Introduction to Software Testing

Test Driven Development (TDD)

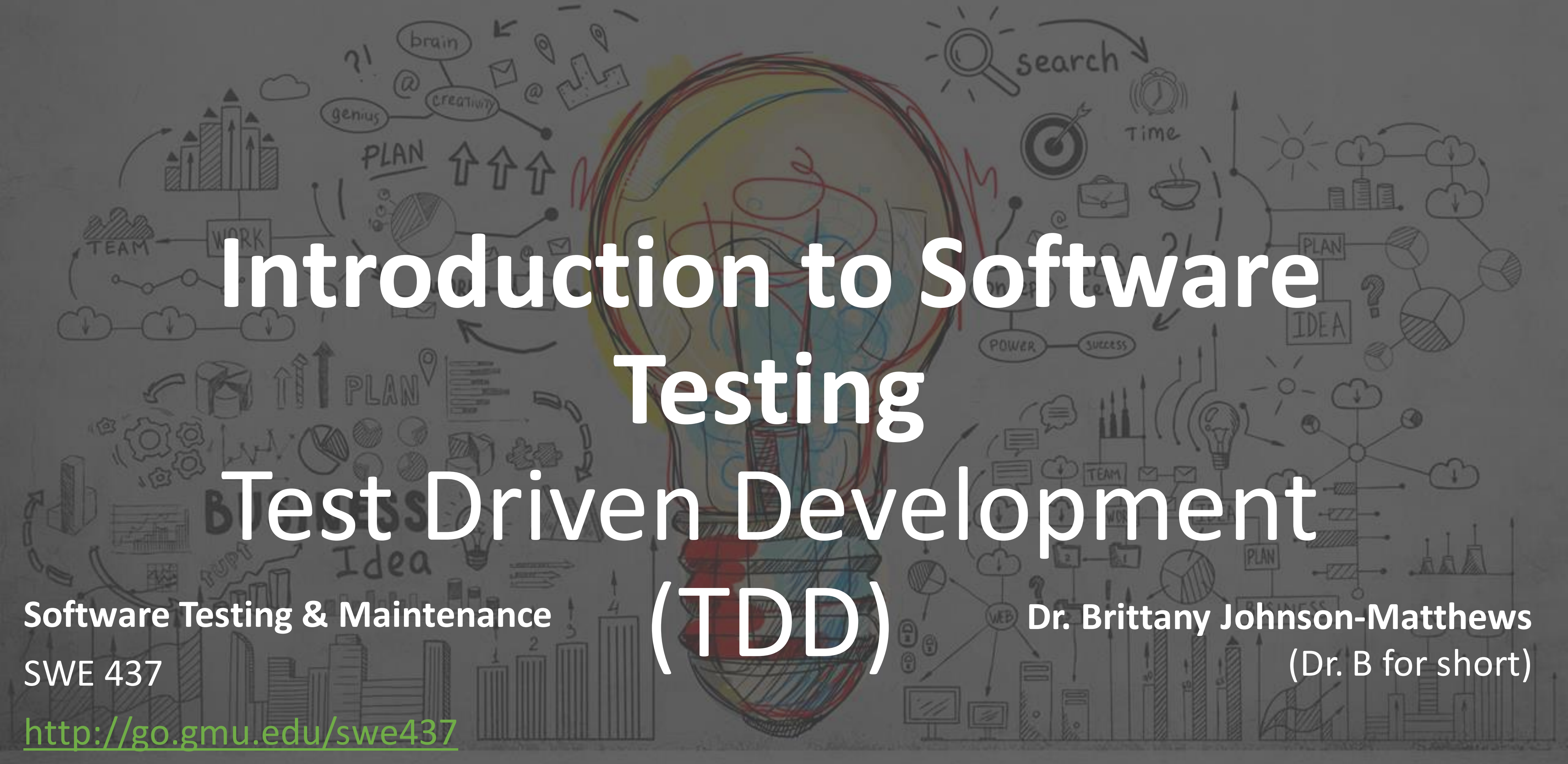
Software Testing & Maintenance

SWE 437

http://go.gmu.edu/swe437

Dr. Brittany Johnson-Matthews

(Dr. B for short)



