

Technical Practices Workshop

# Facilitator Guide

# Overview

Hands-on introduction to key software engineering practices that support an Agile and Lean approach to solution development and delivery.

## Duration

Three days

## Authors

Dave Nicolette, LeadingAgile Technical Practie Lead

Jann Thomas, author of the original version of the workshop

# Technical Prerequisites

Each computer that will be used in the technical exercises needs the following:

* Public Internet access (not firewalled corporate network)
* The Java IDE they will use at work
* If they're using Ruby Cucumber, the Ruby IDE they will use at work
* Git client (if not provided by the IDE)
* MySQL database system installed locally
* Ruby 2.3.1 or later installed locally

# Room Setup

## Option 1: Randori setup

* One workstation at the front of the room connected to an overhead projector.
* Workstation is on a table suitable for two people to sit side-by-side.
* Workstation has public internet access.
* Audio system with three wireless microphones.
* Remainder of room configured for amphitheater-style seating such that participants can clearly see and hear the pair working at the front of the room.
* Movable furniture so we can make a large space in the room (or a large space available).

## Option 2: Pairing setup

* Small tables (preferred) or long tables (if necessary) oriented so that people can sit side-by-side and share one laptop per pair, and everyone can see the front of the room.
* One presenter setup with a connection to an overhead projector.
* Audio system with one wireless microphone.
* All participants must have public internet access from their laptops.
* Movable furniture so we can make a large space in the room (or a large space available)

## General requirements

* Whiteboards and markers or flip-charts and markers
* Sticky flip chart paper and markers, and wall space to display them
* Sticky notes and markers
* Colored markers, tape, yarn, stickers (for decorating name tents)
* Printer paper
* Standard size mailing envelopes
* Prepared sticky notes or index cards for tracking progress in the workshop (like story cards)
* Story card sets for the FizzBuzz exercise
* 100 pennies
* Collaboration-Handouts.pdf printed out
* Eight (8) or Nine (9) tennis balls
* Legos (no particular bricks or pre-made sets)
* Sign-in sheet (if printed)
* Evaluation forms (if printed)
* Printed handouts for the Collaboration activity - Collaboration-Handouts.pdf
* Candy or cheap toys for prizes

## Facilitator room setup

*Task Board*

Set up a card wall or task board using the prepared "story card" sticky notes or index cards. Organize the board to mimic the kind of board the teams will use in their work, based on whether they're using a time-boxed process (e.g., Scrum), a continuous-flow process (e.g., Kanban), or a hybrid process. If you're mimicking a time-boxed process, decide on an iteration length (like 30 minutes, for instance) and update the board accordingly. If you're mimicking a continuous-flow process, set it up in a "to do -> doing -> done" style and update the board each time you pull a ticket. When updating the board, ask for volunteeers to come up and move the tickets rather than doing it yourself.

*Burn Chart*

Put a sticky flipchart sheet on the wall (or designate space on a whiteboard) and draw a grid on it. You decide whether to show progress in the workshop as a burn-up or burn-down.

*Parking Lot*

Put a sticky flipchart sheet on the wall (or designate space on a whiteboard or blank wall) to serve as the Parking Lot. Label it "Parking Lot" (but don't write directly on a blank wall).

*Team Goals*

Put a sticky flipchart sheet on the wall (or designate space on a whiteboard or blank wall where the group will record their goals for the workshop. Label it "Team Goals" (but don't write directly on a blank wall).

*Name Tents*

Put the name tent cards on the tables along with colored markers, stickers, yarn, tape, and whatever else you are providing for decorating the tents.

*Toys and supplies for activities*

Place these in plain sight somewhere in the room but don't explain their purpose. Let people wonder about it as they arrive.

# Background Notes for Facilitator

## Pedagogical methods

The workshop structure makes use of the following methods but does not follow any one method exclusively:

* Experiential Learning - see https://en.wikipedia.org/wiki/Experiential\_learning
* Spaced Repetition - see https://en.wikipedia.org/wiki/Spaced\_repetition

You're free to use your own approach and style, but try to avoid heavy lecture or reading slides aloud, and try not to explain things in detail before participants have had a chance to experience them. General approach is "feel it first, then debrief."

## Agile and Lean concepts and practices

These concepts are baked into the workshop structure and aren't introduced through lecture or slides:

* Team agreement
* Self-organization
* Definition of done
* Card wall or task board
* Feedback
* Empirical planning
* Relative sizing
* Iterative and incremental
* Value focus
* Collaboration
* Dot voting
* Collective ownership
* Chart - Burn Chart or Cumulative Flow Diagram, depending on method
* Retrospective
* WIP limits
* Pull
* Three Cs
* Pairing
* Test-Driven Development
* Refactoring
* Continuous improvement

In the Facilitator Guide, key concepts like these are indicated in square brackets like this:

[Chart]

[Collective ownership]

to help you focus your interaction with the group at key points.

# Outline

## Day 1 Morning

### 00:00-0020 Getting Started

* Name Tents (activity)
* Team Agreement (activity)
* Team Goals (activity)
* Why are you here? (activity)
* Parking Lot

### 00:20-01:00 Section 1: Business Drivers

* Predictability (activity)
* Quality (activity)
* Speed (activity)
* Value (activity)

### 01:00-01:45 Collaboration

* Collaboration game

### 01:45-02:15 Self-Organization

* Ball Point game

### 02:15-02:45 Transparency and Trust

* Transparency and Trust game

### 02:45-03:00 Day 1 Morning Retrospective

* Facilitated retrospective

### 03:00-03:45 Verify technical setup

* Ensure everyone has what they need installed

## Day 1 Afternoon

### 00:00-00:45 TDD Kickstart

* Greenfield TDD round 1 (FizzBuzz)
* Debrief
* Explanation and demonstration

### 00:45-01:45 TDD for Greenfield Development

* Story decomposition
* Greenfield TDD round 2 (FizzBuzz)
* Debrief

### 01:45-02:00 TDD Introduction - Retrospective

* Facilitated recap and retrospective on the TDD intro

### 02:00-02:10 Compass Reading

* Basecamps and technical practices

### 02:10-02:25 Test Automation Pyramid

* Lecture and discussion

### 02:25-02:55 Continuous Delivery

* Lecture and discussion

## Day 2 Morning

### 00:00-02:00 Refactoring Kickstart

* Gilded Rose round 1
* Debrief
* Lecture and demonstration
* Gilded Rose round 2

