OVERVIEW OF TEST DRIVEN DEVELOPMENT

In this lecture we're going to talk about what Test Driven Development is and how it helps to ensure you're writing high quality and bug free code.

WHAT IS TEST DRIVEN DEVELOPMENT?

- A process where the developer takes personal responsibility for the quality of their code.
- Unit tests are written *before* the production code
- Don't write all the tests or production code at once.
- Tests and production code are both written together in small bits of functionality.

So what is test driven development (or TDD)?
<click> TDD is a process for writing code that helps you take personal responsibility for the quality of your code.
<click> The process drives this by having you write the unit tests *before* the production code. This can seem pretty strange at first but after you've used the process for a while it becomes the norm and you'll find it hard to write code any other way.
<click> Even though the tests are written before the production code, that doesn't mean that ALL the tests are written first. You write one unit test for one test case and then you write the production code to make it pass.
<click> The tests and production code are written together with small tests being written and then small bits of production code that make those tests pass. This short cycle of writing a unit test and then writing the production code to make it pass provides immediate feedback on the code. This feedback is one of the essential features of TDD.

WHAT ARE SOME OF THE BENEFITS OF TDD?

- Gives you the confidence to change the code.
- Gives you immediate feedback
- Documents what the code is doing.
- Drives good object oriented design

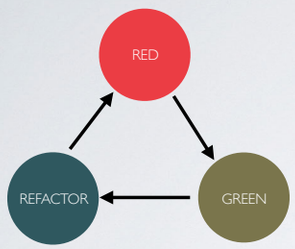
So what are some of the benefits of using TDD?
<click> TDD gives you confidence to make changes in your code because you have a test before you begin that verifies the code is working and will tell you if any of your changes have broken anything.
<click> This confidence comes from the immediate feedback the tests provide for each small change in the production code.
<click> The tests document what the production code is supposed to do. A new developer looking at the code can use the unit tests as a source of documentation for understanding what the production code is doing.
<click> Writing the unit tests first helps drive good object oriented design as making individual classes and functions testable in isolation drives the developer to break dependencies and add interfaces rather than linking concrete implementations together directly.

TDD BEGINNINGS

- Created by Kent Beck in the 1990's as part of the Extreme Programming software development process.
- He wrote the first TDD unit testing framework in Smalltalk called SUnit.
- Collaborated with Erich Gamma to implement the first Java unit testing framework JUnit.
- JUnit has been the basis for many other xUnit testing frameworks written for other languages.

- TDD was created by Kent Beck in the mid 1990's as part of his work for the Chrysler corporation where he also created Extreme Programming (of which TDD is just a part).
- <click> He then went on to create one the first unit testing framework for TDD in Smalltalk called SUnit
- <click> Then he collaborated with Erich Gamma to implement the first Java unit testing framework called JUnit
- <click> JUnit has since been the basis for many other xUnit testing frameworks implemented for other languages.

TDD WORK FLOW: RED, GREEN, REFACTOR



- TDD has the following phases in its work flow:
  - Write a failing unit test (the RED phase)
  - Write just enough production code to make that test pass (the GREEN phase)
  - Refactor the unit test and the production code to make it clean (the REFACTOR phase).
  - Repeat until the feature is complete.

The TDD workflow is broken up into three phases referred to as the Red Phase, Green Phase, and Refactor phase.
<click> The first phase is the RED phase. In the RED phase you write a failing unit test for the next bit of functionality you want to implement in the production code.
<click> Next comes the GREEN phase where you write just enough production code to make the failing unit test pass.
<click> Last is the REFACTOR phase where you clean up the unit test and the production code to remove any duplication and make sure the code follows your team's coding standards and best practices.
<click> Then you repeat the process for all the functionality you need to implement and all the positive and negative test cases.

UNCLE BOB'S 3 LAWS OF TDD

- You may not write any production code until you have written a failing unit test.
- You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
- You may not write more production code than is sufficient to pass the currently failing unit test.

- Robert Martin aka "Uncle Bob" created the following Laws of TDD in his book "Clean Code: A Handbook of Agile Software Development". These laws help ensure you're following the TDD process.
- <click> The first law "You may not write any production code until you have first created a failing unit test" follows along with the idea of writing the unit tests first. Seeing your new unit test fail before you've implemented the production is a sign that the unit test has been implemented properly.
- <click> The second law "You may not write more of a unit test than is sufficient to fail" forces you to write only enough of a unit test for the next test case. And the next test case should always be the *simplest* test case.
- <click> The last law "You may not write more production code than is sufficient to pass the currently failing unit test" keeps you from writing production code without any unit test to verify it.
- These three laws help to keep you in a small tight loop of writing a little test that fails, then writing a little production code to make it pass. Each iteration of the loop should only be a few minutes long and you're always running the unit tests to verify nothing has gotten broken. If something does get broken you can easily backout the changes that caused the problem because you implemented them in just the last couple of minutes!
- In the next lecture we'll go through a working example to see TDD in action!