Automated Testing, Test Driven Development

Assignment

-

- For next week Listen to
- Fallthrough epidode one-ish
  - https://podtail.com/podcast/fallthrough/war-stories/
- Project1
  - Any questions on the current sprint?

Automated Testing

- Test Driven Development vs Automated Testing
  - Let's have a lucky volunteer or few help explain the difference between these two?

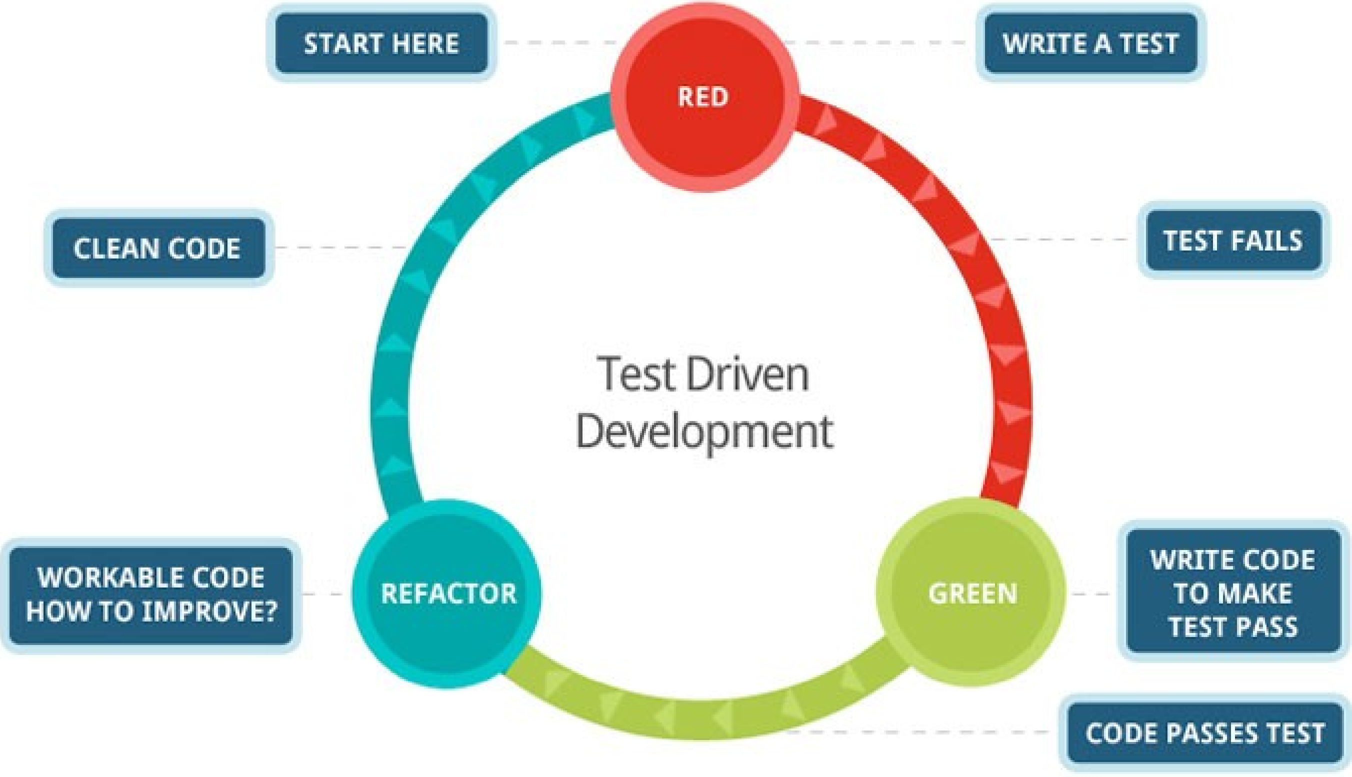
Test Driven Development

- Today Test Driven Development means at least you write the tests before the production code that they test
    - Failing tests before code
    - Then write code
    - When tests pass then your software is done.

And the Original TDD

- The Original TDD
  - And the purist version even today
- Write one test, let it fail, then write the production code to make it pass
- The write one more test, and then make it pass, repeat
  - Want to write a webapp?
    - Before you do anything – including installing the web app libraries
      - Write a test.
      - When it fails do something.
  - So purist: write test, and only write real code when test fails.
- What seems like it might go wrong here?

TDD The Original Way



Automated Tests

- Pretty much every serious software project uses Automated Tests today
  - Code that evaluates the "production code" and run automatically by the CI system
    - And should be run the the programmer on their local machine first.
  - May or may not exercise the entire code base, but does test/exercise at least part of it.
- Not everyone believes in TDD
  - But yes to automated tests.

Kinds of Tests

- There are several ways to classify tests
- One Categorization that is used fairly commonly
  - Unit tests
  - Functional tests
  - Acceptance tests
  - What are each of these? What do they do?