### 02:00-02:30 Agile Planning

* Multiple Planning Horizons

### 02:30-03:00 Agile Requirements

* Big Design Up Front (BDUF)
* Insufficient Design
* Just Enough Design Initially (JEDI)
* Personas (Activity)
* Story Mapping introduction

## Day 2 Afternoon

### 00:00-1:00 Story Mapping

* Activity

### 01:00-03:30 Specification by Example

* Concrete vs. abstract descriptions (Activity)
* Reading Gherkin (Activity)
* What is Specification by Example?
* Writing Gherkin (Activity)
* Value of Specification by Example
* Demonstration of test automation (World's Smallest Bookstore)

### 03:30-04:00 Day 2 Retrospective

* Facilitated retrospective

## Day 3 Morning

### 00:00-00:30 Git Introduction

* Basic interaction with the local git repository
* Basic interaction with the remote git repository
* Semantic versioning

### 00:30-03:00 Working on the World's Smallest Bookstore

* Pomodoro-based randori or pair develpment of User Stories

## Day 3 Afternoon

### 00:00-03:30 Working on the World's Smallest Bookstore

* Continue working User Stories for the World's Smallest Bookstore

### 03:30-04:00 The Rest of the Story

* Platform-as-a-Service (PaaS)
* Continuous Delivery Again

### 04:00-04:15 Section 7: Name Tent Contest

* Vote on the name tents and award prizes

### 04:15-04:30 Workshop Retrospective

* Were expectations met?
* Did you learn anything useful?
* What can we do to improve the workshop?

# Day 1 Morning

### 00:00-00:20 Getting Started

Suggestion: Do not go around the room and have each participant say their name and their role one by one. No one ever remembers any of that. It burns time and kills energy.

***Name Tents***

[Task board]

[Definition of done]

Without explaining what the task board is for, pull the Name Tents ticket and move it to *In Progress*.Tell the group you will know when this ticket is done when you can see all their names when you look around the room.

While they work on their Name Tents, explain logistical details for the workshop such as location of restrooms, sign-in sheet (or link), evaluation form (or link), etc. Don't dictate anything about the use of phones or laptops. That will be handled as part of the Team Agreement (coming up soon).

[Self-organization]

Ask each participant to write their name in large letters on a folding "tent" sheet and place it in front of them. Provide colored markers and any other decorative items you please, and encourage people to decorate their name tents in any way they wish. People can work on this at any time during the day.

[Collective ownership]

[Definition of done]

Whenever it appears that most of the people have finished their name tents, make a show of looking at them and making sure you can see them all. If you can't see them all, pretend you don't know what to do so participants will help each other move their name tents into a position where you can see them. This may not occur in sequence (before the next activity). Just let it happen whenever it happens.

***Team Agreement***

[Collective ownership]

[Self-organization]

Ask for a volunteer to come up and move the current ticket to Done and to pull the next ticket.

[Chart]

At the end of each "iteration" or "cadence" (if doing continuous-flow), update the Burn Chart or CFD.

***Note:*** Updating the task board and chart is not written out for you after every step in the Facilitator Guide, but the intent is that you will do it after every step.

[Team agreements]

[Collaboration]

Without explaining the connection between this activity and the concept of team agreements, ask participants to agree on certain things for the workshop, such as:

* use of electronic devices during sessions
* timing of breaks and lunch
* start and stop times for each day
* protocol for asking questions
* etc.

Write the group's agreements in the Team Agreements area you prepared.

***Team Goals***

[Self-organization]

[Collective ownership]

Stick a sheet on the wall or reserve a section of a whiteboard, and title it "Expectations". Ask participants to come forward and write one or more expectations they have for the workshop, or to make a mark next to an expectation that someone has already written. This is called "expectations" and not "goals" because we want to encourage people to be honest if they have low or negative expectations. We'll retrospect on these points later.

***Why are you here?***

[Feedback]

Use the ESVP method to get a sense of why people are present. Pass out sticky notes and ask each participant to write one capital letter on one sticky note:

* E = Explorer, interested in learning whatever might be learned from the workshop
* S = Seeker, interested in learning something specific from the workshop
* V = Vacationer, interested in being away from their desk for a few hours
* P = Prisoner, present only because their manager told them to attend

Each person brings you their sticky note anonymously. You count up the number of sticky notes in each category and write the number somewhere visible. This will be a point for retrospection at the end. In the meantime, it gives you an idea of how you can interact with the room. A room full of Explorers will have a very different dynamic than a room full of Prisoners.

***Parking lot***

[Feedback]

[Self organization]

Show participants the Parking Lot and explain how they will use it. It's up to you whether they put sticky notes on it, write on it, or just ask questions and you manage it yourself.

### 00:20-01:00 Business Drivers

*Slide: Business Drivers*

Lecture content. Don't go into any depth. Just set the stage for activities. Provide explanations in the debrief after each activity.

*1.1 Why do we care about building software in any particular way?*

* Predictability
* Quality
* Speed
* Value

***Predictability***

*Slide: Predictability*

* Primary goal for Basecamp 1
* Empirical forecasting
  + empirical = by observation
  + forecasting = predicting near-term future performance based on demonstrated near-term past performance

**ACTIVITY (PREDICTABILITY)**

[Empirical planning]

Ask for three volunteers. They are the Team. Bring them to the front of the room or to some place in the room where everyone can see them. They will need a table or other flat surface to work on.

