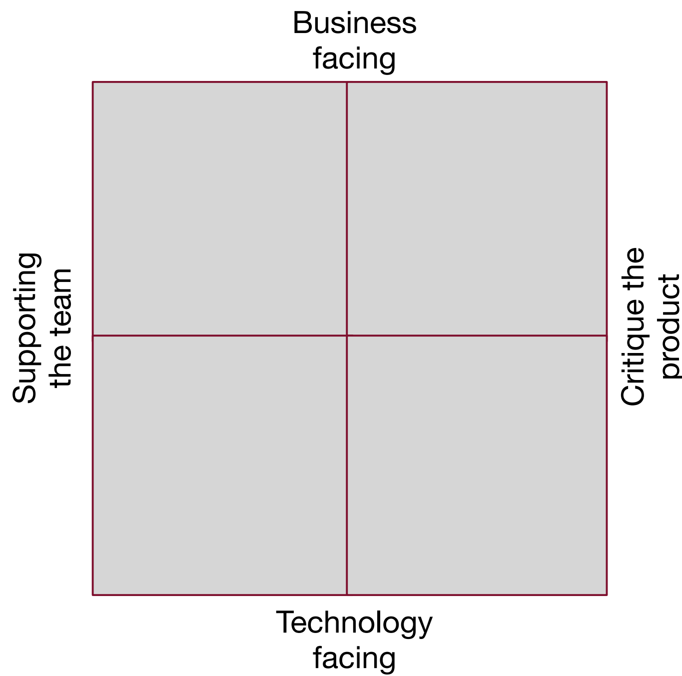
A surprising number of developers are really uncomfortable with checking in this frequently. The more often we integrate with the rest of the code, the easier it will be. If you're not checking in several times an hour then either you're very new to the technique or you're doing something wrong.

START AGAIN

Start all over at the top. Find the next tiny slice of functionality and work through the cycle again.

Except where otherwise specified, all content © 2015-2016 Gargoyle Software Inc

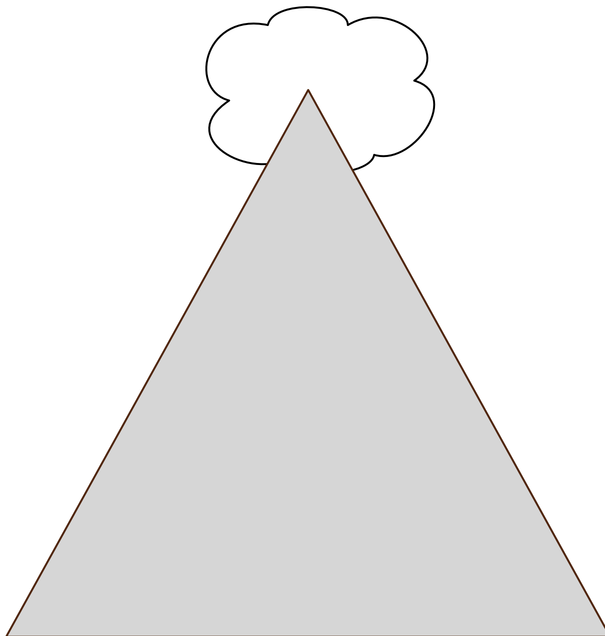
Agile Testing Quadrants and Pyramid



Sources:

The agile testing quadrants
http://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/

Merging the quadrants and the pyramid
http://swtester.blogspot.com/2015/04/agile-testing-automation.html

