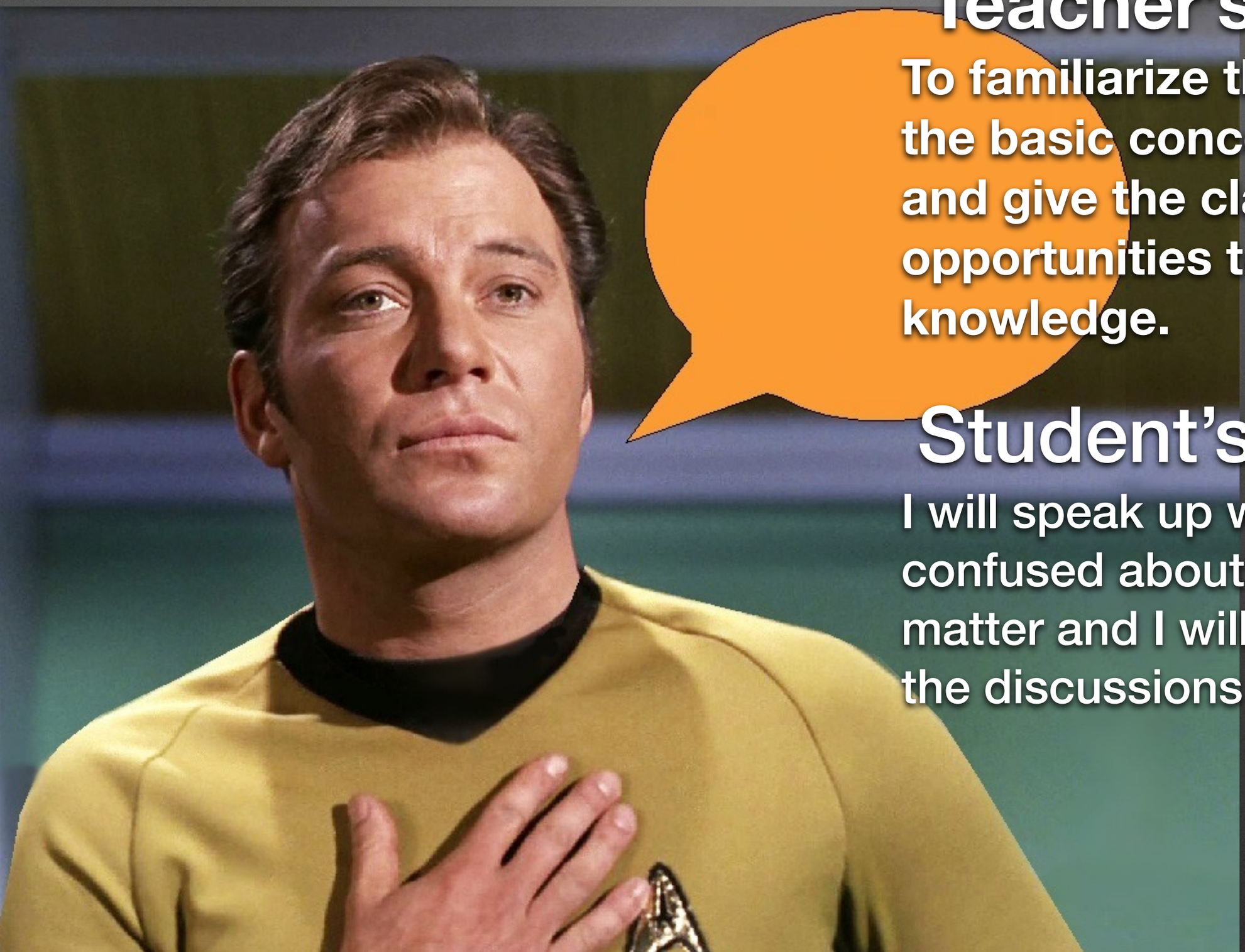
# Test Driver’s Ed

A Crash Course on Test Driven Development

# Howdy!

A little bit about ourselves

# How about you?



## Teacher's pledge:

To familiarize the class with the basic concepts of TDD and give the class opportunities to apply that knowledge.

## Student's pledge:

I will speak up when confused about the subject matter and I will engage in the discussions in good faith.

# What is TDD?

What do you think?

# Test Driven Development Definitions

- “Test driven development (TDD) is a software development approach in which a test is written before writing the code. Once the new code passes the test, it is refactored to an acceptable standard.” - Technopedia

- “Test-Driven Development (TDD) is a technique for building software that guides software development by writing tests…
  - Write a test for the next bit of functionality you want to add.
  - Write the functional code until the next test passes.
  - Refactor both new and old code to make it well structured.”
  - Martin Fowler

# Where did this come from?

- Early on in computing, computations were very costly. Manual error checking was critically important.
- Smalltalk in the early 90s, featuring Kent Beck
- Beck ports SUnit to Java, creating JUnit; popularity explodes.
- Unit testing frameworks have been created for virtually any language you can think of.