Give them a pile of pennies. Ask them to estimate how many pennies they can flip over in one minute. Write down their estimate. Start a timer and let them flip pennies. Write down the number they flipped.

Challenge them to improve on their performance. Ask for an estimate for another one minute of penny-flipping. Start a timer. This time, interrupt them. Reject some pennies for "quality" reasons. Remove one team member to work on a "production issue."

Ask for a third estimate. This time interrupt them a couple of times but not too much, and don't pull anyone off the team.

For the fourth round, ask them to estimate. Also derive a forecast from their performance in the first three rounds. Interrupt them as much as you want. Compare their performance with their estimate and with the forecast. Debrief.

**ACTIVITY (PREDICTABILITY) ENDS**

***Quality***

*Slide: Quality*

Lecture content

* Explicitly defined, not assumed
* Quality attributes are part of acceptance criteria and definition of "done"
* Acceptance criteria include description of how quality will be measured
* Teams can demonstrate their code meets quality objectives at any time

**ACTIVITY (QUALITY)**

[Collective ownership]

[Collaboration]

[Feedback]

[Value focus]

Ask participants to suggest things software teams can do that contribute to quality. It can be anything that a team as a whole can do as well as anything a person in a specialized role can do, including analysts, testers, programmers, DBAs, operations, etc. It need not be anything they currently do or observe in their organization; just anything they think would contribute to quality. Write down their ideas on a large sticky sheet or on a whiteboard.

Studies that examined the correlation between defect density in production code and specific software development and testing practices found a correlation between high quality (by that measure) found two specific things about software development correlated strongly with low defect density. Ask the group to speculate about what those two things were.

Answer:

* Code reviews
* Small source code modules

Suggest that some software engineering practices that we recommend actually help to minimize the overhead involved with code reviews and help keep the size of source code modules from growing too large. There's no need to tell them which practices they are. We'll be introducing those practices as we go along.

**ACTIVITY (QUALITY) ENDS**

***Speed***

*Slide: Speed*

* Not the same as running faster on the same treadmill you've been running on
* Reduce lead times by avoiding waste and applying good practices
* Eliminate process waste
  + Stop doing useless things. Just stop.
  + Minimize the cost (time) of unavoiable non-value-add activities
* Build in quality to avoid re-work
  + There is no trade-off between quality and speed. It's false economy.

**ACTIVITY (SPEED)**

[Collaboration]

[Collective ownership]

[WIP limits]

Envelope-stuffing game or equivalent. Divide the room into groups of 5. Each group comprises:

* Folder
* Stuffer
* Addresser
* Inspector

Ask for one person to serve as Timekeeper.

For each round, the Timekeeper starts a 5 minute timer and says "Go." At the four-minute mark, the Timekeeper says "One minute!" At the 4:30 mark, the Timekeeper says "30 seconds!" The Timekeeper counts down the last 10 seconds aloud and says "Stop!"

Round 1: Specialists. The Folder folds five sheets of paper and then passes them to the Stuffer. At that point, the Folder can begin to fold the next set of five sheet. The Stuffer puts the folded sheets inside envelopes. When five envelopes are stuffed, the Stuffer passes the five envelopes to the Addresser. At that point, the Stuffer can work on the next five. The Addresser writes an X on the outside of each envelope. When 5 envelopes have been marked, the Addresser passes them to the Inspector. At that point, the Addresser can work on the next set of 5. The Inspector examines each envelope and decides whether the result is acceptable. He or she places each envelope in an "accepted" pile or a "rejected" pile. Rejected envelopes are returned to the Folder, who must stop what he/she is doing and re-fold the sheet. Record the number of accepted envelopes.

Round 2: Generalists. Each person folds, stuffs, labels, and inspects the envelopes one at a time. At the end, each team writes down the number of envelopes that were accepted.

Debrief: The group discusses which approach was more effective and why. Ask for ideas on how the same principle could apply to software work.

**ACTIVITY (SPEED) ENDS**

***Value***

*Slide: Value*

* Financial value is measured by dividing the return by the cost
* There are other forms of value for customers, stakeholders, employees
* Long-term value to the enterprise includes improving the organization's ability to deliver effectively

**ACTIVITY (VALUE)**

[Collaboration]

[Collective ownership]

[Self-organization]

[Value focus]

Ask participants to identify different stakeholders/customers of their work. Write down each one on a whiteboard or flip chart sheet, leaving space under or next to each one.

If participants do not think of any of the following, add them yourself and explain as you go:

* customers
* internal stakeholders
* yourself
* the company as a business entity
* the community
* the world

Next, ask participants to suggest different forms of *value* that pertain to each category of stakeholders. Write those down next to or under the stakeholders.

**ACTIVITY (VALUE) ENDS**

### 01:00-01:45 Collaboration

**ACTIVITY (COLLABORATION)**

[Collaboration]

[Feedback]

[Collective ownership]

James Shore's "Offing the Offsite Customer" game.

See Collaboration-Presenter.pdf and Collaboration-Handouts.pdf. The handouts must be printed in advance.

**ACTIVITY (COLLABORATION) ENDS**

### 01:45-02:15 Self-Organization

**ACTIVITY (SELF-ORGANIZATION)**

[Self-organization]

[Collaboration]

[Continuous improvement]

[Feedback]

Ball Point Game. See Ball\_Point\_Game.pdf.

**ACTIVITY (SELF-ORGANIZATION) ENDS**

### 02:15-02:45 Transparency and Trust

**ACTIVITY (TRANSPARENCY)**

[Collaboration]

[Collective ownership]

[Value focus]

[Continuous improvement]

Ask for six volunteers. Divide them into two teams of three. Give most of the Lego blocks to Team A, and two blocks to Team B.

Arrange the two teams so that neither can see what the other team is building with the blocks.

Team A has a backlog of two items. The higher-priority item has a dependency on Team B. The lower-priority item has no external dependencies.

Team B has a backlog of one item. It is to assemble their two bricks. This assembly is required for Team A to complete its high-priority backlog item.

