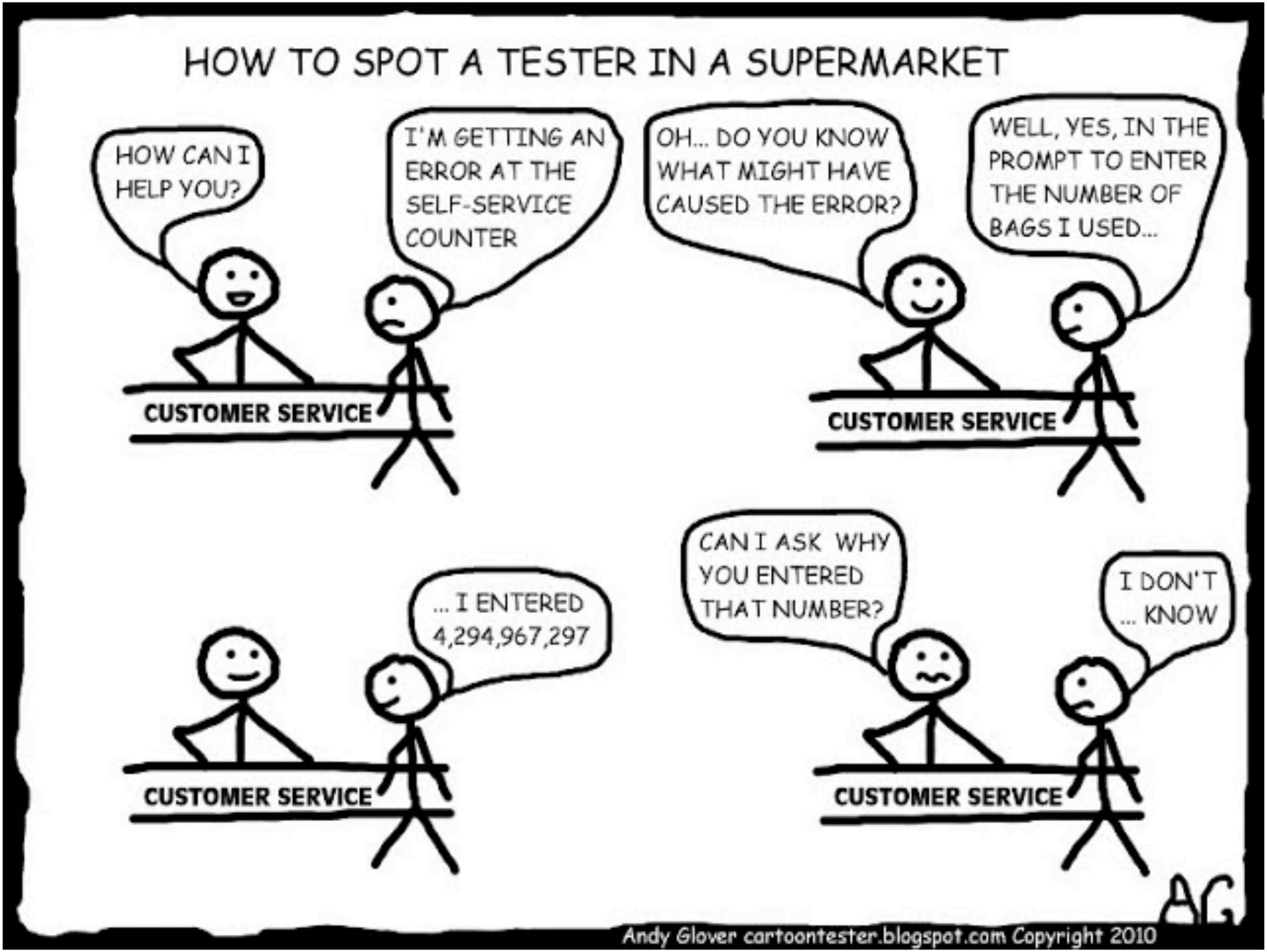

HOW TO SPOT A TESTER IN A SUPERMARKET
HOW CAN I HELP YOU?
I'M GETTING AN ERROR AT THE SELF-SERVICE COUNTER
CUSTOMER SERVICE
OH... DO YOU KNOW WHAT MIGHT HAVE CAUSED THE ERROR?
WELL, YES, IN THE PROMPT TO ENTER THE NUMBER OF BAGS I USED...
CUSTOMER SERVICE
... I ENTERED 4,294,967,297
CUSTOMER SERVICE
CAN I ASK WHY YOU ENTERED THAT NUMBER?
I DON'T ... KNOW
CUSTOMER SERVICE
AG
Andy Glover cartoontester.blogspot.com Copyright 2010

CPEN 321

Testing

Agenda (this and next class)

- Follow-up on mid-term survey
- Into to testing
  - Types of testing
  - When to stop testing
  - …
- Expectations for the Testing Milestone (M5, November 5)

Agenda

- **Follow-up on mid-term survey**
- Into to testing
  - Types of testing
  - When to stop testing
  - …
- Expectations for the Testing Milestone (M5, November 5)

What are you learning in this course?

- Software Engineering != Programming

- CPEN 321 course description:
  Engineering practices for the development of non-trivial software-intensive systems including requirements specification, software architecture, implementation, verification, and maintenance. Iterative development. Recognized standards, guidelines, and models.

Topics Covered in the Course

- Processes: what, how, for which purpose
- Requirements: plan, make explicit, how to capture
- Design principle and patterns
  - High cohesion, low coupling; high fan in, low fan out, …
  - REST and Microservices
- Code Reviews: when, how, what to look for
- Validation and Verification, Testing
- CI and DevOps
- Containers for development and production
- Teamwork, Version Control, Release Management
- …

