COMP 5700/6700/6706
Software Process

Spring 2016

David Umphress

Construction

- Lesson: Construction
- Strategic Outcome:
  - To understand the code construction process
- Tactical Outcomes:
  - To know the rationale of the design-test-code philosophy
  - To understand the TDD approach
  - To be able to apply TDD to a sample problem
- Readings
  - "Test-Driven Development"
    - http://en.wikipedia.org/wiki/Test-driven_development
  - "unittest – Unit Testing Framework"
    - http://docs.python.org/library/unittest.html
- Instant take-aways:
  - TDD
- Bookshelf items
  - Koskela, Lasse. 2008. Test Driven: Practical TDD and Acceptance TDD for Java Developers. Manning Press.

Syllabus

- Software engineering raison d'être
- Process foundations
- Common process elements
- Construction
- Reviews
- Refactoring
- Analysis
- Architecture
- Estimation
- Scheduling
- Integration
- Repatterning
- Measurements
- Process redux
- Process descriptions*
- Infrastructure*
- Retrospective

- **Construction overview**
- **TDD**
  - **rationale**
  - **process**
  - **demo**

COMP5700/6700/6706 Goal Process

Minimal Guiding Indicators

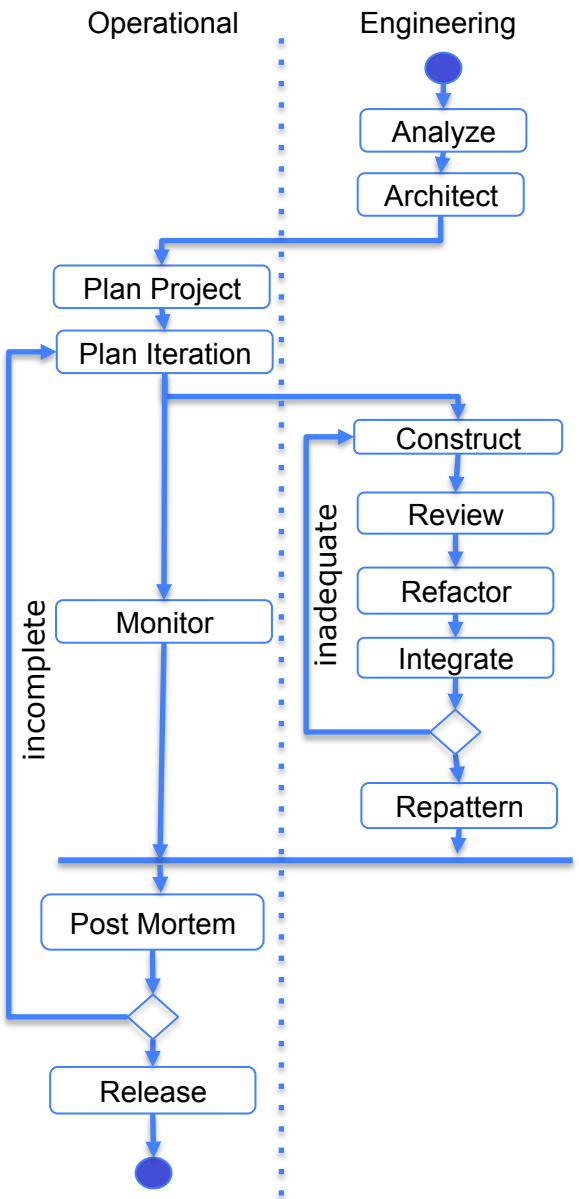
Goal	Indicator
Cost:	Don't care
Schedule:	PV/EV >.75
Performance:	
Product:	
NFR:	none
FR:	100% BVA
Process:	pain < value

Minimal Sufficient Activities

Engineering Activities
- Envision
  - Analyze
- Synthesize
  - Architect
- Articulate
  - Construct
  - Refactor
- Interpret
  - Review
  - Integrate
  - Repattern

Operational Activities
- Plan
  - Plan project
  - Plan iteration
- Monitor
- Release

Minimal Viable Process



Minimal Effective Practice

MSA	MEP
Analyze	Scenarios
Architect	CRC
Plan Project	Component-based estimation
Plan Iteration	Component-iteration map
Construct	TDD
Review	Review checklist Test code coverage
Refactor	Ad hoc sniffing
Integrate	Ad hoc
Repattern	Ad hoc
Monitor	Time log Change log Burndown
Post Mortem	PV/EV
Release	git zip spreadsheets

Until Now

Minimal Guiding Indicators

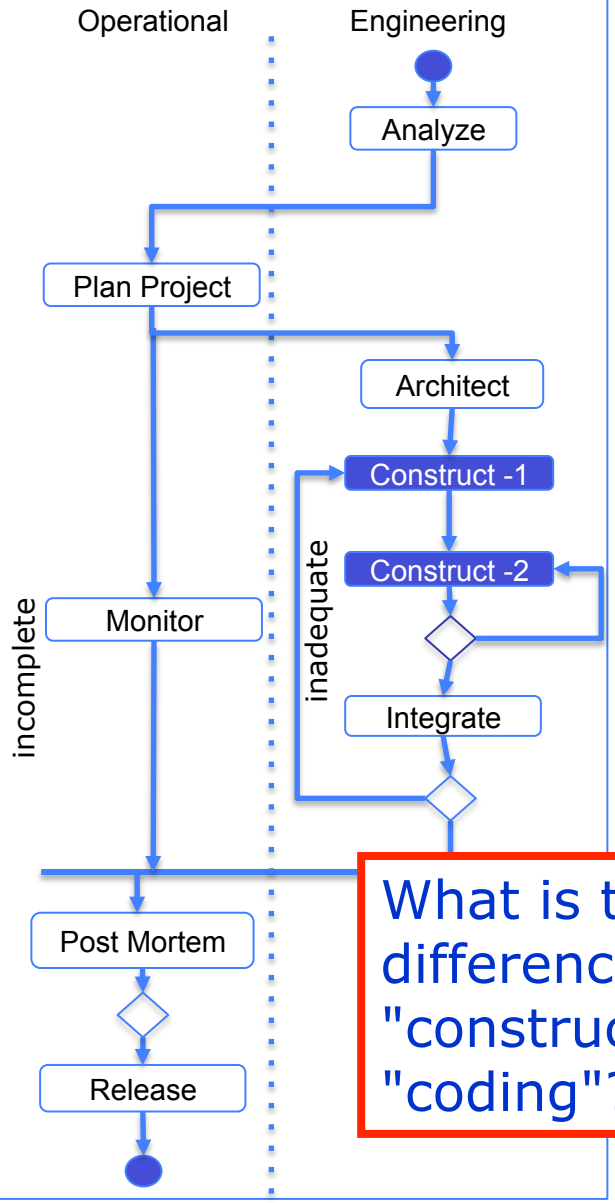
Goal	Indicator
Cost:	None
Schedule:	PV/EV >.75
Performance:	
Product:	
NFR:	none
FR:	100% BVA
Process:	pain < value