In each "iteration" the teams can add one block to one of their products.

Secretly instruct Team B not to assemble their bricks and to conceal their progress from Team A.

At the end of each iteration, Team A determines which backlog item to work on in the next iteration. They ask Team B how things are going, and Team B always reports they are on track to complete their work on time.

After *n* iterations, stop and see how much value the organization has delivered. Most likely, Team A elected to work on their high-priority item because Team B was telling them their dependency would be completed on time. If so, then the organization as a whole didn't deliver any value.

Debrief. Discuss trust.

**ACTIVITY (TRANSPARENCY) ENDS**

### 02:45-03:00 Day 1 Morning Retrospective

[Feedback]

[Collaboration]

[Collective ownership]

[Continuous improvement]

Facilitate a retrospective using whatever format you prefer, or a format that the coaches are encouraging these teams to use. Let people express what they thought was useful or not useful, what went well, what could be improved, etc.

Refer to the Parking Lot and address any questions and comments there.

Also connect the dots between some of the key concepts and the activities the people experienced in the morning. Refer to the Chart and discuss how it shows progress and blockers. Refer to the Team Goals and Team Agreements and solicit feedback from the group about how well or poorly those are being achieved.

### 03:00-03:45 Verify technical setup

The rest of the workshop will focus on technical activities. Help the group get the starter project downloaded and ensure their tools are installed and their internet access is working. They need *public* internet access, not their corporate network.

Each computer that will be used in the software development activities needs:

* Public Internet access (not firewalled corporate network)
* Git client
* The IDE they will use at work
* MySQL database system
* The FizzBuzz starter project (http://github.com/neopragma/java-fizzbuzz)
* The World's Smallest Bookstore starter project (http://TBD)

# Day 1 - Afternoon

### 00:00-00:45 TDD Kickstart

*No slides*

The purpose of this section is to "shock the water" with respect to TDD. The reason for approaching it this way is that everyone already has some level of understanding about TDD. In an organization where technical coaching is needed, the technical staff's understanding of TDD is likely to be incorrect and inconsistent. Participants in the workshop will come to the table with a set of preconceptions and (possibly) closed minds. So the first step has to be to pull people out of their comfort zone.

[Pull]

[Iterative and incremental]

[Collaboration]

[Pairing]

[Test-Driven Development]

[Feedback]

**Important**

* Don't explain what TDD means beforehand. Let participants act initially on their current understanding.
* Don't explain the mechanics or good practices for pairing.
* Don't tell them explicitly that they can ask you clarifying questions. They *can* do so, but wait and see if they think of it.
* Don't explain anything about decomposing stories into tasks.
* Don't explain anything about frequent commits to version control.
* Don't tell them all the usual requirements for FizzBuzz. The user stories that have been prepared for the exercise will spoon-feed the requirements to them in a way that is designed to support the experiential aspects of the exercise.

**Development round 1 (randori)**

The initial setup for this is randori-style even if we're using pairing for the rest of the exercises. We'll do the next to activities in this format.

* Ask for two volunteers who have not used TDD before. (If everyone has done TDD before, ask for people who don't use it often or who feel they don't understand it deeply.) They will be the first pair, working at the front of the room on a computer connected to an overhead projector.
* Set a timer for 10 minutes.
* The first story card has a lot of acceptance criteria on it. Let the first pair work with it on their own. Debrief later.
* The starter project has a single example to serve as an example of how to write examples, but it isn't the example we would want as the first red-green-refactor pass.

**Debrief**

*Gut feel impressions*

Immediately after the initial round, ask participants to write single adjectives that describe their feelings about what they just experienced or observed, writing *a single adjective on each sticky note* and using as many sticky notes as they need. Give them a 60-second time limit.

Collect the sticky notes and affinity-group them on the wall.

*Discussion*

Guide participants toward thinking about the practical effects of TDD and pairing that they just experienced or observed, as well as how those techniques made them feel, and steer them away from moralistic or philosophical comments about whether TDD is a "good practice" or not. Connect their comments with the adjectives they wrote in the immediate aftermath of the first pairing session. When interesting questions and comments come up that are outside the immediate scope of this section, add them to the Parking Lot.

* What about the TDD process made it easy or hard to complete the work?
* What about pairing made it easy or hard to complete the work?
* In what way(s) did the experience make you feel competent or productive or effective?
* In what way(s) did the experience make you feel frustrated or confused or ineffective?
* What effects did these methods have on the speed and quality of delivery?
* What do you think could be done differently (better) next time?
* If you visualize using these techniques in your real work, in what ways do you see them helping or hindering effective delivery?
* (Additional questions based on how the group responds, at your discretion)

**Explanation and demonstration**

Briefly explain there's more to effective TDD than an *ad hoc* approximation of the red-green-refactor cycle. It's a technique that supports *emergent design*. Emergent design is a process of driving out the low-level technical design of software by running fine-grained, concrete examples of interactions with the API and ensuring the results of those interactions are as expected.

There are different approaches to TDD depending on whether we are writing new code or modifying existing code. To accomplish that goal effectively when developing new code, as we're doing in this exercise, practitioners have learned that it's best to start with the simplest possible implementation and gradually add complexity, taking care not to implement any more behavior than is called for by the examples.

To extend or refine the behavior of the implementation, we design more and more examples. The rule of thumb is: As the examples become more specific, the implementation becomes more general.

*Slide: Transformations*

Display the slide that shows the code transformation priorities and leave it up while participants work the rest of the exercise.

Using the results of the first pair as a starting point, demonstrate the process of emergent design of FizzBuzz through fine-grained examples. Start by deleting the implementation that was done before.

Switch between the slide and the code as you show how to build up some of the FizzBuzz functionality in this manner. Go through a few of the transformations step by step. As you work, commit to the local git repository frequently and call that out as a recommended practice.

Opportunistically point out when you are changing hats between programmer thinking and tester thinking to be sure you aren't leaving holes in the design.

Finish the demonstration by deleting your implementation and examples and committing.