# The Recipe!

- Test!
  - Describe in the native language of the program what it is supposed to do.
- Make it work!
  - Make your program actually DO the thing described!
- Refactor!
  - Dress up your code like a professional.

# Test!

- Write in English what you want your program to do.
- Translate those requirements into the native language of your program.
- Run the test!
- It should FAIL!
- Confirm your test fails for the *correct* reason, and that it plainly describes that reason.

# Make it work!

- Write the least amount of code possible to make your test requirements pass. If you think of new requirements while implementing, convert those requirements into tests… later.

- Run your tests again! If you succeeded at implementing the feature, all of your tests should pass!

# Refactor Without Mercy!

- Massage your code until you're satisfied that it is understandable at a glance.
- Keep running your tests as you refactor to make sure everything works as intended!
- Ask others for input as to what is confusing about your code, and refactor accordingly!
- Do your best not to leave code in a state you'll be ashamed of later. You're a professional!

# Rinse, Repeat!

As you incrementally add features to your program, do all of the steps again!
Test, Make it Work, and Refactor!

# That's all great but… I'm still stuck.

- Having trouble getting started usually means you need to spend a little more time thinking about how to organize your requirements
- Try writing in English the responsibilities you'd like to give to each part of your system, and a general overview of how your systems work together. Informally diagram it when possible!
- Do your best to make your Classes reflect the Single Responsibility Principle:
- "The single responsibility principle states that every context (class, function, variable, etc.) should define a single responsibility, and that responsibility should be entirely encapsulated by the context. All its services should be narrowly aligned with that responsibility." - Wikipedia.

# Kata!

Fizz Buzz.

# Test Morphology

It takes all kinds.

# Unit Tests

- The nuts and bolts of testing
- Demonstrating what a function does.
- Generally isolated from other systems are costly to use (speed, capacity)
- Fast, fast fast!

# Functional / Integration

- Do the components of my system work correctly when assembled together in a realistic way?
- Frequently, functional tests can reflect very clearly the business requirements
- No nitty gritty details about how low level systems work; these should reflect highly realistic problems.

# End to End

- Tests that use your entire system, in a way the user would encounter it. Examples:
  - Does my program correctly display the data in the right section of the screen?
  - Are all of the buttons accessible?
- Does a user's entire workflow work correctly? Example:
  - Open a file, make some changes, do work.
- Slow! These are the highest cost tests to run. ESPECIALLY if they are not automated.
- Failures in end to end tests identify problems that should be fixed at the functional and unit test levels.

# Kata!

Integration Kata

# Stuff happens.

How do I test with complex, expensive, unreliable or uncontrollable components?

# Problem:

You're running a magazine company that's been losing subscribers.

Conceptually design something that will send an email to 'at-risk' customers.

# Fakes, Mocks, Spies and Stubs

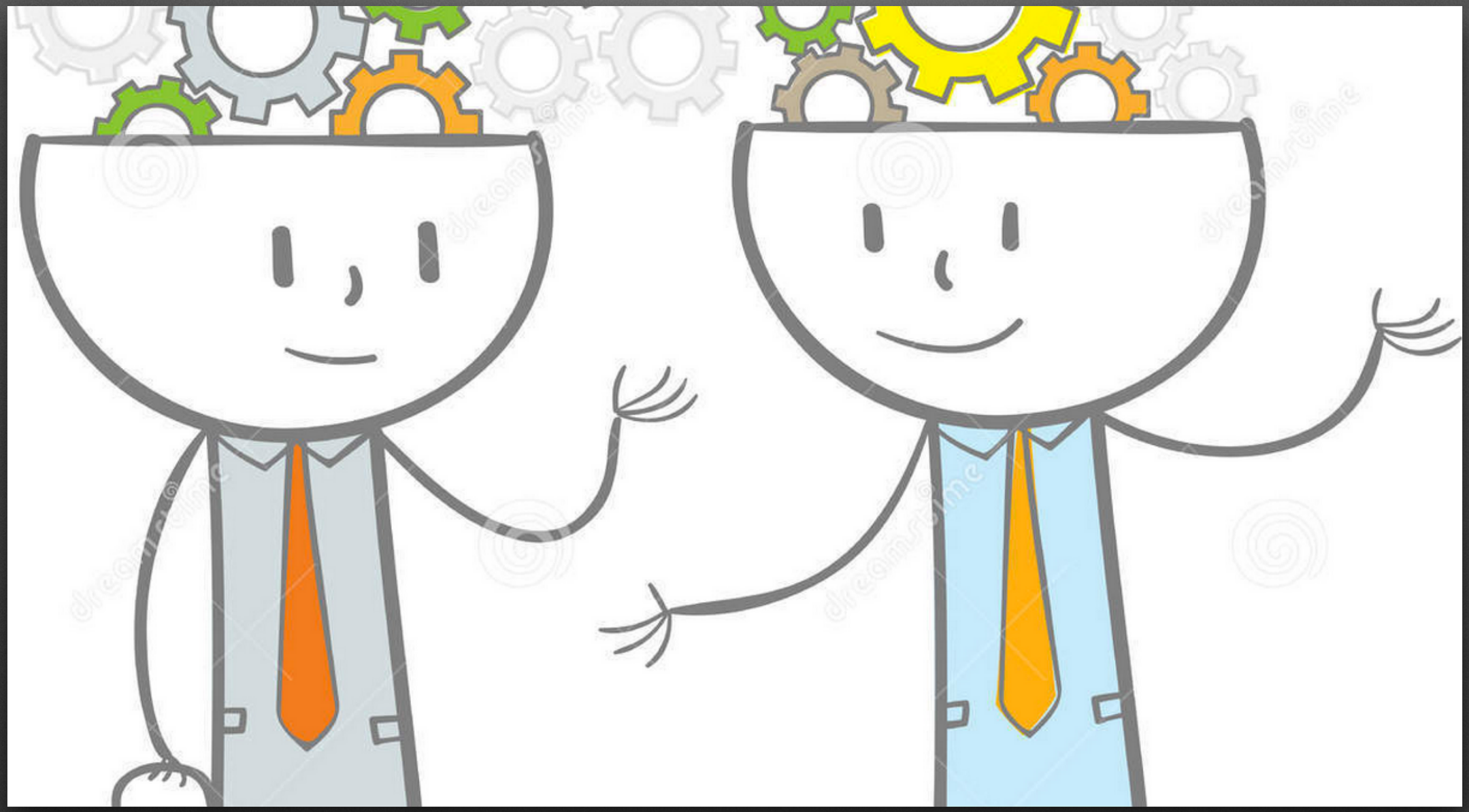
- Many words to describe the same (or very similar) concept.