Minimal Sufficient Activities

Engineering Activities
- Envision
  - Analyze
- Synthesize
  - Architect
- Articulate
- Interpret

Operational Activities
- Plan
  - Plan project
  - Plan iteration
- Monitor
- Release

Minimal Viable Process



Minimal Effective Practice

MSA	MEP
Analyze	• Write acceptance tests .
Plan Project	• Guess projected LOC. • Guess projected effort.
Architecture	• Identify components. • Write tests which, when passed, designate component completion.
Construct-1	• Select one component. • Write production code sufficient to pass unit tests for that component. • Commit the code to git.
Construct-2	• Run unit tests.
Integration Test	• Run acceptance tests.
Monitor/Cntrl	• Record activities in Time Log. • Record defects/changes in …nge Log.
	…nt and record LOC.
	…aded zipped Eclipse …ct.
	…ad spreadsheet.

What is the difference between "construction" and "coding"?

Until Now

Minimal Guiding Indicators

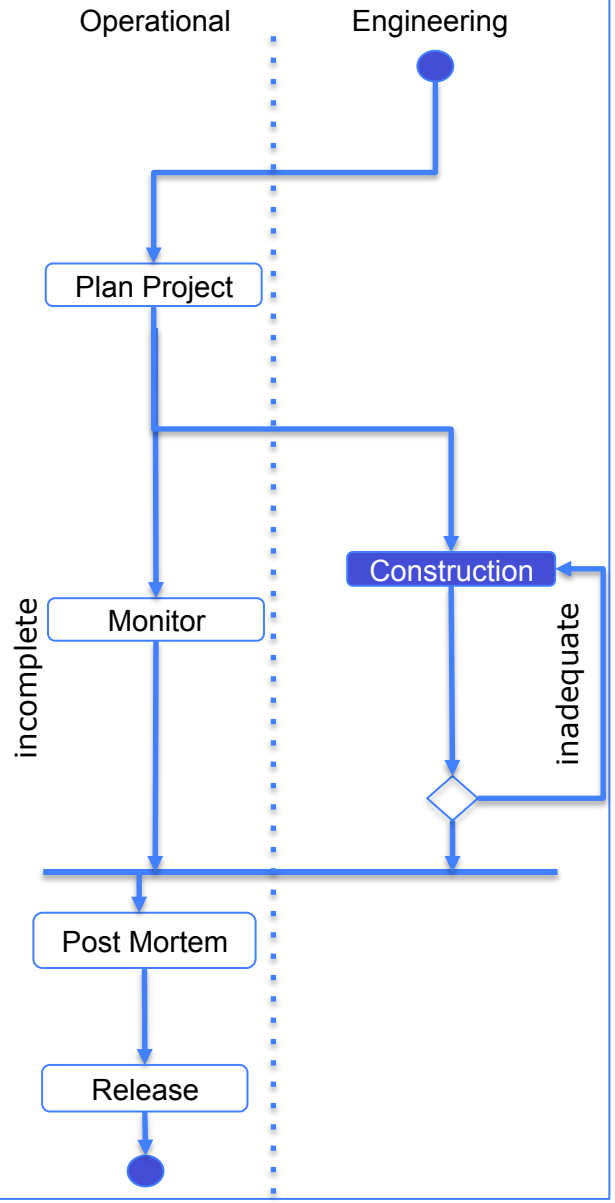
Goal	Indicator
Cost:	None
Schedule:	On time
Performance:	
Product:	
NFR:	Delivery rqt
FR:	BVA
Process:	Evident

Minimal Sufficient Activities

Engineering Activities
- Envision
  - Analyze
- Synthesize
  - Architect
- Articulate
- Interpret

Operational Activities
- Plan
  - Plan project
  - Plan iteration
- Monitor
- Release