### 00:45-01:45 TDD for Greenfield Development

This section continues with the FizzBuzz project.

**Story decomposition**

*No slides*

Walk the group through a lightweight decomposition process to identify the sequence of steps to test-drive the implementation of the first FizzBuzz user story. Refer to the transformations as needed to keep them from going overboard. Result is a simple list of steps on a whiteboard or flip chart sheet, such as:

* handles null input
* echoes the number entered
* returns "Fizz" for 3
* returns "Fizz" for 9

and so on at that level of detail, nothing more elaborate. They'll be getting another story for round 3 and they will be able to decompose it on their own.

**Development round 2 (randori or in pairs)**

*Slide: Transformations*

Set a timer for 10 minutes.

Let participants work through the first couple of FizzBuzz stories. If some people get ahead, there are some additional stories they can pull.

**If using the randori format:** At the end of each 10 minute time-box, one person retires and another member of the group joins in at the front of the room. After each time-box, the person who has been at the front for two consecutive turns retires. Each person stays up there for two turns. If there is only one overhead projector, you may have to switch to the Transformations slide from time to time for clarification.

**If using the pairing format:** At the end of each 10 minute time-box, tell participants that one person in each pair must now shift to the workstation to their right, with the right-most person wrapping around to the left-most workstation. Leave the Transformations slide up.

Let participants work through the first FizzBuzz user story. As each pair completes a story, let them pull the next user story, decompose it, and work it.

Depending on time available and your perception of the group's learning, stop the exercise after about four rounds or when you feel it's time to stop to ensure there will be time for the remaining work of the day.

### 01:45-02:00 TDD Introduction - Retrospective

Facilitate a short retrospective about the TDD exercise so far. Connect the dots with Team Goals and Parking Lot items, as appropriate.

Ask for participants' ideas about what would make TDD challenging when they're working with an existing code base instead of writing new code. Capture their ideas on a whiteboard or flipchart sheet for later reference.

Ask how participants' previous understanding or assumptions about TDD have been challenged by their experiences in this exercise.

### 02:00-02:10 Compass Reading

Now we'll give participants a break from heads-down coding, provide some contextual information, and also avoid starting a long activity late in the day.

Lecture content

*Slide: Basecamps & Technical Practices*

Briefly review the LeadingAgile Basecamp model, which may already be familiar to the participants, and describe the part of the journey their teams are taking in their current expedition.

Mention that there is no one-for-one correlation between Basecamps and technical practices. We identify the capabilities an organization must have in order to opertate at each Basecamp, and we back up from there to the point where the organization is currently in order to determine when to begin introducing technical practices.

It take a long time for organizations to internalize and master these practices. For that reason, it's necessary to *begin* introducing them well ahead of the time when the organization plans to reach a Basecamp where such practices are a prerequisite to success.

Put this into context according to where the teams and the organization are. This will vary from one group to another.

### 02:10-02:25 Test Automation Pyramid

Lecture content

*Slide: Test Automation Pyramid*

Describe the purpose and scope of tests and checks at each level of the pyramid.

Put it into context based on what you know about the way teams currently work in the organization.

### 02:25-02:55 Continuous Delivery

Lecture content

*Slide: Continuous Delivery*

Describe the concept of Continuous Delivery (CD) and supporting concepts and practices like Continuous Integration, DevOps, automated deployment, feature toggles, dynamic infrastructure, Platform as a Service, elastic cloud, containerization, canary deployment, blue/green deployment, configuration drift, and phoenix servers.

Briefly demonstrate the World's Smallest Bookstore application to show how it starts in Github and flows through various cloud-based services to end up in production on Heroku. This will be the application they work with tomorrow, when they apply good practices to existing code.

# Day 2 - Morning

### 00:00-02:00 Refactoring Kickstart

*No slides*

The Gilded Rose Refactoring Kata is one of the most popular refactoring katas. The scene is set in a fictional establishment in Stormwind City, a locale from the game World of Warcraft (see http://wowwiki.wikia.com/wiki/Gilded\_Rose). The gnarly code, however, is disturbingly real. We're going to have the participants in the workshop tackle it without any preparation.

**Important: Before the first round...**

* Don't explain anything about refactoring
* Don't ask participants anything about their previous experience with refactoring
* Don't demonstrate refactoring

Have participants clone the Git repo for the Gilded Rose Refactoring Kata:

**git clone https://github.com/emilybache/GildedRose-Refactoring-Kata.git**

The Java version is in subdirectory Java.

Have participants import the project into their IDE. They can see if it was installed correctly by running the unit tests:

**mvn test**

They should see output like this:

**Failed tests: foo(com.gildedrose.GildedRoseTest): expected:<f[ixme> but was:<f[oo]>**

**Tests run: 1, Failures: 1, Errors: 0, Skipped: 0**

The requirements for the exercise are documented in the root directory of the downloaded project (not in the Java subdirectory) under the name GildeRoseRequirements.txt.

**Gilded Rose round 1 (randori or in pairs)**

**If using the randori format:** Read the requirement aloud to the group and solicit questions.Don't spend more than about five minutes on this. Set the timer for 5 minutes and rotate someout out of the pair after each 5 minute interval. Let them work for 5 rounds (25 minutes).

**If using the pairing format:** Give everyone five minutes to read over the requirements and discuss them with their partners. Set the timer for 12 minutes. After 12 minutes, rotate people around so that everyone is working with a different partner. Set the timer for another 12 minutes.

**Round 1 debrief**

**If using the randori format:** Walk through the code and discuss the results.

**If using the pairing format:** Ask for two volunteer pairs (one at a time) to plug into the projector and show their code. Walk through the code and discuss the results.

Ask the group what they thought was challenging about the exercise.

**Lecture and demonstration**

*Slide: Working with existing code*

Ways to get started with existing code.