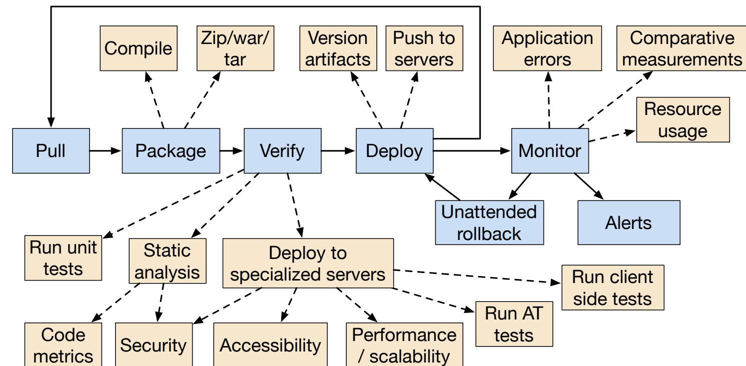


Except where otherwise specified, all content © 2015-2016 Gargoyle Software Inc

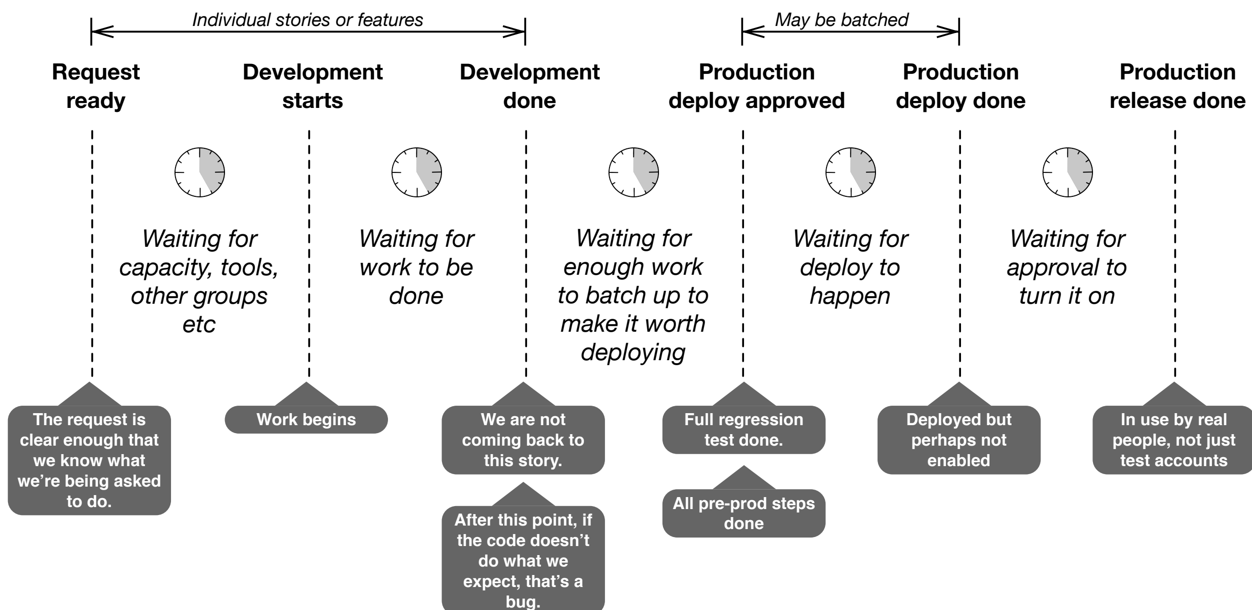
Continuous Delivery

Prerequisites

- Extreme quality
- Automate and Streamline
  - Continuous integration
  - One click deploy
  - Single branch
- Feature flags
- Monitoring and alerts
  - Absolute and comparative metrics
- Whole team



Measurements



Except where otherwise specified, all content © 2015-2016 Gargoyle Software Inc

Appendix A: Rules for the Gilded Rose exercise

Hi and welcome to team Gilded Rose. As you know, we are a small inn with a prime location in a prominent city ran by a friendly innkeeper named Allison. We also buy and sell only the finest goods. Unfortunately, our goods are constantly degrading in quality as they approach their sell by date. We have a system in place that updates our inventory for us. It was developed by a no-nonsense type named Leeroy, who has moved on to new adventures. Your task is to add the new feature to our system so that we can begin selling a new category of items.

First an introduction to our system:

- All items have a SellIn value which denotes the number of days we have to sell the item
- All items have a Quality value which denotes how valuable the item is
- At the end of each day our system lowers both values for every item

Pretty simple, right? Well this is where it gets interesting:

- Once the sell by date has passed, Quality degrades twice as fast
- The Quality of an item is never negative
- "Aged Brie" actually increases in Quality the older it gets
- The Quality of an item is never more than 50
- "Sulfuras", being a legendary item, never has to be sold or decreases in Quality
- "Backstage passes", like aged brie, increases in Quality as it's SellIn value approaches; Quality increases by 2 when there are 10 days or less and by 3 when there are 5 days or less but Quality drops to 0 after the concert

We have recently signed a supplier of conjured items. This requires an update to our system:

- "Conjured" items degrade in Quality twice as fast as normal items

Feel free to make any changes to the UpdateQuality method and add any new code as long as everything still works correctly. However, do not alter the Item class or Items property as those belong to the goblin in the corner who will insta-rage and one-shot you as he doesn't believe in shared code ownership (you can make the UpdateQuality method and Items property static if you like, we'll cover for you).

Just for clarification, an item can never have its Quality increase above 50, however "Sulfuras" is a legendary item and as such its Quality is 80 and it never alters.

Source: https://github.com/emilybache/GildedRose-Refactoring-Kata

Except where otherwise specified, all content © 2015-2016 Gargoyle Software Inc

Appendix B: Rules for the elephant carpaccio exercise

Instructions

1. Break into teams of 2-3 people, one workstation per team.
2. Preparation - Each team writes down on paper the 10-20 demo-able user stories ("slices") they will develop and possibly demo. Each should be doable in 3-8 minutes. No slice is just mockup of UI, creation of a data table or data structure. All demos show real input & output (not test harness).
3. Discussion - Instructor/facilitator leads discussion of the slices, what is and isn't acceptable, solicits ways to slice finer.
4. Development - A fixed time-box of 40 minutes, five 8-minute development sprints, clock does not stop. At the end of each sprint, each team shows its product to another team.
5. Debrief

Product

Accept 3 inputs from the user:

- How many items
- Price per item
- 2-letter state code

Output the total price. Give a discount based on the total price, add tax based on the state and the discounted price.

Order value	Discount rate
$1000	3%
$5000	5%
$7000	7%
$10000	10%
$50000	15%

State	Tax rate
UT	6.85%
NV	8%
TX	6.35%
AL	4%
CA	8.25%

Source: http://blog.crisp.se/2013/07/25/henrikkniberg/elephant-carpaccio-facilitation-guide

Except where otherwise specified, all content © 2015-2016 Gargoyle Software Inc