Minimal Viable Process

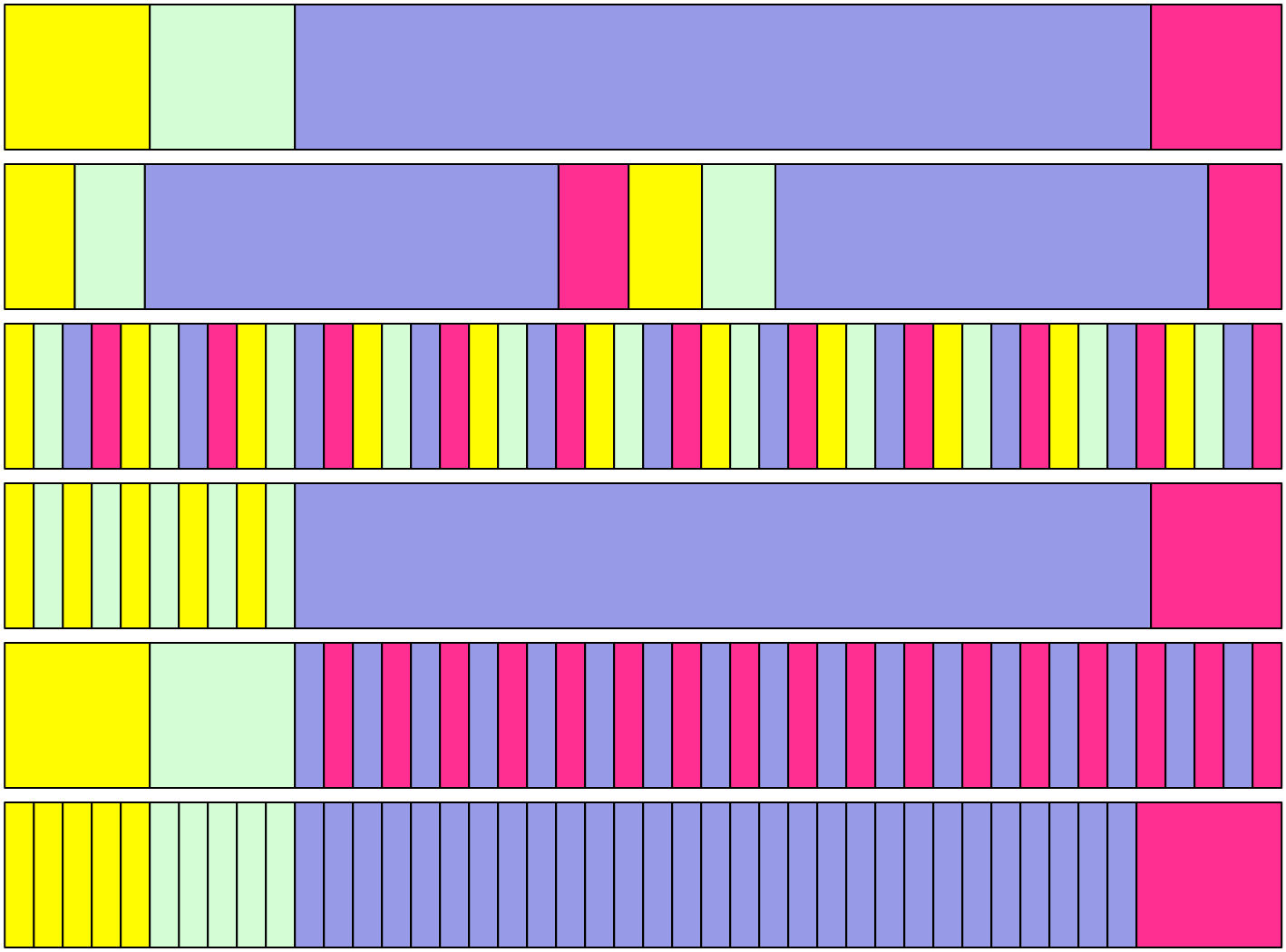


Minimal Effective Practice

MSA	MEP
Plan Project	• Guess projected LOC. • Guess projected effort.
Construction	• Write production code
Monitor	• Record activities in Time Log. • Record defects/changes in Change Log.
Post Mortem	• Commit to git • Count and record LOC.
Release	• Uploaded zipped Eclipse project. • Upload spreadsheet.

What is the difference between "construction" and "coding"?

Construction



Construction …

- … is the activity of building code given a high-level design
- … includes
  - low-level design (aka "how-design" vs "what-design")
  - coding
  - unit testing

Construction

- Traditional approach
  - construction = design* -> code -> test**
  - problems:
    - tests are often written based on non-code artifacts, which may work for black-box tests, but not necessarily for white-box tests
    - tests may be written by non-coders who lack white-box knowledge
    - test is done after code is written
      - no one wants to write code, then write tests … test burnout
      - writing module after module before writing tests creates error compounding effect

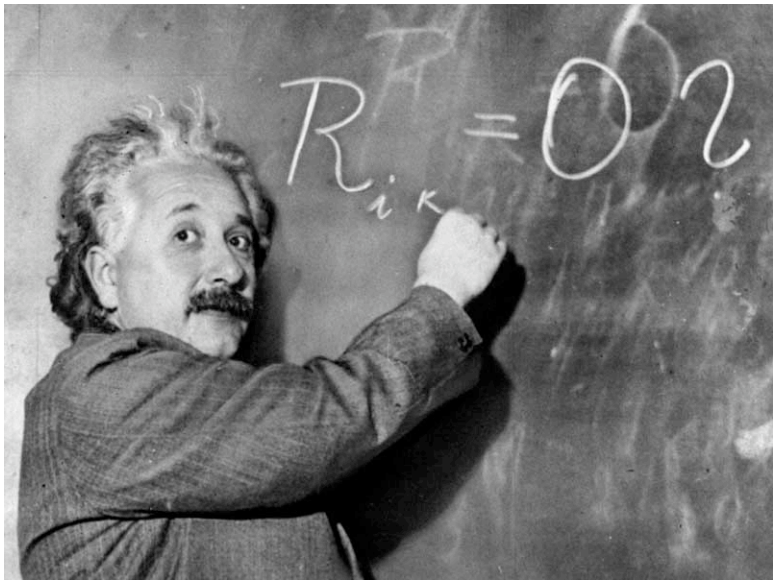
* low-level design
** unit test

Construction

- Contemporary approach
  - construction = design -> test -> code
    - test-driven development
      - a systematic approach to programming where the tests determine what code to write
        - » concept: the tests drive the design
      - also referred to as "programming by intent"
        - » we make our intent clear (via a test case) before we start to write code
  - advantages
    - all delivered code is accompanied by tests
    - no code goes into production untested
    - writing tests first yield a better understanding of what the code is to do
  - challenges
    - overcoming the "gotta code" urge
    - appearance of lots of work

TDD Philosophy

- The most powerful force in the universe is compound interest
  - Albert Einstein



- TDD outlook:
  - don't go into debt
  - make regular payments
  - pay down the principle

Are you ready?