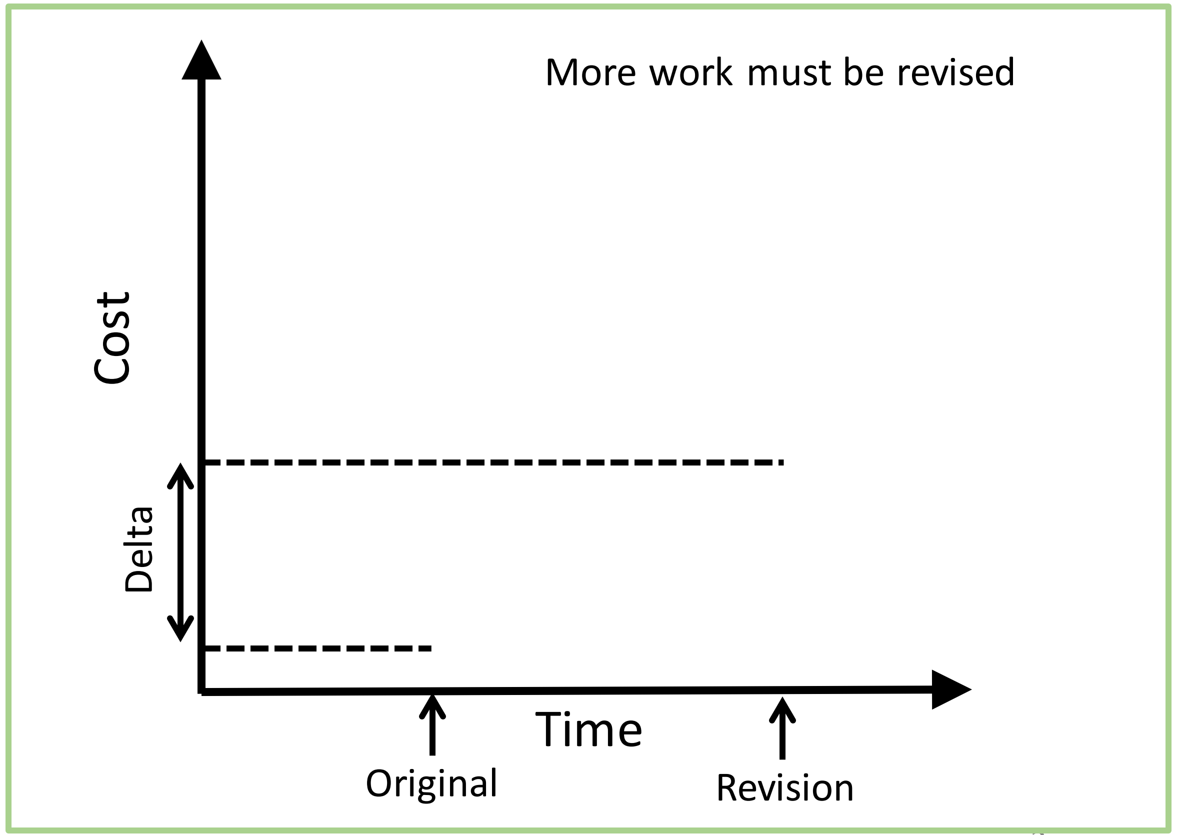
Growing importance of testing

Philosophy of **traditional** software development methods

-**Upfront** analysis

-Extensive **modeling**

-Reveal **problems** as early as possible



Scaling assumptions

Traditional assumptions are…

1. *Modeling and analysis can identify potential problems early in development*
2. *Savings implied by the cost-of-change curve justify the cost of modeling and analysis over the life of the project*

These are true if requirements are always complete and current

But customers always change their minds!