Engineering

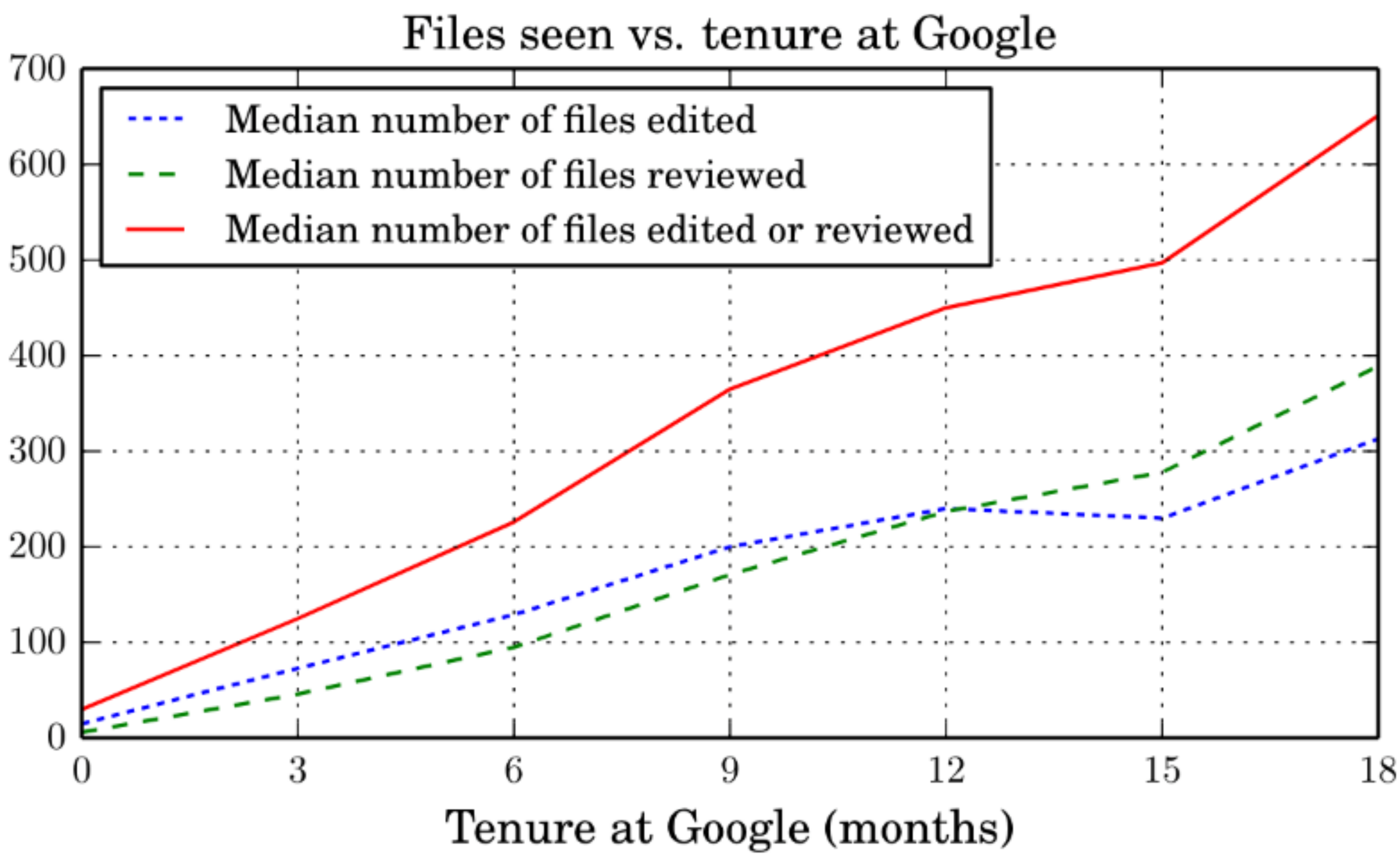
If multiple solutions exist, engineers **weigh each design choice** based on their merit **and choose the solution that best matches the requirements**.

The crucial and unique task of the engineer is to identify, understand, and interpret the constraints on a design in order to yield a successful result. **It is generally insufficient to build a technically successful product**, rather, it must also meet further requirements.

Project

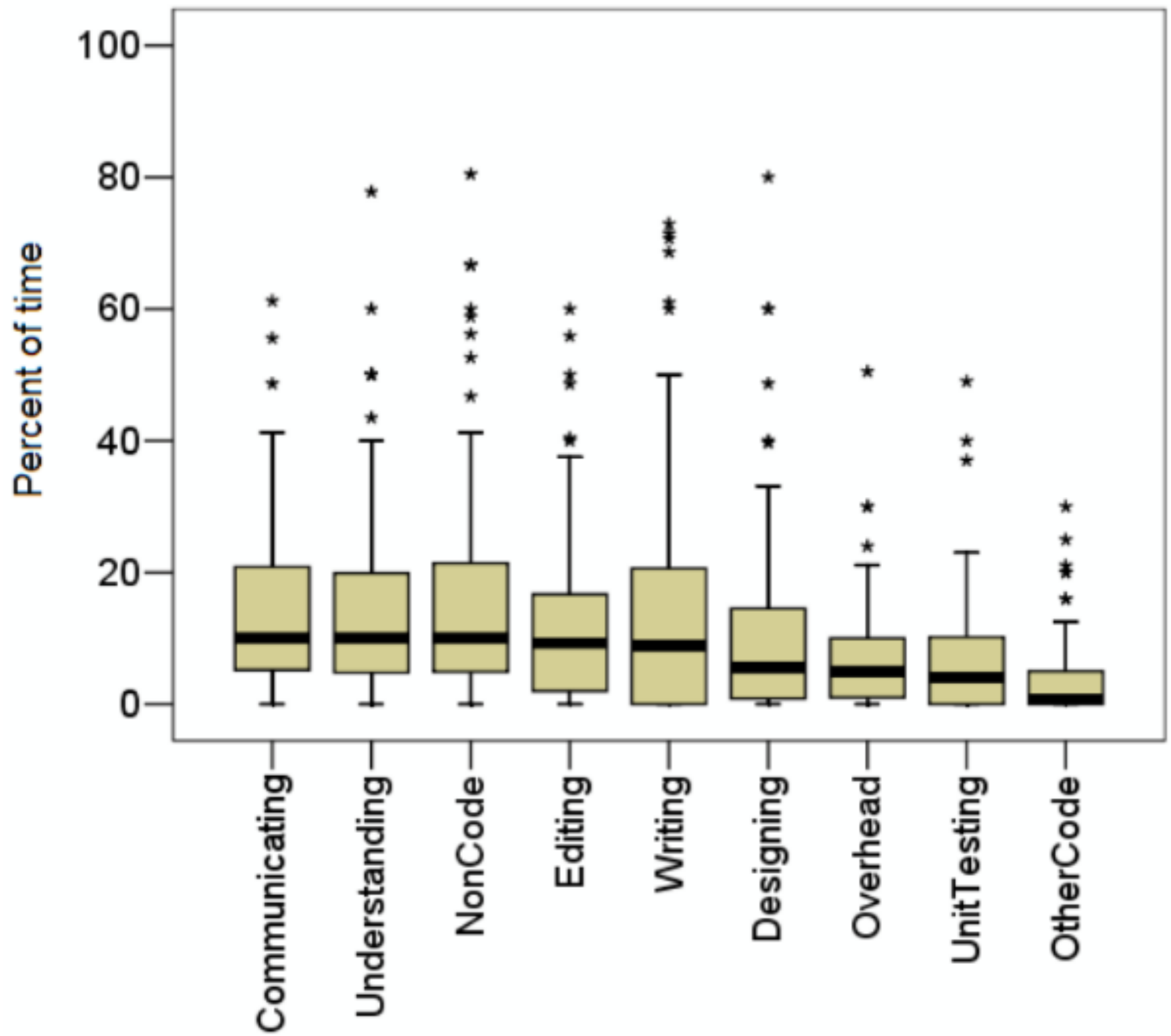
- Very important (and fun) part of the course
- Not solely about producing the end product
- … but also about learning to:
  - Follow the process and produce the product "right"
  - Manage time (weekly milestones)
  - Work in teams
  - Independently learn complex new technologies
  - Explicitly define and communicate ideas (requirements milestone)
  - Consider and pick from multiple alternatives (design milestone)
  - Read other people's code (code review milestone)
  - …

Reading Other People Code



Modern Code Review: A Case Study at Google
(https://storage.googleapis.com/pub-tools-public-publication-data/pdf/80735342aebcbfc8af4878373f842c25323cb985.pdf)

What Software Engineers Do?