TDD

- Test-Driven Development is administered by oath in many companies.
- Some countries have thought about requiring a license to use it.
- Be prepared to be hunted for your TDD skills.

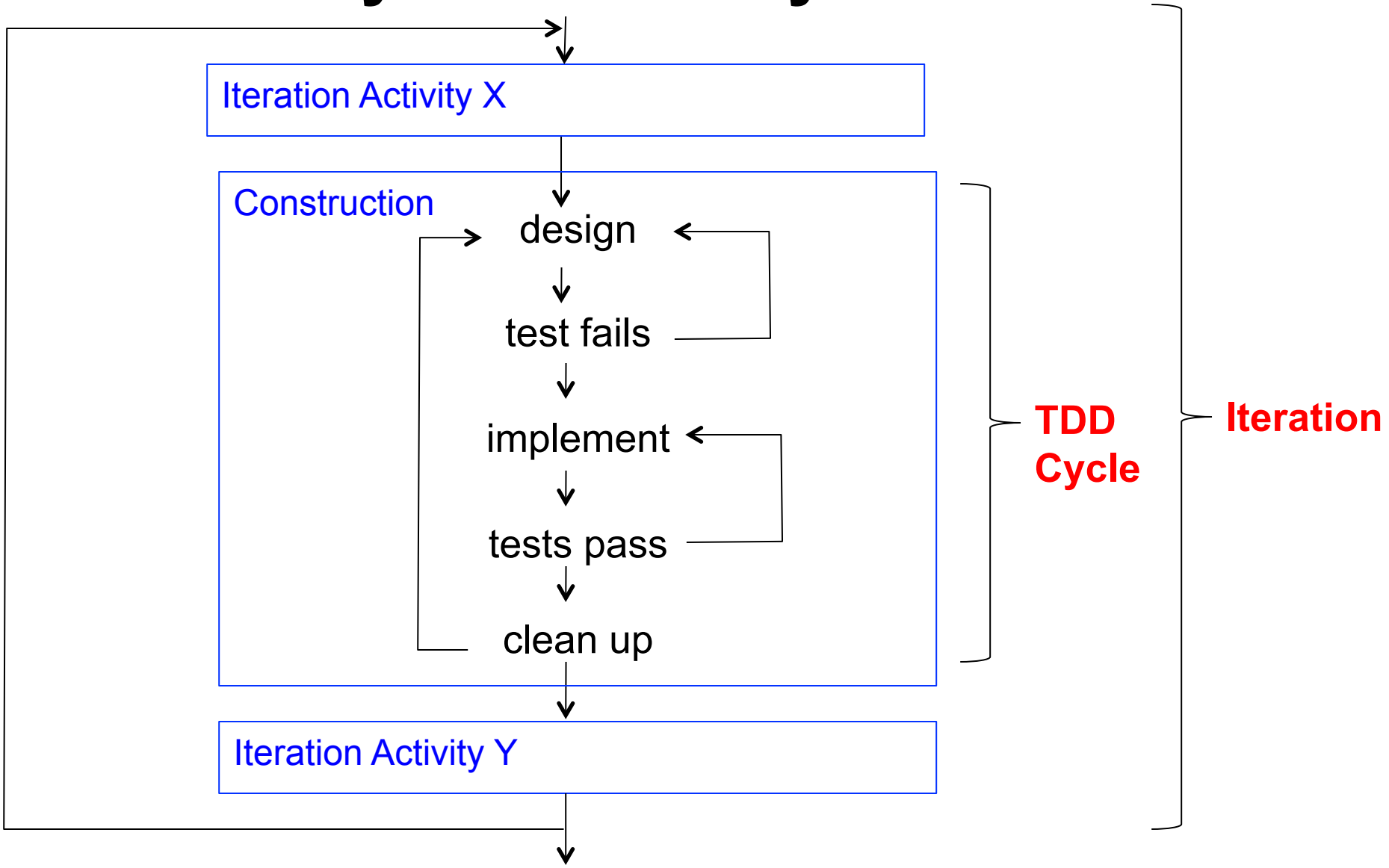
Are you sure you are ready for the secret?