*Slide: The Squint Test*

Info: https://www.youtube.com/watch?v=8bZh5LMaSmE

After Sandi Metz: Squint your eyes and look at the code. You're looking for changes in shape and changes in color. Changes in shape mean there are nested conditionals, which are always hard to reason about. Changes in color (in an IDE) mean the code is at different levels of abstraction, which is always hard to reason about.

Things to squint about:

* if statements
* !=
* != with &&
* magic strings
* magic numbers

Also from Sandi Metz: People have a tendency to follow whatever pattern they see. Most people are afraid to add new objects, etc. The first impulse in changing the code is to look for a pattern similar to what we want to implement and add more code at that point. If the pattern is a good one, we get away with it. If the pattern is a bad one, we exacerbate the problem.

*Slide: Important!*

You don't have to understand the code in order to refactor it. Refactoring means changing the structure of code without changing its behavior. You can see and change structure without understanding the "meaning" or "purpose" of the application.

*Slide: Not Anna Karenina*

"Happy families are all alike; unhappy families are unhappy in their own way." (Leo Tolstoy, *Anna Karenina*).

Bad code is all alike; good code bases are good in their own way. When you can deal with any bad code, you can deal with all bad code.

*Slide: The Golden Master*

Switch to the code. Open src/test/java/com/gildedrose/TesttextFixture.java and show the code. Execute the code and show the output. Describe how you could extend this and use it as a crude check to be sure you haven't ruined everything as you proceed with refactoring. It's just one approach. It isn't required.

*Slide: Finding Seams*

Switch to the slides.

The idea of "seams" comes from Michael Feathers' book, *Working Effectively With Legacy Code* (highly recommended). The metaphor is that if you ripped your clothes off, they would likely come apart wherever there are seams rather than in the middle of fabric panels. Code is the same way. It's easier to split it apart along its natural seams than it is to chop it up in a random way.

Switch to the code.

Show src/main/java/com/gildedrose/GildedRose.java. Walk through the code and ask participants where they see seams. Do an *extract method* refactoring to pull something out, like an else block or whatever looks convenient to you. Explain what you're doing and why.

Switch to the slides.

*Slide: Making Seams*

Sometimes it's hard to discern where the natural seams are in a long procedural routine. It's possible there aren't any natural seams.

Switch to the code.

Note that this much of the code:

public void updateQuality() {  
 for (int i = 0; i < items.length; i++) {  
 if (!items[i].name.equals("Aged Brie")  
 && !items[i].name.equals("Backstage passes to a TAFKAL80ETC concert")) {  
 if (items[i].quality > 0) {  
 if (!items[i].name.equals("Sulfuras, Hand of Ragnaros")) {  
 items[i].quality = items[i].quality - 1;  
 }  
 }

looks like "Aged Brie," "Backstage passes," and "Sulfurus" appear to be special in some way. Otherwise, quality is decremented by one each cycle.

So we could write a test case, modeled on the existing one, for a single Item that isn't one of the special ones, and see if its quality is decremented by one:

@Test  
public void qualityDecrementsByOne() {  
 Item[] items = new Item[] {  
 new Item("foo", 2, 50)  
 };  
 GildedRose app = new GildedRose(items);  
 app.updateQuality();  
 *assertEquals*(49, items[0].quality);  
}

That passes. So, we've made our own "seam" and we've started to get tests around the existing code. This is only one approach and isn't required.

One of the most fundamental software design principles is DRY - don't repeat yourself. The idea is to minimize the amount of duplicate code in the system. Sometimes we relax that rule in the name of performance or clarity, but most of us have a deeply-ingrained instict to eliminate duplication the moment we see it.

When we're in the midst of refactoring and writing test cases to explore the current behavior of an existing code base, duplication can be helpful. It can help expose the underlying algorithm that's hiding in a mass of conditional logic. So we want to refrain from "cleaning up" the duplication too early.

Switch to the slides

*Slide: Sensing Variables*

Michael Feathers coined the term "sensing variable" in his book, *Working Effectively With Legacy Code*, but he probably didn't invent the concept. One of the things we normally do is to minimize the number of local variables in our code, unless statements become so concise and dense that they're hard to understand.

In the context of exploring and refactoring existing code that lacks tests, we may do exactly the opposite of that, even if only temporarily. We might introduce a variable just so that we have something to look at, either in the debugger, on the console, or in a test case.

Switch to the code.

Let's say we were curious to see what sorts of items fell into the last else block in the updateQuality() method:

} else {  
 if (items[i].quality < 50) {  
 items[i].quality = items[i].quality + 1;  
 }  
}

We might add a display, like this:

} else {  
  
 //*TODO remove this!* System.*out*.println("Last else, item name: " + items[i].name);  
   
 if (items[i].quality < 50) {  
 items[i].quality = items[i].quality + 1;  
 }  
}

to help us see what's going on.

After adding that display, run TexttestFixture.java. Review the output, and show that the sensing display doesn't appear in the output. What does that tell us about the quality of the test fixture?

**Gilded Rose round 2**

Let participants work five rounds of 10 minutes on the Gilded Rose.

### 02:00-02:30 Agile Planning

Switch to the slides.

*Slide: Planning is Indispensable*

The old saying about the importance of planning is a fundamental part of agile thinking. Agile teams plan almost continuously.

*Slide: Multiple Planning Horizons 1/5*

Long-term planning dovetails with medium-term and short-term planning. The Product Backlog drives planning. It's kept in priority order by key stakeholders with input from delivery teams. Looking far into the future, we plan at a high level only.

*Slide: Multiple Planning Horizons 2/5*

We take a subset of the top-priority items from the Product Backlog and elaborate them in further detail to prepare for the next few months of work.

*Multiple Planning Horizons 3/5*

Following the same pattern, we focus on the top of the prioritized Product Backlog to prepare the highest value items for execution.

*Multiple Planning Horizons 4/5*

To keep the queue ready with the right amount of work for each development cadence or iteration, we flesh out most of the details of the top priority items on the backlog just in time for each cadence or iteration.