Maintaining Mental Models: A Study of Developer Work Habits

Qualities of Great Software Engineers

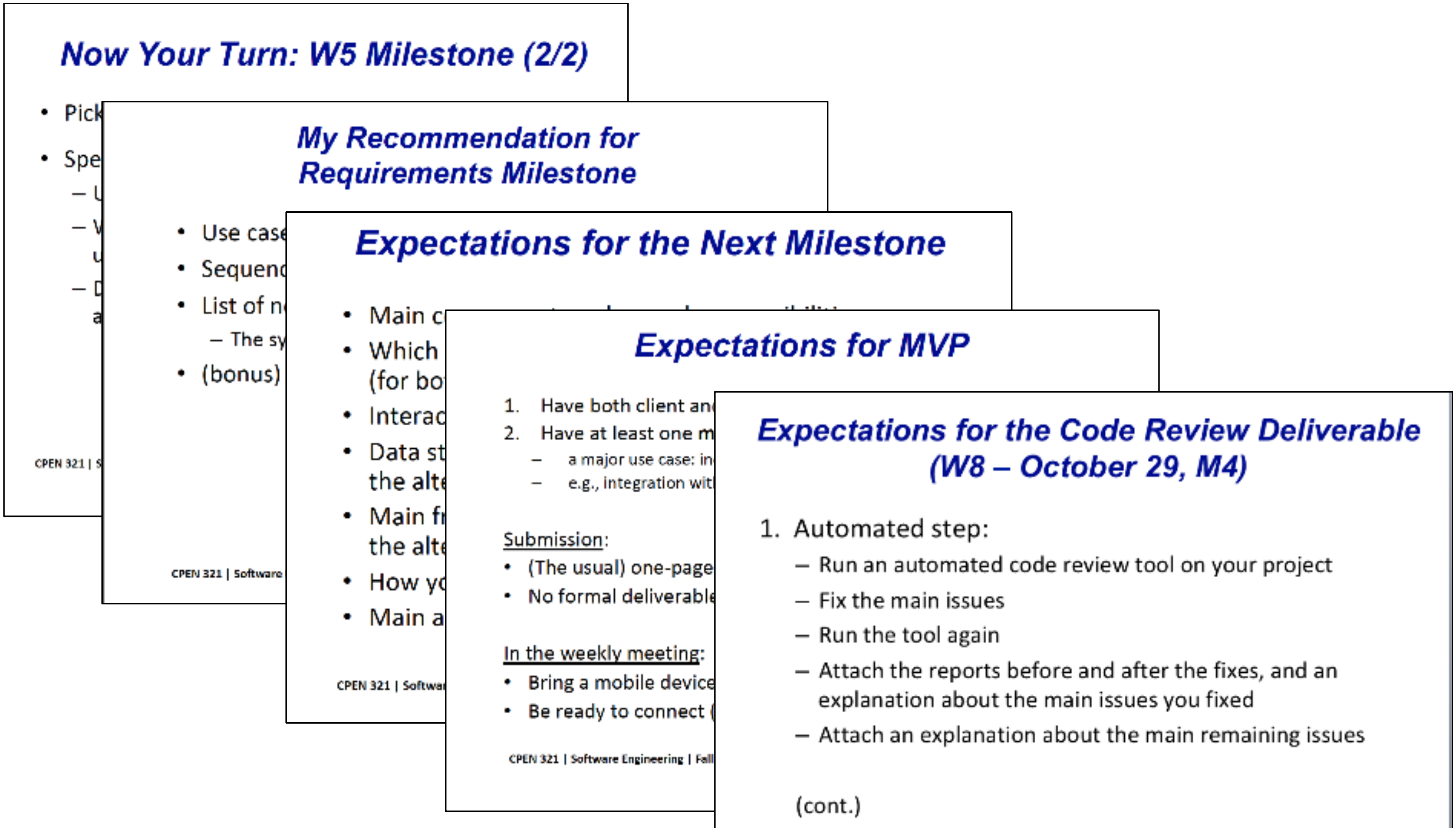
https://www.quora.com/What-makes-a-good-software-engineer:

- Excellent communication skills
- Great at time and task management
- Quick learning ability
- Technical programming skill and experience
- A good team player
- High end-user focus
- Stay positive and patient

Summary: What you are learning in this course

- Software Engineering (! Programming)
- Engineering practices for the development of non-trivial software-intensive systems including requirements specification, software architecture, implementation, verification, and maintenance. Iterative development. Recognized standards, guidelines, and models.
- “Soft” skills: time management; teamwork; ability to independently evaluate multiple options and make decisions, etc.

Description of the Milestones



If you follow the guidelines, you will score 100%

Expectations for the Code Review Deliverable (Same as Last Class)

1. Automated step:
   - Run an automated code review tool on your project
   - Fix the main issues
   - Run the tool again
   - Attach the reports before and after the fixes, and an explanation about the main issues you fixed
   - Attach an explanation about the main remaining issues

2. Manual step:
   - Review the code of your customer team
   - Attach a report describing one **major** issue you identified and how this issue was fixed (not about comments!)

3. Be ready to dig into your code with your customer and TA during the meeting

Example TA's Grading Rubric (No Surprises)

- 50 points for automated part
  - [0..-10] for not fixing major things
  - [0..-10] for not clarifying before and after
  - [0..-10] for not explaining the remaining things

- 50 points for manual part
  (major issues, e.g., security, inefficiency in algorithms)
  - [0..-10] for not focusing on major issues (deduce points to the customer)
  - [0..-5] for customers not knowing what they were looking for (deduce points to the customer)
  - [0..-10] for not fixing the found problem (deduce point to the developer)

Topics to Cover in the Rest of the Class

- Continuous Integration, DevOps
  - I have a co-op for DevOps starting in January and I don't know what it is
- Design patterns
  - At my interview, the guy asked me if I knew any design patterns and said it should be taught because it's super important
- Security
- Aside from Agile, what other design processes are being practiced effectively in the real world? [also aside from waterfall]
- Proper maintenance of a software system.
- How to actually build an app
- Docker and Kubernetes
  - I have done multiple interviews and went to multiple career fairs and almost all employers want these 2 aspects to be understood (especially Docker)

Agenda

- Follow-up on mid-term survey
- **Into to testing**
  - Types of testing
  - When to stop testing
- Expectations for the Testing Milestone (M5, November 5)

Terminology

Error: incorrect software behavior

- Example: Software controller for the Ariane 5 rocket crashed (and so did the rocket).