Here it is

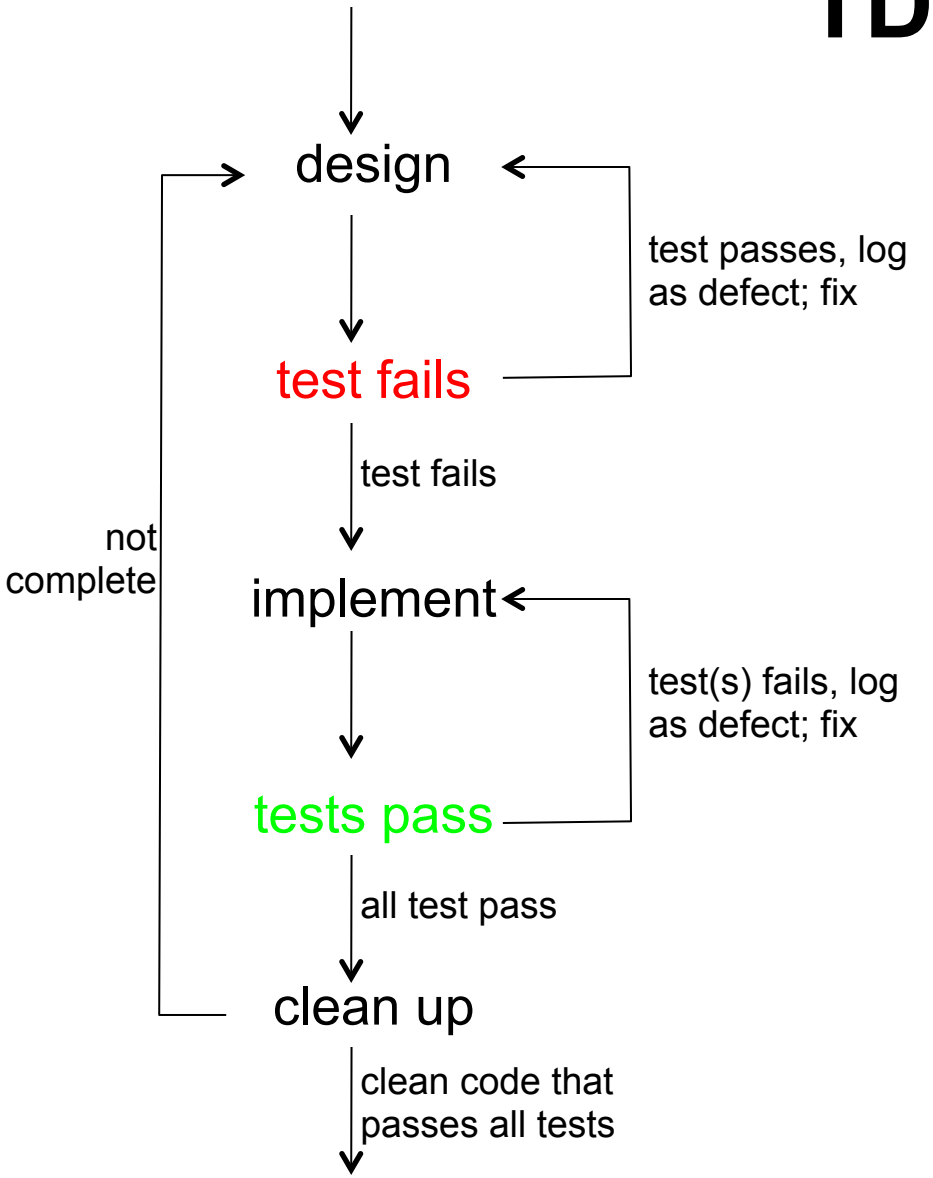
Last chance …

TDD = only write code to fix a failing test case

TDD Cycle vs Lifecycle Iteration



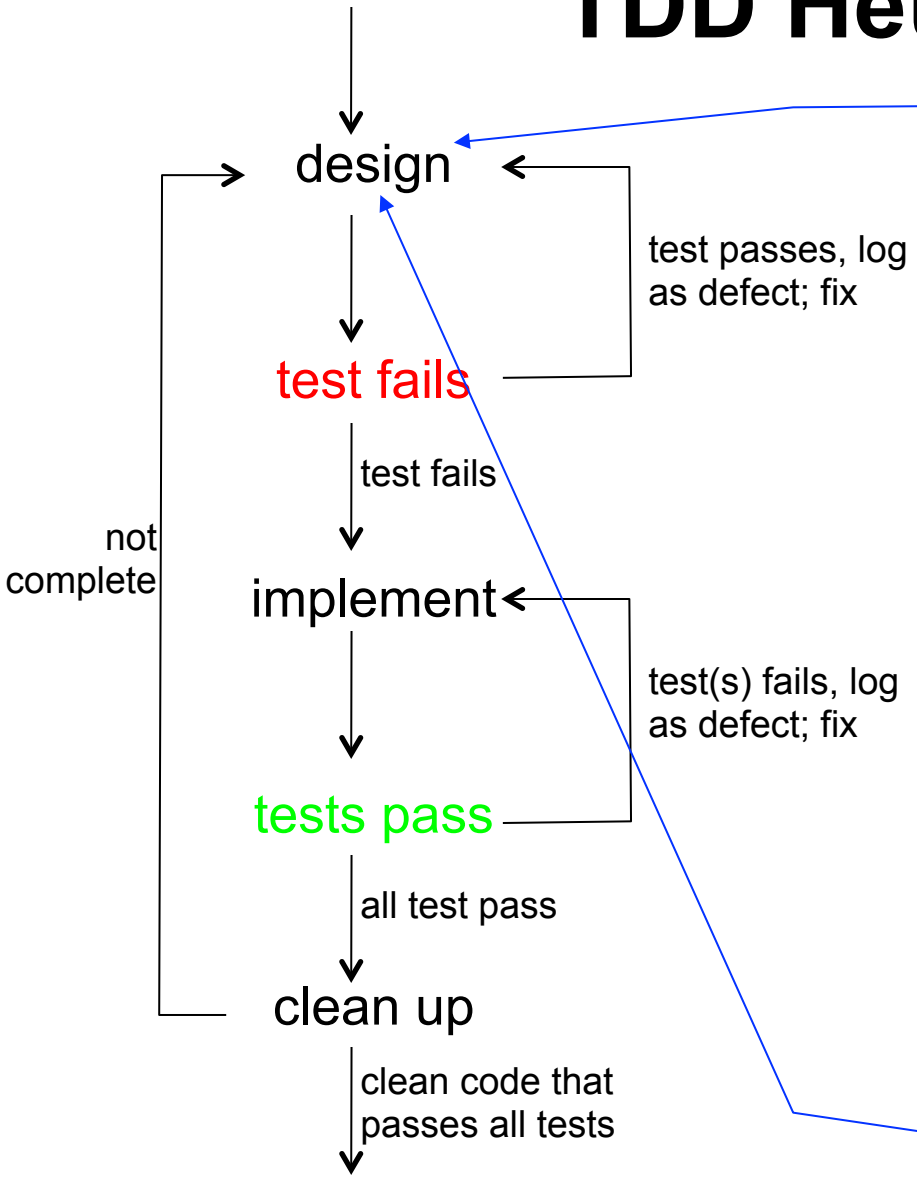
TDD



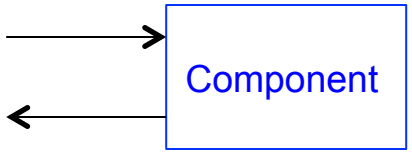
- Process
  - Entry criteria
    - idea of "what" component is to do
    - "acceptance" test cases*
  - Tasks
    - select/write a test to expose the design
      - it should fail
    - implement the business logic with as little code as possible
    - test again
      - it should pass
    - clean up the code and retest
    - repeat until selected acceptance tests pass
  - Verification
    - all tests must pass before component construction is considered complete