-Humans are naturally good at approximating

-But pretty bad at perfecting

These two assumptions have made software engineering frustrating and difficult fo decades

Thus, agile methods…

Why be agile?

Agile methods start by recognizing that **neither assumption** is valid for many current software projects

- -Software engineers are **not good at developing requirements**
- -We do not anticipate many **changes**
- -Many of the changes we do anticipate are **not needed**

Requirements (and other "non-executable artifacts") tend to go **out of date** very quickly

- -We seldom take time to **update** them
- -Many current software projects **change continuously**

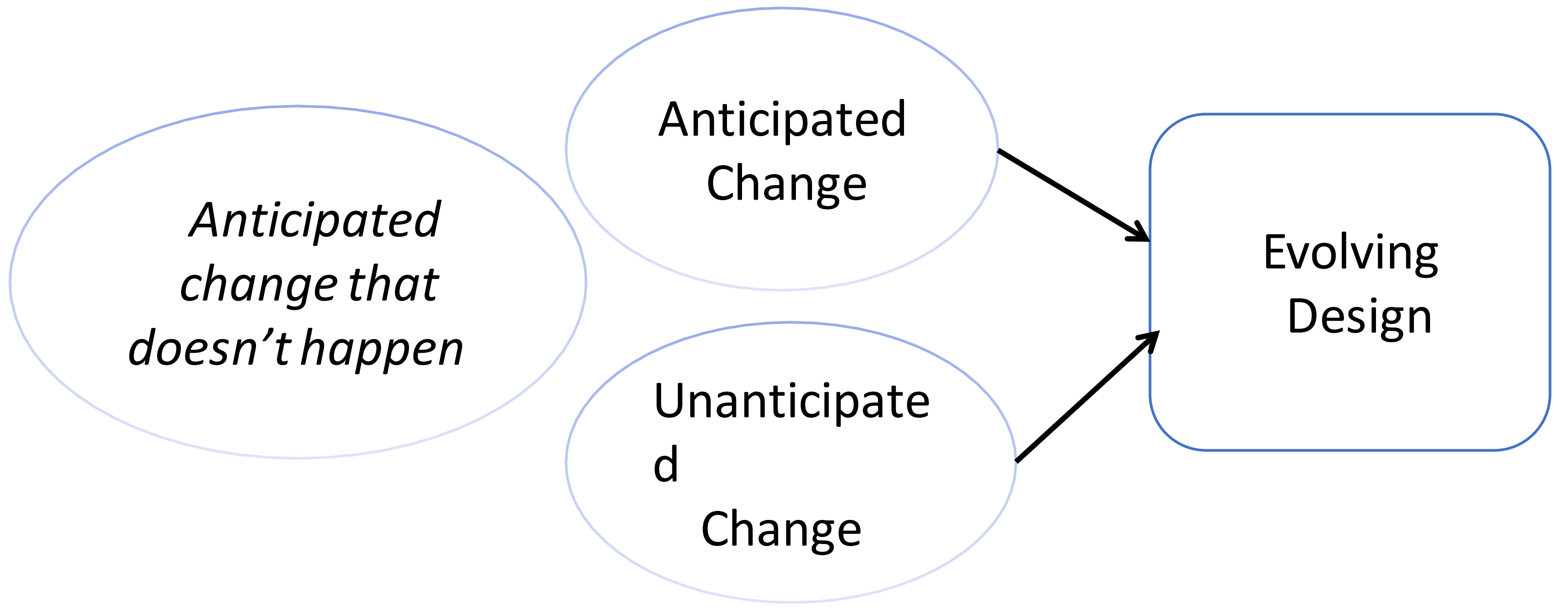
Agile methods expect software to **start small and evolve** over time