Fault (bug): mechanical or algorithmic cause of error

- Example: Conversion from 64-bit floating point to 16-bit signed integer value caused an exception.

Software Quality Control Techniques

Fault avoidance: prevents errors before the system is released.

- reviews, inspections, walkthroughs, development methodologies, testing, verification

Fault tolerance: enables the system to recover from (some classes of) errors by itself.

- rollbacks, redundancy, adaptations

Software Quality Control Techniques

Fault avoidance: prevents errors before the system is released.

- reviews, inspections, walkthroughs, development methodologies, **testing,** verification

Fault tolerance: enables the system to recover from (some classes of) errors by itself.

- rollbacks, redundancy, adaptations

Bug

- In 1946, operators traced an error in Harvard's Mark II computer to a **moth** trapped in a relay, coining the term bug.

- This bug was carefully removed and taped to the log book.

More Terminology

- **Test plan**: A document describing the scope, approach, resources, and schedule of intended test activities
- **Test case**: a single unique unit of testing code
- **Test suite**: collection of test cases
- **Test oracle**: expected behavior
- **Test harness:** collection of all the above

Software Tests, Simply Put

1. Choose input data
2. Define the expected outcome
3. Run on the input to get the actual outcome
4. Compare the actual and expected outcomes

Software Testing

- The dynamic verification of the behavior of a program
- … on a finite set of test cases
- … **suitably** selected from the usually infinite executions domain
- … against the specified expected behavior (oracle)

Questions

- How to select the right test cases?
- How to capture test cases?
- When to test?
- When to stop testing? (the domain of test cases is infinite)
- How to determine the expected result?

Example: Ad-Hoc Testing

```
// Effects: If x==null throw Null Exception
// else return the number of positive elements in x

public int countPositive(int[] x) {
  int count = 0;
  for (int i=0; i<x.length; i++) {
    if (x[i] >= 0) {
      count++;
    }
  }
  return count;
}
```

Tasks

(a) Identify the **fault**.

(b) Write a test case that does not **execute** the statements related to the fault.

(c) Write a test case that executes the statements related to the fault, but does not result in a **detectable error state**.

(d) Write a test case that **detects the fault**.

Example: Ad-Hoc Testing

```
// Effects: If x==null throw Null Exception
// else return the number of positive elements in x

public int countPositive(int[] x) {
  int count = 0;
  for (int i=0; i<x.length; i++) {
    if (x[i] >= 0) {
      count++;
    }
  }
  return count;
}
```

*(a) Identify the **fault**.*

*(b) Write a test case that does not **execute** statements related to the fault.*

*(c) Write a test case that executes the statements related to the fault, but does not result in a **detectable error state**.*

*(d) Write a test case that **detects the fault**.*

(a) The fault

The fault is **=> 0 inside the if**. The correct check must have been:

if (x[i] > 0) {

```
public int countPositive(int[] x) {
   int count = 0;
   for (int i=0; i<x.length; i++) {
      if (x[i] >= 0) {
         count++;
      }
   }
   return count;
}
```

(b) Write a test case that does not execute the fault.

x must be either null or empty. All other inputs result in the fault being executed. We give the empty case here.

Input: x = []

Expected Output: 0, Actual Output: 0

```
public int countPositive(int[] x) {
  int count = 0;
  for (int i=0; i<x.length; i++) {
    if (x[i] >= 0) {
      count++;
    }
  }
  return count;
}
```

(c) does not result in a detectable error state.

Any **nonempty** x without a 0 entry works fine.

Input: x = [1, 2, 3]

Expected Output: 3, Actual Output: 3

```
public int countPositive(int[] x) {
  int count = 0;
  for (int i=0; i<x.length; i++) {
    if (x[i] >= 0) {
      count++;
    }
  }
  return count;
}
```

(d) detects the fault.

(d) Input: x = [-4, 2, 0, 2]

Expected Output: 2, Actual Output: 3

First Error State: i = 2; count = 1;

```
public int countPositive(int[] x) {
  int count = 0;
  for (int i=0; i<x.length; i++) {
    if (x[i] >= 0) {
      count++;
    }
  }
  return count;
}
```

Testing Dimensions

- Black-box vs. white-box testing
- Manual vs. automated testing
- Test-last vs. test-first
- Regression testing

Black-Box Testing

- Software program or system under test is viewed as a “**black box**”.
- Emphasizes on the **external** behavior of the software entity.
- Selection of test cases for functional testing is based on the **requirement** or design **specification** of the software entity under test.

White-Box Testing

- Emphasizes on the **internal** structure of the software entity.
- Goal of selecting such test cases is to cause the execution of specific **statements**, program **branches** or **paths**.
- Expected results are evaluated against certain assertions, on a set of **coverage criteria**. Examples: path coverage, branch coverage, and data-flow coverage.

Black-Box vs. White-Box Testing

Black-box:

- Process is not influenced by component being tested
  - Assumptions embodied in code not propagated to test data.
- Robust with respect to changes in implementation
  - Test data need not be changed when code is changed
- Allows for independent testers
  - Testers need not be familiar with code

White-box:

- Can find bugs in the implementation that are not covered by the specification
  - Control-flow details
  - Performance optimizations
  - Alternate algorithms for different cases
- Yields useful test cases
- BUT: test might have same bugs as implementation

Levels of Automation

- Manual Testing
  - Manually creating test cases
  - No automation
  - Hard to repeat
- Test Scripting
  - Manually creating test cases
  - Automated test execution
  - Repeatable
- Test Generation
  - Automatically generate test cases
  - Based on some criteria, e.g., path coverage
  - Oracle problem

Manual Versus Automated Testing

Manual Testing

+ Clever test case design

+ Interaction with system inspiration for new tests

+ Human oracle

- Single test case execution

- Limited data

- Ad hoc and might not be repeatable

Automated Testing

+ Clever test case design

+ Repeatable, facilitates continuous testing

+ More test cases and input data possible

+ Human Oracle (documented)