  - Exit criteria
    - production code that has been tested

* The acceptance tests depend on the scope of the iteration

TDD Heuristics



This means to formulate a test that exposes an unimplemented portion of design.

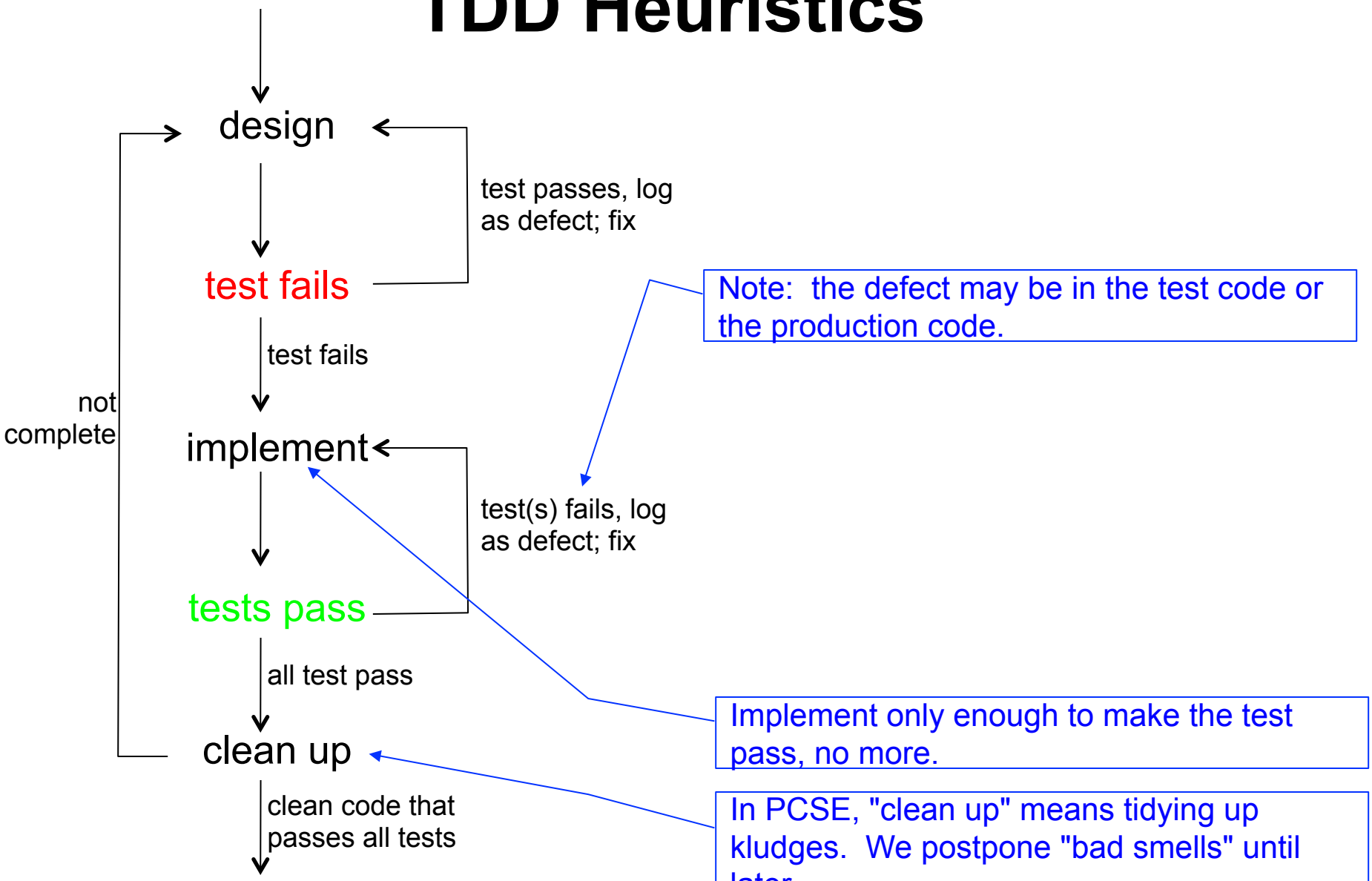


The level of risk you are willing to assume determines the extent of the design you expose (meaning, the amount of functionality exercised by your test):

- test a small amount of functionality if the corresponding production code contains constructs that you commonly have problems with
- test a larger amount of functionality if you have confidence in the production code (key: be honest with yourself).