Types of Tests

- Unit tests
  - Item by item – function by function tests
- Functional tests
  - Does the app do what it is supposed to do?
- Acceptance tests
  - Does the app do what the client thinks it is supposed to do?

Why?

- So what are the tests supposed to do for us in Test Driven Development or other methods of using automated tests?
  - Why has Testing (TDD?) become so accepted in the last 10-15 years?
    - Going from something more avant-garde that many managers resisted to "table stakes" at most software development places?
  - Well actually some people still call it TDD but ‘automated tests’ might be a better term
  - What does Automated testing buy us? (especially with CI)

Why?

- So what are the tests supposed to do for us in Test Driven Development or other methods of using automated tests?
    - Why has Testing (TDD?) become so accepted in the last 10-15 years?
        - Going from something more avant-garde that many managers resisted to "table stakes" at most software development places?
    - Well actually some people still call it TDD but ‘automated tests’ might be a better term
    - What does Automated testing buy us? (especially with CI)
    - Tests are run every time code is compiled/interpreted.
    - Tests become an extension of the compilers ability to catch errors.
    - Always better to let the compiler catch the error.
    - Why?
        - What does it buy us?

TDD vs Automated Tests

- Very few people do old-school TDD today
- But the automated tests technique are still valuable
- Automated tests of some sort are more or less mandatory today.
- Turn your specs into tests
  - Unit tests
  - And functional tests
  - Write them,
  - Then write the code (or the other way around)
  - Then run the tests
  - Every time you change anything and build
    - Run all tests again

Assignment

- For those of you new to automated testing
  - Read a couple of introductions
    - https://katalon.com/resources-center/blog/what-is-automation-testing
    - https://medium.com/tenable-techblog/automation-testing-with-pytest-444c8b34ead2
  - And a quick look at doing some of this in pytest (we'll look at some examples later)
    - https://bas.codes/posts/python-pytest-introduction
  - For those of you who have done automation tests before let's move on

Unit Tests

- First an easiest tests to understand/automate are Unit Tests

- Testing Smallest Testable part of application
  - Functions, methods, etc
  - Sometimes the entire public interface to a class
  - Extend compiler's error checking capability.
- Traditionally each unit test should be done in isolation
  - Even if your class relies on a database, mock database and test class
  - Recently lots of conference talks pushing back against mocks, tests on each unit will include its dependencies
    - We'll see if this takes

Unit/Automated Tests

- There are libraries/packages to support automated tests in nearly every important language
- Java : JUnit (the granddaddy of all)/Mockito/cucumber
- Python : pytest (and older unittest and nose)
- C++ : Catch 2, google-test, unittest++
- C# : Mstest
- Newer language like Go and Rust:
  - Tools are built in to the language tooling, no library or framerwork required

Pytest: the current preferred python test framework

- pip install pytest
  - I suggest through pycharm unless you have a linux distro with a package manager.
  - <file><settings> menu (or <pycharm><preferences> or Mac)
  - Then choose the project item from the left list
    - And the project interpreter
    - Then push the '+' icon to add a package
    - From there select pytest and install it.

Best Practices

- For best practices,
  - Have a separate test directory
  - Create a new directory as a subdirectory in your project
- Lets call it tests.

What sorts of tests

- What sorts of tests should we write?
  - Many people suggest at least as much test code as production code

What sort of Tests

- What sorts of tests should we write?
  - Remember that many people suggest at least as much test code as production code
  - Want ‘happy path’ tests
    - When all data is as expected
  - Want bad data tests
    - When we enter junk
    - c.f little bobby tables
  - Especially want to check unusual values
    - Like the (in)famous $0 billing statements
  - Eventually want to try restricting resources
    - Simulate network outage for example.

- The first/easiest automated tests
  - Test a single function that computes a value
  - Usual starting demo online
  - Lets take a look at the TestingDemo project that I have on github
    - https://github.com/jsantore/TestingDemo
  - Let's write a couple of automated tests for the simpler functions
  - that automated test should find ‘error’

Another Test

- So the first happy path tries some easy wins
  - 3,4,5 triangle
  - Then we add in floating point answers
  - But floating point has precision and rounding issues for repeating decimals and irrational decimals
    - you’ve heard this since CS1
    - Now we run into it with these tests
  - For floating point numbers in pytest use
    - Pytest.approx(<expected number>, <acceptable tolerance>)
    - Eg
    - assert pretendProductionCode.simple_distance(0, 0, 6, 5) == pytest.approx(7.81024967590, .000001)

JUnit Equivelent of Pytest Approx

- JUnit provides an equivalent
  - public static void assertEquals(double expected,
  - double actual,
  - double delta)
  - Version without delta is deprecated
- Example:
  - double myPi = 22.0d / 7.0d; //Don't use this in real life!
  - assertEquals(3.14159, myPi, 0.001);
    - From: https://stackoverflow.com/questions/5939788/junit-assertequalsdouble-expected-double-actual-double-epsilon
  -

The save function

- Let's try to test the output function
  - Let's look at the two options
  - And then test the one that can be tested.