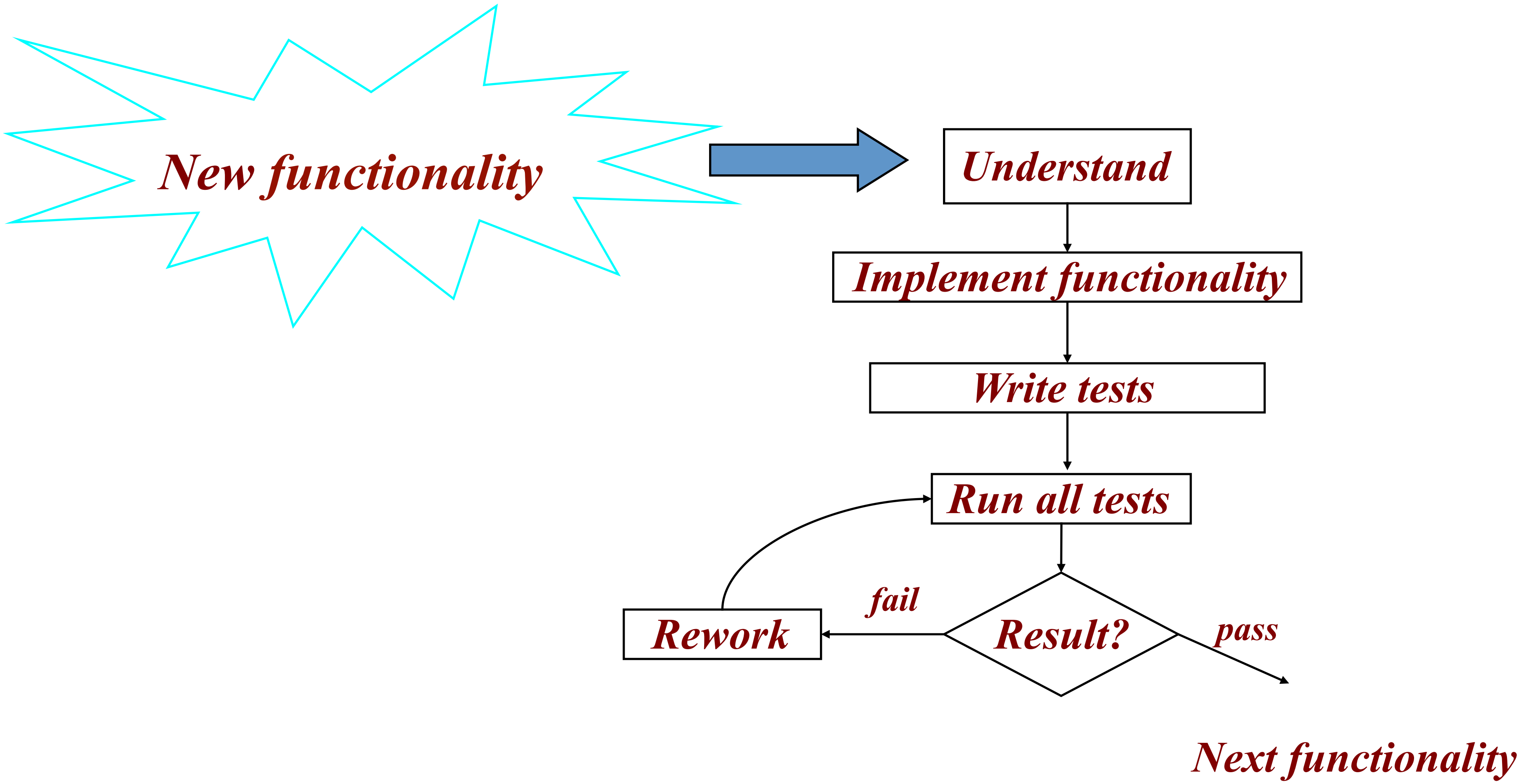
- Cost of setting up test infrastructure

- Maintenance cost of test suites

When to test?

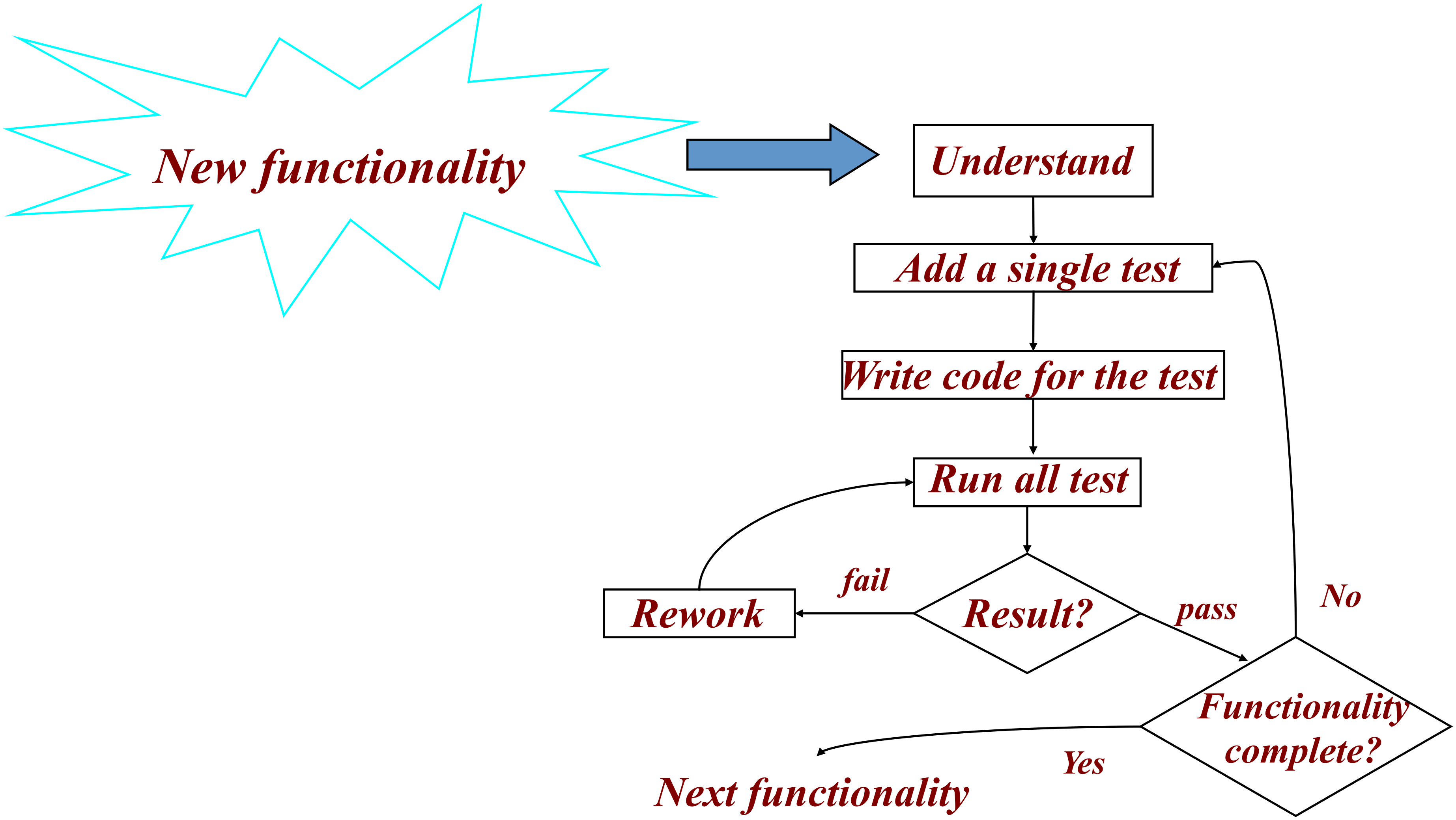
- *Test last*
  - *The conventional way for testing in which testing follows the implementation*
- *Test first*
  - *The agile view in which testing is used as a development tool*

Test last



New functionality
Understand
Implement functionality
Write tests
Run all tests
fail
Rework
Result?
pass
Next functionality

Test First


New functionality
Understand
Add a single test
Write code for the test
Run all test
Rework
fail
Result?
pass
No
Functionality complete?
Yes
Next functionality

Regression Testing

Verifies that software which was previously developed and tested still performs the same way after it was changed or interfaced with other software.