- -Embraces **software evolution** instead of fighting it

Supporting evolutionary design

Traditional design advice says to anticipate changes

Designers often anticipate changes that don’t happen

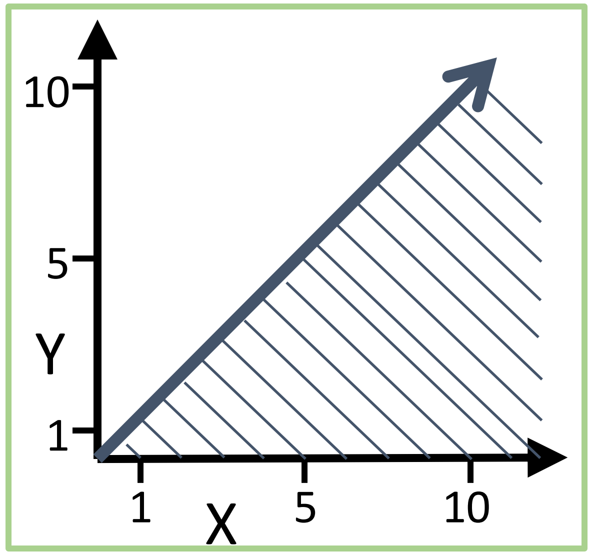


Both anticipated and unanticipated changes affect design

The test harness as guardian (4.2)

What is correctness?

Traditional Correctness
(Universal)
$\forall x,y,\ x \geq y$



Agile Correctness
(Existential)

$\{\ (1, 1) \rightarrow T$
$(1, 0) \rightarrow T$
$(0, 1) \rightarrow F$
$(10, 5) \rightarrow T$
$(10, 12) \rightarrow F\ \}$

Supporting evolutionary design

In **traditional** methods, we try to define **all correct behavior** completely, at the beginning

-What is **correctness**?

-Does "correctness" **mean anything** in large engineering products?

-People are **VERY BAD** at completely defining correctness

In **agile** methods, we redefine correctness to be **relative** to a specific set of tests

-If the software behaves correctly **on the tests**, it is "correct"

-Instead of **defining all** behaviors, we **demonstrate some** behaviors

-**Mathematicians** may be disappointed at lack of completeness

But software engineers ≠ mathematicians!

In-class Exercise

Discuss

limited correctness

Do you understand the distinction?

How does limited correctness related to evolutionary design?

Verifying ”correctness”

A **test harness** runs all automated tests and reports results to the developer

Tests must be **automated**

-Test automation is a **prerequisite** to test driven development

Every test must include a **test oracle** that can evaluate whether that test executed correctly

The tests replace the **requirements**

Tests must be **high quality** and must **run quickly**

We run tests **every time** we make a change to the software

Continuous integration

Agile methods work best when the current version of the software can be run against all tests at any time

A ***continuous integration server*** rebuilds the system, returns, and re-verifies tests whenever *any* update is checked into the repository

Mistakes are caught earlier

Other developers are aware of changes early

The rebuild and reverify must happen as soon as possib

-Thus, tests need to execute quickly

A ***continuous integration server*** doesn't just run tests,

it decides if a modified system is **still correct**.

Continuous integration reduces risk

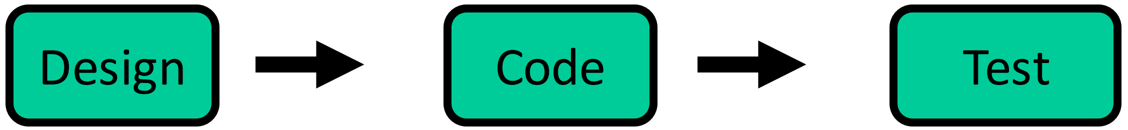
TDD encourages incremental integration of functionality