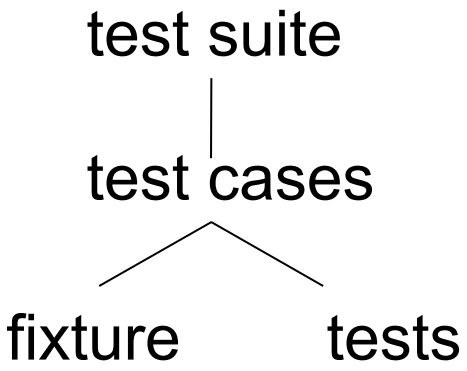
Restrict your test to one assert

TDD Heuristics



TDD

- History
  - TDD originated with XP
  - Motivated creation of testing framework: sUnit
  - Popularity spawned generic xUnit
    - instantiated as jUnit, cppUnit, vbUnit, rUnit, etc



test suite
- test case$_1$
  - setup
  - assertion$_1$
  - …
  - assertion $_n$
  - teardown
- test case$_n$

TDD with Python

- unittest (aka PyUnit)
  - built-in Python to support test automation
  - http://docs.python.org/library/unittest.html:
    - test fixture
      - "A *test fixture* represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process. "
    - test case
      - "A *test case* is the smallest unit of testing. It checks for a specific response to a particular set of inputs. unittest provides a base class, TestCase, which may be used to create new test cases. "
    - test suite
      - "A *test suite* is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together. "
    - test runner
      - "A *test runner* is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests. "

First: a word about unittest

- Pattern:

```
import thingToBeTested
import unittest

class TC001(unittest.TestCase):

  def setup(self)
    # set up test initialization here (if needed)

  def teardown(self)
    # reset environment to "before test" state (if appropriate)

  def test_001(self)
    #  write assertion that tests a facet of the component
    #  under consideration
    #  write it in such a way that "pass" indicates the facet has
    #  been tested successfully

  def test_002(self)
    # write another assertion

unittest.main()
```

Notes:

- If a test assertion fails, the test is id'ed as "F"
- All other exceptions are flagged as "E" (and the test case is considered a failure)
- setup() is executed before each test
  - results in "E" for all tests if it is unable to complete
- teardown() is executed after each test
  - is executed regardless of outcome of tests
- Other methods whose names start with "test" are executed [in sorted order]

unittest example

Assert methods in unittest.TestCase

Most *assert* methods accept an optional *msg* argument, which is used as an explanation for the error.

`assert_(expr[ , msg)` `assertTrue(expr[ , msg] )`	Fail if *expr* is False
`assertFalse(expr[ , msg] )`	Fail if *expr* is True
`assertEqual(first, second[ , msg] )`	Fail if *first* is not equal to *second*
`assertNotEqual(first, second[ , msg] )`	Fail if *first* is equal to *second*
`assertAlmostEqual(first, second [ , places[ , msg]] )`	Fail if *first* is equal to *second* up to the decimal place indicated by *places* (default: 7)
`assertNotAlmostEqual(first, second [ , places[ , msg]] )`	Fail if *first* is not equal to *second* up to the decimal place indicated by *places* (default: 7)
`assertRaises(exception, callable, ...)`	Fail if the function *callable* does not raise an exception of class *exception*. If additional positional or keyword arguments are given, they are passed to *callable*.
`fail([ msg] )`	Always fail

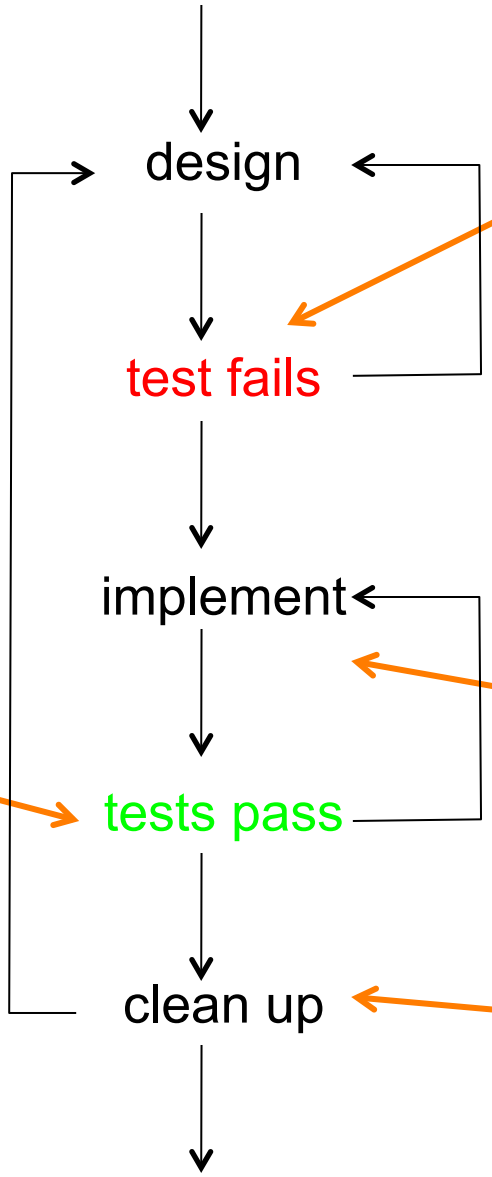
http://docs.python.org/library/unittest.html

TDD Demo

Let's work it through in real time

About Defects

Each time through this loop will be called a TDD cycle. It is not a product iteration



This test is supposed to fail. If it doesn't, it should be recorded as a defect in either the 1) test code or 2) the production code (or maybe both).