Regression Testing

- Whenever you find a bug
  - Store the input that elicited that bug, plus the correct output
  - Add these to the test suite
  - Check that the test suite fails
  - Fix the bug and verify the fix

- Why is this a good idea?
  - Ensures that your fix solves the problem.
  - Helps to populate test suite with good tests.
  - Protects against versions that reintroduce the bug.
  - It happened at least once, and it might happen again

Summary: Testing Best Practices

- **TDD:** When you need to add new functionality to the system, write the **tests first**. Then, you will be done developing when the test runs.
- **Regression:** When someone discovers a bug in your code, first write a test case that fails (finds the failure). Then debug and repair the code until the test succeeds.
- **Automation!**

Test Automation Frameworks

- Automated testing: JUnit, Selenium, Appium, Cucumber
- GUI testing: Selenium, Robotium, Monkey
- Running regression tests: TravisCI

When to Stop Testing?

Bill Sempf
@sempf

QA Engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 999999999 beers. Orders a lizard. Orders -1 beers. Orders a sfdeljknesv.

Systematic Testing

Black-box testing:

- *Test cases come from requirements / user stories.*

White-box testing:

- *Inspect the code / coverage criteria to see if you missed cases*

Measuring Test Suite Quality with Coverage

- Various kinds of coverage
  - **Statement**: is every statement run by some test case?
  - **Branch**: is every direction of an if or while statement (true or false) taken by some test case?
  - **Path**: is every path through the program taken by some test case?

Statement coverage

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once

```
void foo (z) {
  var x = 10;
  if (z++ < x) {
    x=+ z;
  }
}
```

```
void testFoo() {
  foo(10);
}
```

- Coverage:

$$\frac{\text{\# executed statements}}{\text{\# statements}}$$

Branch Coverage

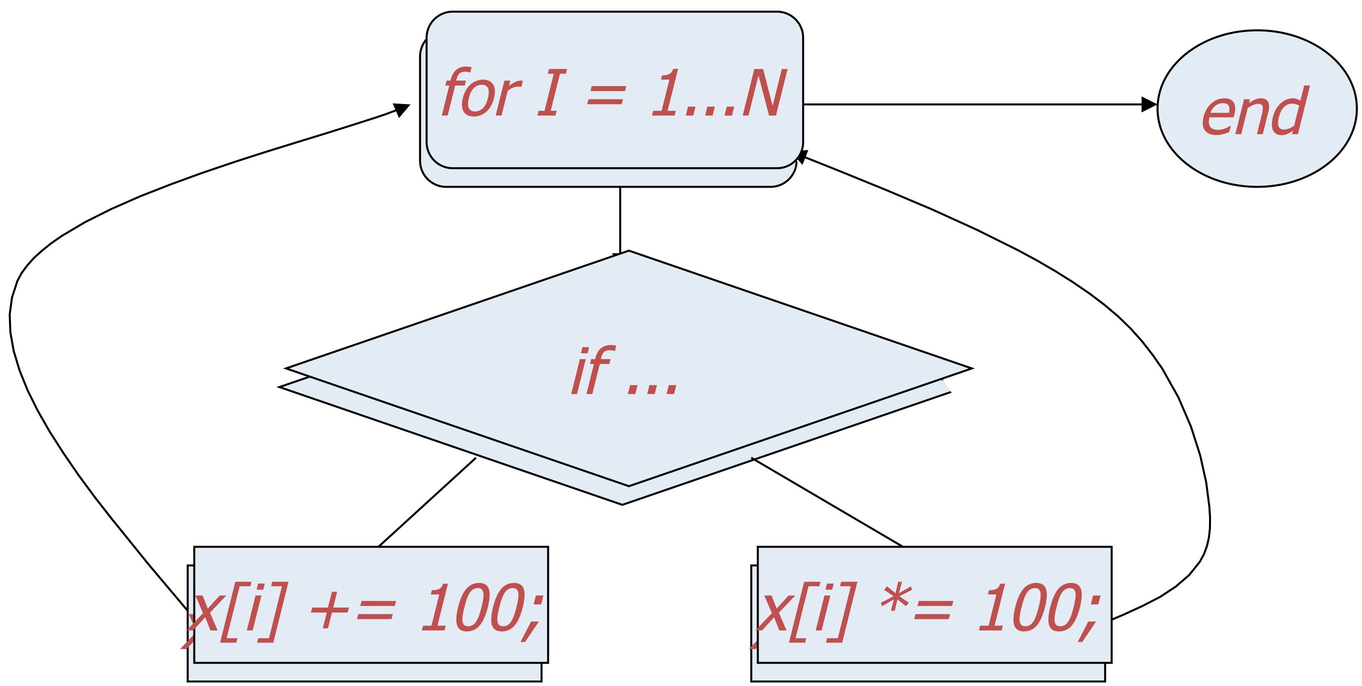
- Every path going out of a node executed at least once
  - Coverage: percentage of edges hit.

- *To achieve 100%, each predicate must be both true and false*

Path Coverage

Each path must be executed at least once

Coverage:
<u># executed path</u>
 # paths



Loops?

- Limit the number of traversals of loops: Zero, once, many

Branch vs. Path Coverage

```
if( cond1 )
    f1();
else
    f2();

if( cond2 )
    f3();
else
    f4();
```

How many test cases to achieve branch coverage?

Branch vs. Path Coverage

```
if( cond1 )
    f1();
else
    f2();

if( cond2 )
    f3();
else
    f4();
```

How many test cases to achieve branch coverage?

Two, for example:

1. ***cond1: true, cond2: true***
2. ***cond1: false, cond2: false***

Branch vs. Path Coverage

```
if( cond1 )
    f1();
else
    f2();

if( cond2 )
    f3();
else
    f4();
```

What about path coverage?