Non-integrated functionality is dangerous

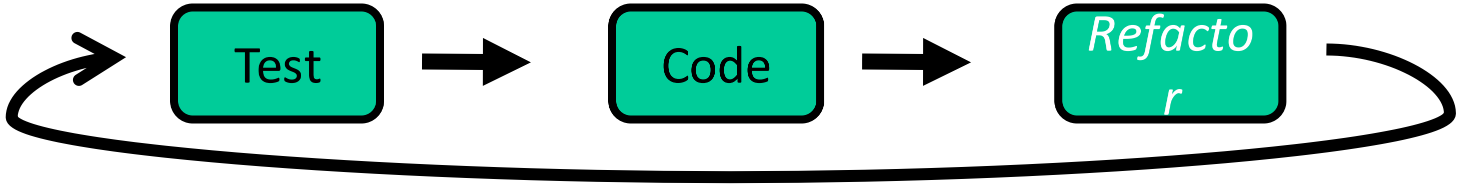
Build it right: TDD

The heart-beat of TDD = Test-Code-Refactor

- -The rule: **only write code to fix a failing test**
- -Traditional development cycle



- -Test-driven development cycle



Sometimes called red-green-refactor

Build it right: TDD

First, we write a test

This really amounts to design by example

- -We make decisions about how the **Application Programmer Interface (API)** works
    - -Class name, method names, return results, etc.
    - -This is essentially the user interface
- -We're thinking hard about how code is used
- -We're taking a client perspective
- -We're working at a very small scale

Example for a stack

```
stack = … ;
stack.push (x);
y = stack.pop();
assertEquals (x, y);
```

Start with one concrete client interaction

In-class Exercise

You are asked to write a program to **merge two lists**

Design the **FIRST test case** (test values and expected output)

Do NOT consider software design, or details of the behavior that are not needed for the first test

Build it right: TDD

Then we write **just enough** code

- -We don’t write more code
- -All we want is to make the test pass
  - -It should be a very small step
  - -Implementation probably not optimal
  - -We don’t care (yet)

Goal: Make code base (just) pass test suite

Build it right: TDD

And then we refactor

TDD without refactoring just makes ugly code

- -Maintenance debt

We have **numerous transformations** to address this

Developing with **small steps**

- -The code always runs!
    - -Changes are small enough to fit in our heads
    - -Time-frame is minutes to (maybe) hours
- -**Evolutionary design**
    - -Anticipated vs unanticipated changes
    - -Many "anticipated changes" turn out to be unnecessary

Build it right: TDD

Keeping code healthy with refactoring

Refactoring: *A disciplined technique for restructuring an existing body of code, and altering its internal structure without changing its external behavior*

- -Refactoring is disciplined
  - -Wait for a problem before solving it
- -Refactorings are transformations
  - -Many refactorings are simply applications of patterns
- -Refactorings alter internal structure
- -Refactorings preserve behavior

Focus is on current code, not future code.

User stories

A **user story** is a few sentences that capture what a user will do with the software

Withdraw money from checking account

Support technician sees customer's history on demand

Agent sees a list of today's interview applicants

-In the language of the **end user**

-Usually small in scale with **few details**

-**Not** archived

In-class Exercise

In assignment 3, you added new functionality.