We don't want to overplan, or we'll be wasting our time on details that are likely to change. This creates the Lean waste known as "inventory."

We don't want to underplan, or delivery teams will run out of work that is ready to do. This is called *queue starvation*, and it creates the Lean waste known as "delay."

*Multiple Planning Horizons 5/5*

When delivery teams start work on a set of work in a cadence or iteration, they continue the process of continuous planning at a fine-grained level. It helps them focus on the most valuable aspects of the stories in flight, to keep themselves aligned with stakeholder needs, and to steer their efforts toward delivery of value.

### 02:30-03:30 Agile Requirements

*Slide: Big Design Up Front (BDUF)*

Traditional approach has been to define every detail of a software solution before beginning development.

*Slide: Insufficient Design*

Immature agile teams sometimes try and begin work before they have enough information to understand what is really needed, or how to approach the design.

*Slide: Just Enough Design Initially (JEDI)*

The idea of just enough design initially, or JEDI, originated with practitioners of a method called Feature Driven Development (FDD). The idea is that we want to begin to produce working subsets of the solution as early as possible based on a high-level, but more-or-less accurate concept of what stakeholders are looking for.

We want to balance the cost of *rework* due to overdesign and the cost of *delay* in getting something into stakeholders' hands.

To that end, we want enough information to *get started* with development, and we want to do things in such a way that changes in direction will not be too costly. That philosophy applies to all aspects of the work:

* analysis
* requirements
* planning
* measurement
* tracking
* architecture
* design
* programming
* testing
* packaging
* deploying
* releasing
* monitoring
* supporting

*Slide: Personas 1/2*

Explain what Personas are and how they fit into the overall requirements and analysis process.

*Slide: Personas 2/2*

Describe the kinds of information that are usually included in a Persona and walk through creating one interactively with the group, writing information on a whiteboard or flip chart sheet.

*Slide: World's Smallest Bookstore*

Present an overview of the World's Smallest Bookstore system. This will be the system participants work with. There is a partially-complete code base and a set of features that stakeholders would like to have them implement. You can show some of the code to give them an idea of where things are in the code base, but there's no need to go into great depth at this point. They mainly need to get a general idea of the product so they can define reasonable Personas for it and do reasonable Story Mapping.

**ACTIVITY (PERSONAS)**

[Collaboration]

[Collective ownership]

[Value focus]

[Continuous improvement]

[Feedback]

Divide the participants into small groups. Ask each group to develop a Persona representing someone who would use the World's Smallest Bookstore. Set a time limit of 15 minutes.

After the 15 minute time box, ask each group to show their Persona to the group and explain it. Then ask the group to discuss the Persona and offer suggestions to improve it.

**ACTIVITY (PERSONAS) ENDS**

### 03:30-04:00 Story Mapping Introduction

*Slide: Decomposition*

Through multiple-horizon planning, big ideas are decomposed into smaller pieces until we get things down to the level that we can build them. At shorter time horizons, the delivery teams are more closely involved with elaborating the details of the work items.

*Slide: Story Mapping 1/5*

Walk them through the basic flow of Story Mapping: Goal => Activity => Task => Story. In this context, the word "task" doesn't mean the same thing as it does when wer're breaking down User Stories into small tasks. It's a "thing to do" while engaged in an Activity. So, if an Activity is *Browse through product category tree*, a Task might be *View list of products for a selected category.*

*Slide: Story Mapping 2/5*

The Goal, *Find Product*, could be Find a book in the World's Smallest Bookstore system.

*Slide: Story Mapping 3/5*

Any number of Activities might support the Goal, *Find Product*. Here we have a single example, *Browse through product category tree*.

*Slide: Story Mapping 4/5*

Tasks are smaller things you do while engaged in an Activity. While engaged in *Browse through product category tree*, you might *Choose a product to view details*.

*Slide: Story Mapping 5/5*

Ultimately, *User Stories* pop out of the model. They support the Tasks, or sometimes directly an Activity or a Goal depending on the scope.

*Slide: Story Map Example*

A Story Map is a living model that changes frequently as people collaborate to improve their understanding of the problem and potential solutions. Once the model starts to settle down, people often start to carve out sections of it as targets for release. The most valuable bits and pieces of all the stories will be at the top of the model. Choosing the scope to include in the next release is often as simple as drawing a horizontal line under the row of stories we want to deliver in the release.

# Day 2 - Afternoon

### 00:00-01:00 Story Mapping Practice

Talk through the Features that have been defined for the World's Smallest Bookstore. Each group gets the same set of Story Cards to work with in the exercise.

**ACTIVITY (STORY MAPPING)**

[Collaboration]

[Collective ownership]

[Value focus]

[Continuous improvement]

[Feedback]

Divide the participants into large-ish groups, or have the whole group work together if there aren't too many people. Ask each group to develop a Story Map based on the Features presented to them. They are free to add Features that they think of, too.

Set a time box of 30 minutes. Walk around and help them stay on track.

Spend 30 minutes with each group explaining their Story Map to the group and debriefing. Have the group agree on combining their results into a single Story Map. This will guide their work on the code.

**ACTIVITY (STORY MAPPING) ENDS**

### 01:00-03:30 Specification by Example

*Slide: Specification by Example (ants)*

\*\* Don't show the next slide until after the exercise \*\*

**ACTIVITY (REQUIREMENTS)**

[Collaboration]

[Collective ownership]

[Value focus]

[Continuous improvement]

[Feedback]

15 minutes

Divide the participants into small groups. Give each group the handout on Traditional Requirements and Test Plans. Ask them to read the handout and discuss among themselves how easy or hard it is to understand what is to be done and how to test it, as well as how useful the document would be as a long-lived artifact to help future teams understand and work with the application.

15 minutes

Have each group share the outcome of their discussion with the other participants. Group discussion.

**ACTIVITY (REQUIREMENTS) ENDS**

*Slide: Concrete vs. Abstract 1/2*

Here is an abstract description of something. What kind of thing is it?