Branch vs. Path Coverage

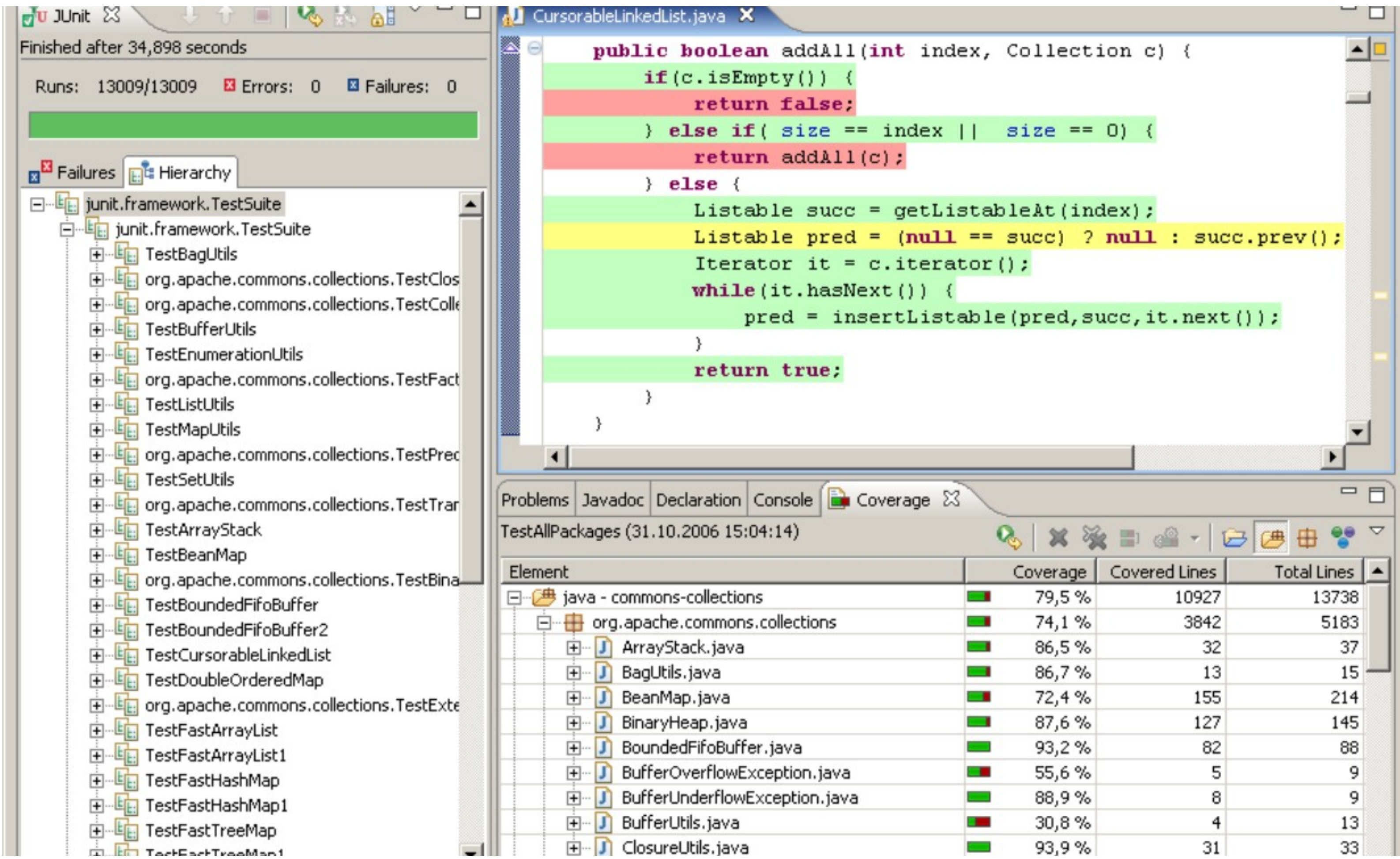
```
if( cond1 )
    f1();
else
    f2();

if( cond2 )
    f3();
else
    f4();
```

What about path coverage?

Four:

1. ***cond1: true, cond2: true***
2. ***cond1: false, cond2: true***
3. ***cond1: true, cond2: false***
4. ***cond1: false, cond2: false***

Tools for Measuring Coverage (e.g., Emma, EclEmma)



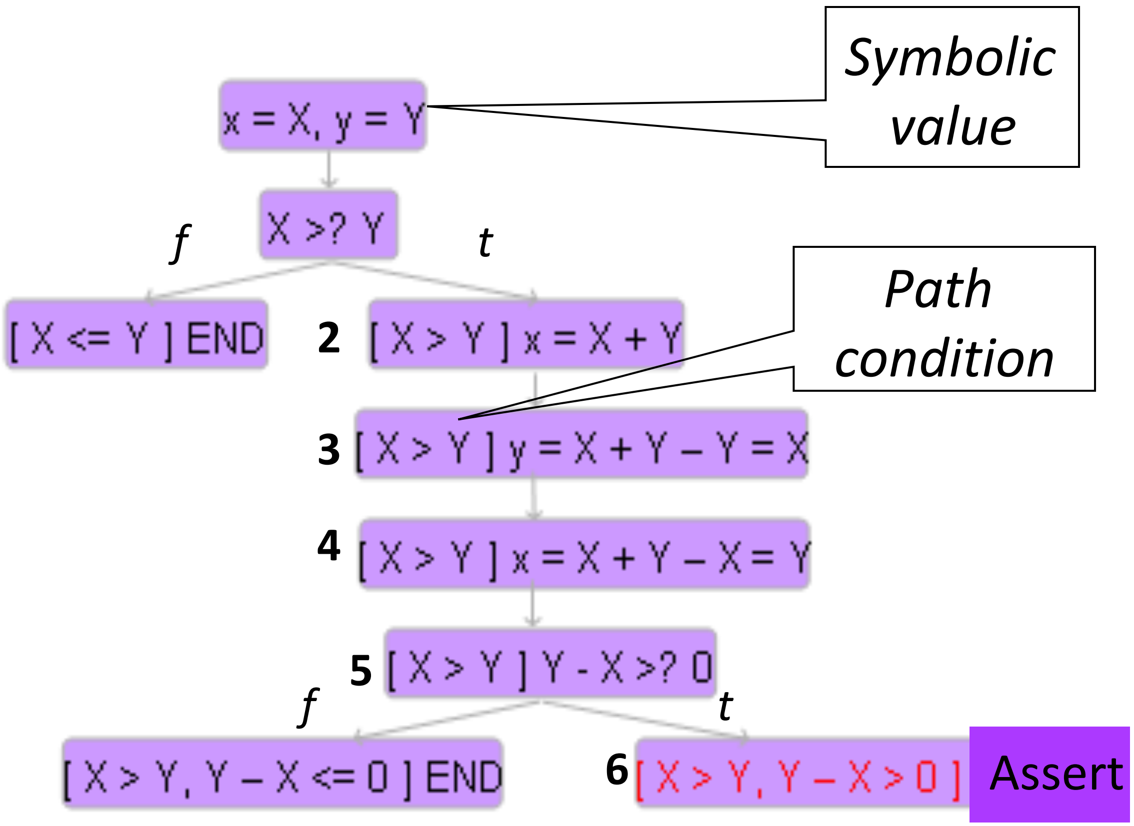
How to achieve perfect path coverage?

```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:        assert(false); //error
   }
```

Is line 6 reachable?