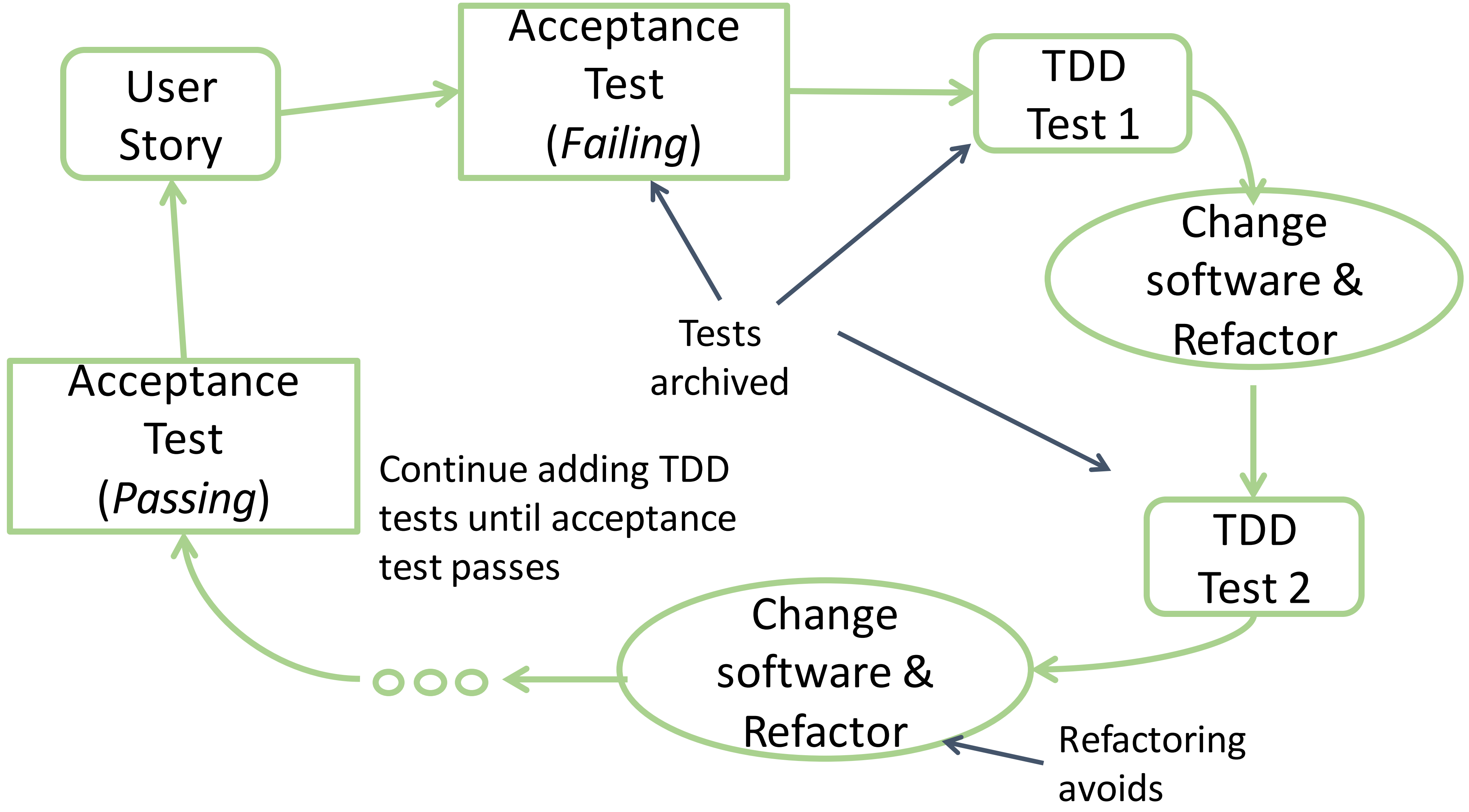
Each individual in your group:

write a user story that would start the need for that functionality

Share the user stories in your group and critique them

- Are they the right size?
- Are they in the user’s vocabulary?

Acceptance tests with agile


User Story
Acceptance Test (*Failing*)
TDD Test 1
Change software & Refactor
Tests archived
TDD Test 2
Change software & Refactor
Acceptance Test (*Passing*)
Continue adding TDD tests until acceptance test passes
Refactoring avoids

The caveat

Do **TDD tests** (acceptance or otherwise) test the software well?

- -Do the tests achieve good **coverage** on the code?
- -Do the tests find most of the **faults**?
- -If software passes, should management feel confident the software is **reliable**?

NO!

Why not?

Most agile tests focus on “happy paths”

- What should happen under normal use

They often miss things like

- **Confused**-user paths
- **Creative**-user paths
- **Malicious**-user paths

The agile methods literature does not give much guidance

Summary – take small steps

More companies are putting **testing first**

This can dramatically **decrease cost** and **increase quality**

A different view of **"correctness"**

-Restricted but practical

Embraces **evolutionary design**

TDD is definitely **not** test automation

-Test automation is a prerequisite to TDD

TDD tests aren't enough