“Mock objects are simulated objects that mimic the behavior of real objects in controlled ways. A programmer typically creates a mock object to test the behavior of some other object, in much the same way that a car designer uses a crash test dummy to simulate the dynamic behavior of a human in vehicle impacts.” - Wikipedia

# Kata!

Renewal Mock Kata

# Beware Mock Pollution!



# Discussion!

Lets talk about how today went.

# Supercharged refactoring

Keeping your code fit

# Is your code Mature? or just Old….

- That greenfield feeling
- After the honeymoon
- What problems have you observed? Why does it happen?

# Staying in shape with TDD

- Let TDD be your guide to keeping things simple
- Understanding pain : What are my tests trying to tell me
  - Tests are long and hard to read
  - One change breaks hundreds of tests
  - Random failures
  - Tests are slow
  - Hacking around tests

# A clean workspace is a safe workspace

- Refactor mercilessly. Good thing we have tests.
- IDE makes it easy
- Never read something twice. If you have to work to understand a section, use a refactor to improve readability
- Readability test. One look should be enough

# Refactors : Difficult to Read Code

- Rule of thumb
- Well named methods and variables
- Logic block with method
- Minimize non-final variables

# Refactors : Repeated Code

- Extract method and reuse
- Create parameter object for related values
- Consolidate repeated values (Constants)
- Extract Utility/Service

# Refactors : Excessively heavy methods

- Replace switch or if statements with micro strategies
- Replace repeated if statements sprinkled throughout code with an object to represent that state

# Kata!

RefactorLegacyKata

# Integration.

Come together, right now.

# How often should teams integrate their code?

- Lets talk about it!

# Make Integration Continuous

- Continuous integration (CI) is the practice of merging all developer working copies with a shared mainline several times a day.
- The goal of CI is to find integration problems as soon as possible… possibly before a feature is complete!
- Never be ashamed to integrate refactoring… in fact, do right away so other programmers can benefit!

# Integrating Incomplete Features

- Never, ever integrate failing tests.
- Plan your changes such that all of your programming is ‘under the hood’ and will not affect a user’s experience
- Create all the features you need, but do not actually connect them to the program. Integrate with the codebase, but not with the application.
- Use a feature toggling strategy to disable your feature until its showtime

# Kata!

Roman Numerals

# Code Archaeology

What kinds of problems do you have when you inherit a system with large amounts of untested code?

The more practiced you are with your refactoring toolkit, the better an archaeologist you will be.

# Kata!

Gilded Rose

# Myths, gotchas, watch out for…

- Don't sweat too much about your test coverage statistic! TDD is about *driving* your code. Thinking in a test first way is an act of DESIGN.
- The power of testing really kicks in when the team can be COMPLETELY confident that their code works as intended. If they don't seem confident, discuss which tests should be added that will help.
- Compromising your test coverage in times of crisis will undermine the team's ability to trust the whole system.
- Be willing to reorganize your code along 'test driven' lines; separate new untested code into different modules. Otherwise that untested code will undermine trust in the suite.
- Test driving is not about creating perfect code; its about asking your programmers to actively and concretely design.

# That’s all folks!

How did we do?