*Slide: Concrete vs. Abstract 2/2*

Here are some specific examples of a kind of thing. What kind of thing is it?

Which of the two representations of the thing did you find easier to recognize? Why do you think that is so?

When you were reviewing the traditional requirements specification, test plan, and technical documentation, do you think it would have been easier to understand in the form of concrete examples?

*Slide: Given-When-Then*

This is a form people use to express concrete examples of interactions between people and computer systems or between two computer systems.

*Given* expresses preconditions - things that are assumed to be true at the outset.

*When* expresses the action a person (or client system) takes to interact with a system.

*Then* expresses postconditions - observable or measurable results of the interaction.

\*\* Don't show the next slide until after the exercise \*\*

**ACTIVITY (READING GHERKIN)**

[Collaboration]

[Collective ownership]

[Value focus]

[Continuous improvement]

[Feedback]

15 minutes

Divide the participants into small groups. Give each group a set of the sample Gherkin scenarios. Have people discuss the scenarios within their group and organize them along a spectrum of Tautolotical => Domain Language => Scripty => Technical. You can write this spectrum on a whiteboard or flip chart.

15 minutes

Have each group call out one or two of the scenarios, which category they placed it in, and why. Group discussion.

**ACTIVITY (READING GHERKIN) ENDS**

**ACTIVITY (WRITING GHERKIN)**

[Collaboration]

[Collective ownership]

[Value focus]

[Continuous improvement]

[Feedback]

15 minutes

Divide the participants into small groups. Have each group write scenarios for the User Stories that came out as top priority in the Story Mapping exercise, for the World's Smallest Bookstore.

15 minutes

Have each group call out one or two of the scenarios, which category they placed it in, and why. Have the group reach consensus to combine their results into a single set of scenarios.

**ACTIVITY (WRITING GHERKIN) ENDS**

*Slide: Value of Specification by Example*

Emphasize the majority of the value comes from the interactive communication among stakeholders. There is also value in having a form of documentation that is always in sync with the code, and that is shared and understandable by all interested parties regardless of their technical or business background.

Many scenarios can be automated, which adds value to the practice. Relate this back to the test automation pyramid.

Switch to the code (World's Smallest Bookstore).

Show the group what the Gherkin scenarios look like and walk them through the connection between the scenarios and the step definitions. Run the scenarios and show them the output.

### 03:30-04:00 Day 2 Retrospective

Facilitate a retrospective of the day's activities. Connect the dots with any outstanding Parking Lot items.

# Day 3 Morning

### 00:00-00:30 Git Introduction

*Slide: Trunk-Based Development*

Explain trunk-based development principles and practices. Explain that we're going to use this strategy in the workshop.

*No slide*

Explain Semantic Versioning. This is the recommended way to version artifacts.

Major.Minor.Patch. Minor changes are backward compatible within Major release number. Patch is a bug-fix, also backward compatible within major release number.

**ACTIVITY (GIT)**

Walk participants through the steps to perform basic operations against their local git repos, such as

* git status
* git add
* git commit
* git push
* resolving merge conflicts

Tell them to work on the master branch and not to create additional branches.

**ACTIVITY (GIT) ENDS**

### 00:30-03:00 Working on the World's Smallest Bookstore

*Slide: Bookstore Application*

Give participants a brief overview of the structure of the Bookstore application.

*Slide: Optional UI Application*

*Slide: Testing Tools*

Give participants a brief overview of the tools included in the development setup. Show them the code, run examples, etc.

*Time to work*

Have the participants work the User Stories according to the priority they came up with in earlier exercises, and based on the acceptance criteria they defined.

By intent, they are starting work without very much of an introduction. Help them find their bearings in the code by using tests to explore the current functionality so they can get started on the right track.

Work in 25-minute Pomodori with a 5-minute break after each one. Switch pair partners after each Pomodoro. Reinforce fundamental software design principles opportunistically and in context. Don't make an abstract "presentation" about it. (Details like that will be reinforced during ongoing coaching. Participants won't be able to internalize them during the workshop.)

**If using the randori format:** Let the pair at the front of the room work with the rest of the participants speaking up as needed. When you see the need, offer explanations and suggestions.

**If using the pairing format:** Walk around and help them find seams, break dependencies, and remediate existing code as well as test-driving all new and modified code at all levels of abstraction.

Last 30 minutes: Debrief and walk through the code. Discuss the design decisions that were made.

# Day 3 Afternoon

### 00:00-03:30 Working on the World's Smallest Bookstore

Continue working User Stories for the World's Smallest Bookstore

### 03:30-04:00 The Rest of the Story

We've covered the front end of the delivery process:

* team/stakeholder collaboration
* team ownership of the work
* self-organization
* agile requirements
* collaborative working methods - pairing, mobbing
* test-driven development at multiple levels
* working with existing code

The complete delivery pipeline comprises additional steps, all of which can be automated (in principle).

*Slide: Platform-as-a-Service (Paas)*

Today, there is rapid movement toward outsourcing operations in the form of "cloud" services. At the same time, there is growing emphasis on automation of every aspect of software delivery that is routine and repetitive, to free humans for creative, value-add work.

*Slide: Continuous Delivery Again*

Walk through this process and explain to participants how each of the steps could be realized in a cloud-based continuous delivery pipeline, focusing on the technologies they use (or that they are planning to use).

### 04:00-04:15 Section 7: Name Tent Contest

[Dot voting]

[Feedback]

Place a sticky note or sheet of paper next to each Name Tent. Ask participants to go around the room and dot vote for their favorites. Each person gets three dots which they can allocate any way they please. At the end, count the dots and announce the winners. Award candy or cheap toys as prizes.

### 04:15-04:30 Retrospective

*Were expectations met?*

Revisit the list of expectations that participants came up with at the beginning of the workshop. Discuss whether and how each one was met or not met.

*Why did you stay?*

Revisit the ESVP numbers from the beginning of the workshop. Ask if anyone would change theirs now that they have experienced the workshop, and why.