Testing on github

- Lets use secrets on github
  - We will use the github secrets mechanism to create a file in the ephemeral docker container during testing that will disappear after the github actions are done
    - The container along with everything ever on it is gone
  - "To create secrets for a user account repository, you must be the repository owner. To create secrets for an organization repository, you must have admin access."
  -

Adding a Secret to github

- To add a secret to github
  - On GitHub.com, navigate to the main page of the repository.
  - Under your repository name, click Settings.
  - In the left hand side menu in the security section open the secrets and variables menu
    - Then pick actions
  - The secrets tab is active by default, in the upper right is a green button called "new repository secret" push it
  - Name your secret (name requirements next slide)
  - Put your secret (no quotes!) in the secret text box

Github's rules for naming secrets

- Secret name rules
  - Names can only contain alphanumeric characters ([a-z], [A-Z], [0-9]) or underscores (_). Spaces are not allowed.
  - Names must not start with the GITHUB_ prefix.
  - Names must not start with a number.
  - Names are not case-sensitive.
  - Names must be unique at the level they are created at.

Building the secrets file

- My file called api_secrets.py is in my gitignore, so I want to rebuild it in the ephemeral docker container in github actions
- I called my secret LLM_API_KEY, and in my github actions I put the following between Install dependencies and linting
- My api_secrets.py needs a line like
  - gemini_api_key='<my key here>'

```
- name: Build Secrets
  env:
    API_KEY: ${{ secrets.LLM_API_KEY }}
  run: |
    echo 'gemini_api_key = "'$API_KEY'"' >> api_secrets.py
```

In context

- Here I did this for my version of the project
- https://github.com/jsantore/Project1ProfDemoPython2025/blob/master/.github/workflows/python-app.yml
- You would have to echo slightly different (more complex) things into the file for go (several lines more complex for java) but it works the same way.
  - For those if you with multi line secrets/env files, you would need multiple echo lines
  - For more information
  - https://unix.stackexchange.com/questions/77277/how-to-append-multiple-lines-to-a-file
-

Now run the tests

- Now that everything is set up, I like to run the tests by replacing the simple
  - pytest
- Line that is in the default github actions test runner with
  - python -m pytest tests/*
- Which will run all tests in all python files in the tests subfolder

Back to better tests

- Now that we have everything we need for sprint2
  - Lets add a little more to our ability to build tests

Accepting Exceptions

- Sometimes you want your code to throw an exception
  - Want you automated tests to expect those
  - Lets look at code in class
  -
  - In test:

    -
      ```
      with pytest.raises(TypeError):
          pretendProductionCode.add_interest("4", .05)
      ```
  -

Test Coverage

- Want to have your production functions do proper error checking and sanity checking
  - Want your tests to cover a full suite of possibilities
- Should add checks for 0 and 1 at least to the test suite.
  - Edge cases
  - Maybe a really big number too
    - At least for java and go and other fixed width number language
    - Python's Bignum class is a little different

JUnit version of expecting exception

- In java/junit
- Use ‘decorators’
  - @Test(expected = IndexOutOfBoundsException.class)
  - public void empty() {
  - new ArrayList<Object>().get(0);
  - }

Testing and Design

- Variety of philosophies about production code and testing
  - Oldie and still used – but less and less commonly:
    - Production code is what produces value for the company so it is the focus
  - TDD/BDD influenced
    - Build production code to be easier to test
  - More and more we see:
    - Need enough tests to be reasonably sure that new commits didn't break anything from before.

Build code to be easy to test

- Recommendation: build code to be easy to test
  - Generally it is better code
  - Clean code/Lack of code smells etc.
- If you write the entire project in the main function
  - It might work
  - But it is hard to maintain and extend
  - And impossible to test.
  - Story of a former student and my colleague's semester-long quest to break bad habits.

A hard to test function

- Some functions are hard to test:

-
  ```
  def show_output():
      #this is hard to test
      initial_bal = 300
      balance = add_interest(1000, 0.025)
      print(f"Your new balance is ${balance}")
  ```

- How can we test this function?

Hard to test

- How can we test `show_output` From the last slide?
  - We could do some crazy shell programming
  - Or Monkey-patch print and then put it back
  - Or we could write a testable function in the first place.
  - Suggestions?

Easier To Test

- We can make the printing easier to test by taking another parameter.

-
  ```
  def testable_show_output(initial_bal, rate, outfile):
      balance = add_interest(initial_bal, rate)
      if not outfile:
          outfile = sys.stdout
      print(f"Your new balance is ${balance}", file=outfile)
  ```
- So now when called from your production code, print prints to the screen as normal,
  - But we can write tests to have it print to a file
- With Java you can take a param of type PrintStream
  - Production code uses System.out (which is just a prebuilt PrintStream)