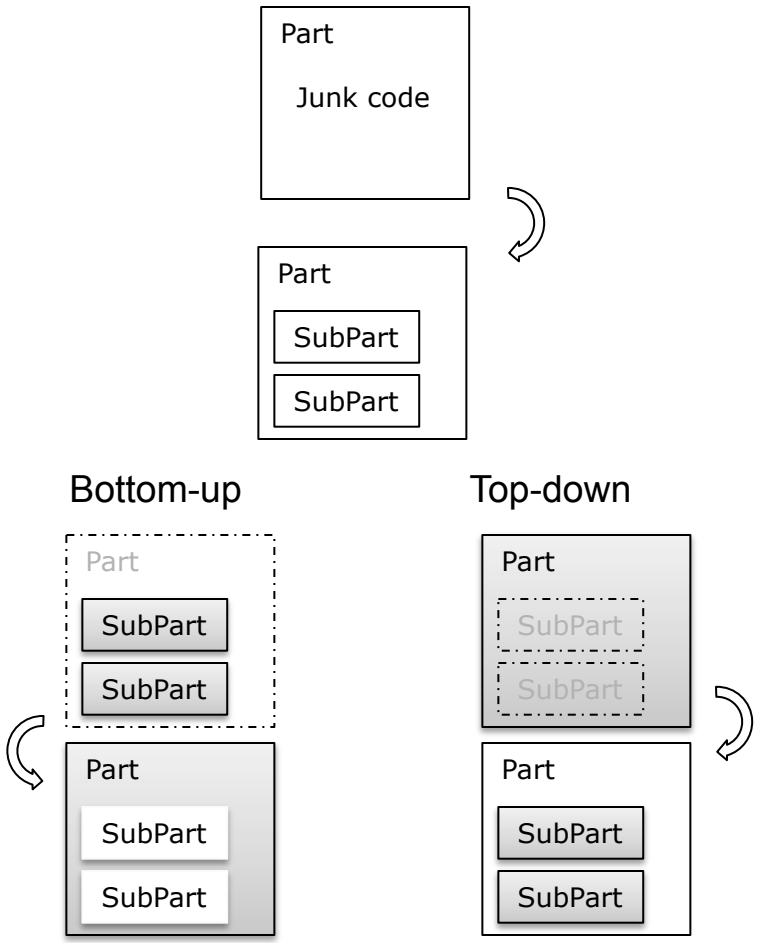
This test is supposed to pass. If it fails, it should be diagnosed and recorded as a defect in either the 1) test code or 2) the production code (or maybe both).

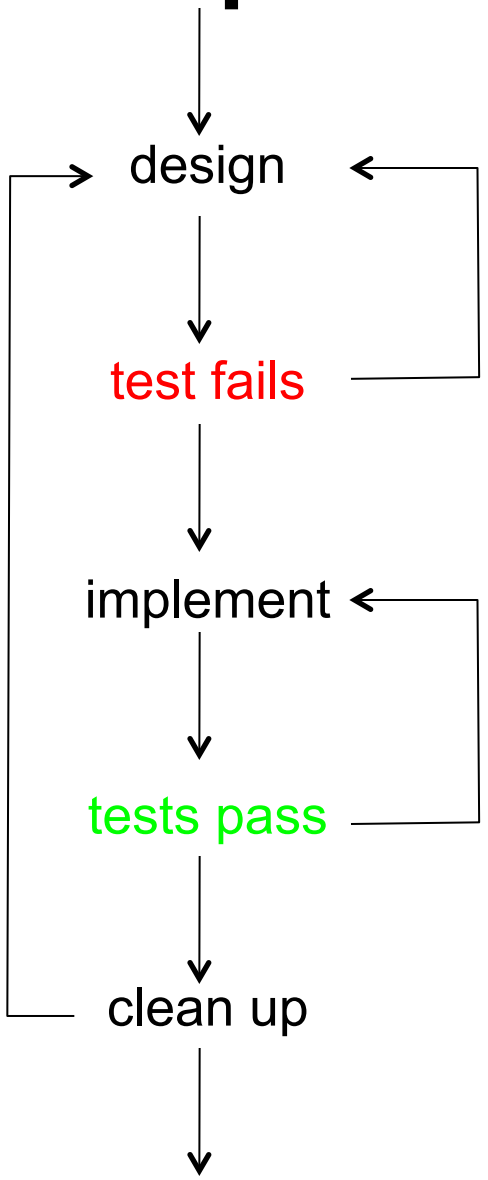
Don't worry about recording defects here. You aren't executing the code yet.

Don't worry about recording defects here either, unless you discover a problem injected previous to clean up

About Cleanup

Cleanup may (should?) result in identifying parts.





About TDD Cycles

I get paid for code that works, not for tests, so my philosophy is to test as little as possible to reach a given level of confidence … If I don't typically make a kind of mistake (like setting the wrong variables in a constructor), I don't test for it. I do tend to make [boundary testing] errors, so I'm extra careful when I have logic with complicated conditionals. When coding on a team, I modify my strategy to carefully test code that we, collectively, tend to get wrong.

Kent Beck

How much production code should you write at each TDD cycle?

http://en.wikipedia.org/wiki/Transformation_Priority_Premise

About TDD Cycles (con't)

This slide is blank on purpose. It will be revealed to you later.

About TDD Cycles (con't)

This slide is blank on purpose. It will be revealed to you later.

Keeping TDD from being TDDious

- For every class
  - for i in constructor..methods:
    - develop interface (stubbed body)
      - parms
        - » present?
        - » number?
          - ✦ for each parm:
            - default value?
            - range?
      - attributes
        - » present?
          - ✦ class vs instance level
          - ✦ type
          - ✦ range
          - ✦ visibility
          - ✦ initial value
    - develop body
      - return value
        - » present?
          - ✦ type
          - ✦ range
      - exceptions?
        - » present?
          - ✦ type
          - ✦ precondition
          - ✦ message
      - side effects?

Summary

Topics

- Construction Overview
- TDD
  - rationale
  - process
  - demo

Key Points

- Traditional construction = "coding" = design -> code -> test
  - This results in compounding bugs
- Contemporary construction = design -> test -> code -> etc.
- Contemporary contemporary construction = design -> test -> code -> test -> cleanup -> etc.
  - a.k.a. TDD
  - write tests incrementally and concurrently
  - result: tested production code and regression tests