Symbolic Execution by Example

```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:        assert(false);
   }
```

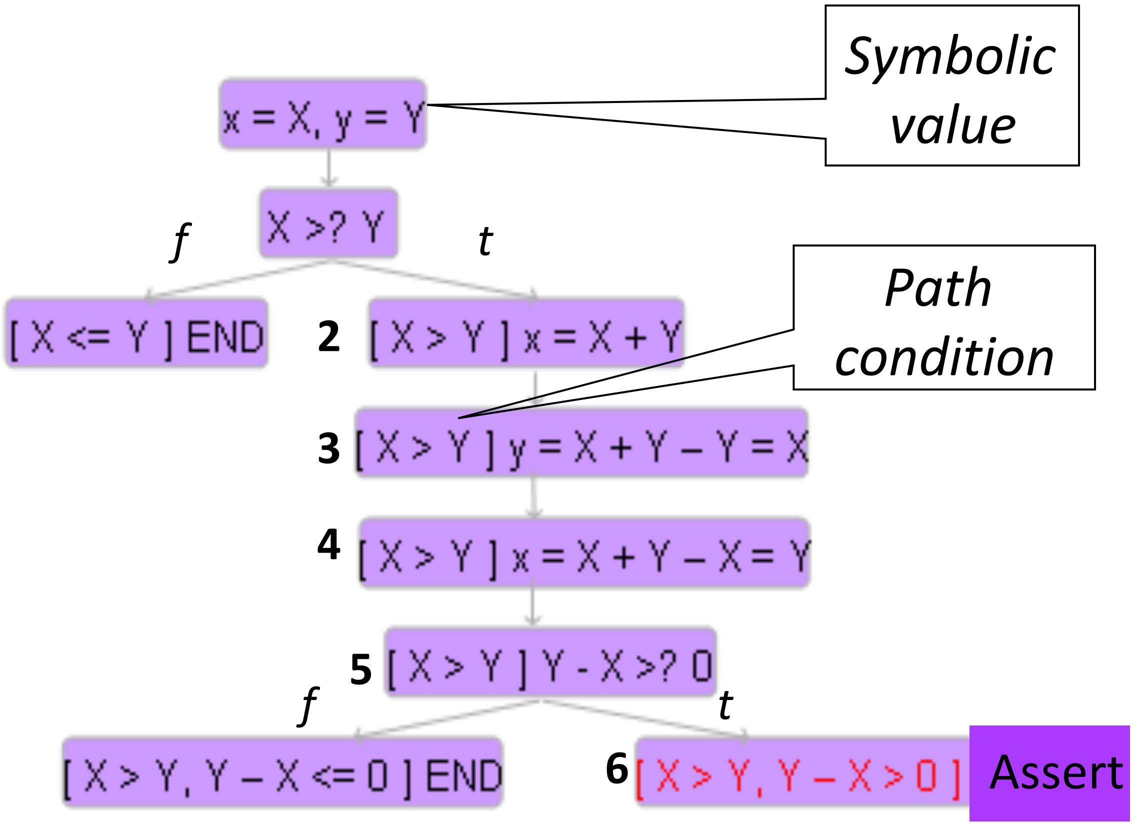


Is the assert (line 6) reachable?

2: x = X+Y, y=Y

Symbolic Execution by Example

```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:         assert(false);
   }
```

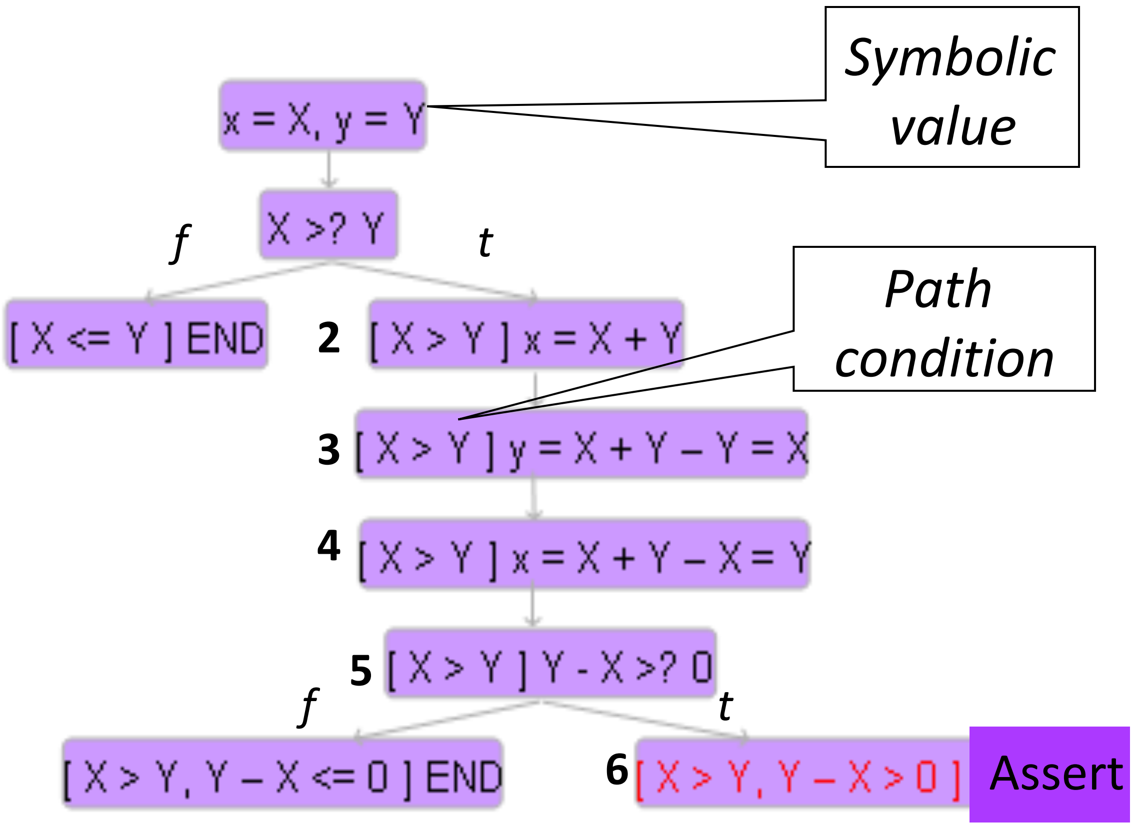


Is the assert (line 6) reachable?

3: x = X+Y, y=X

Symbolic Execution by Example

```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:         assert(false);
   }
```

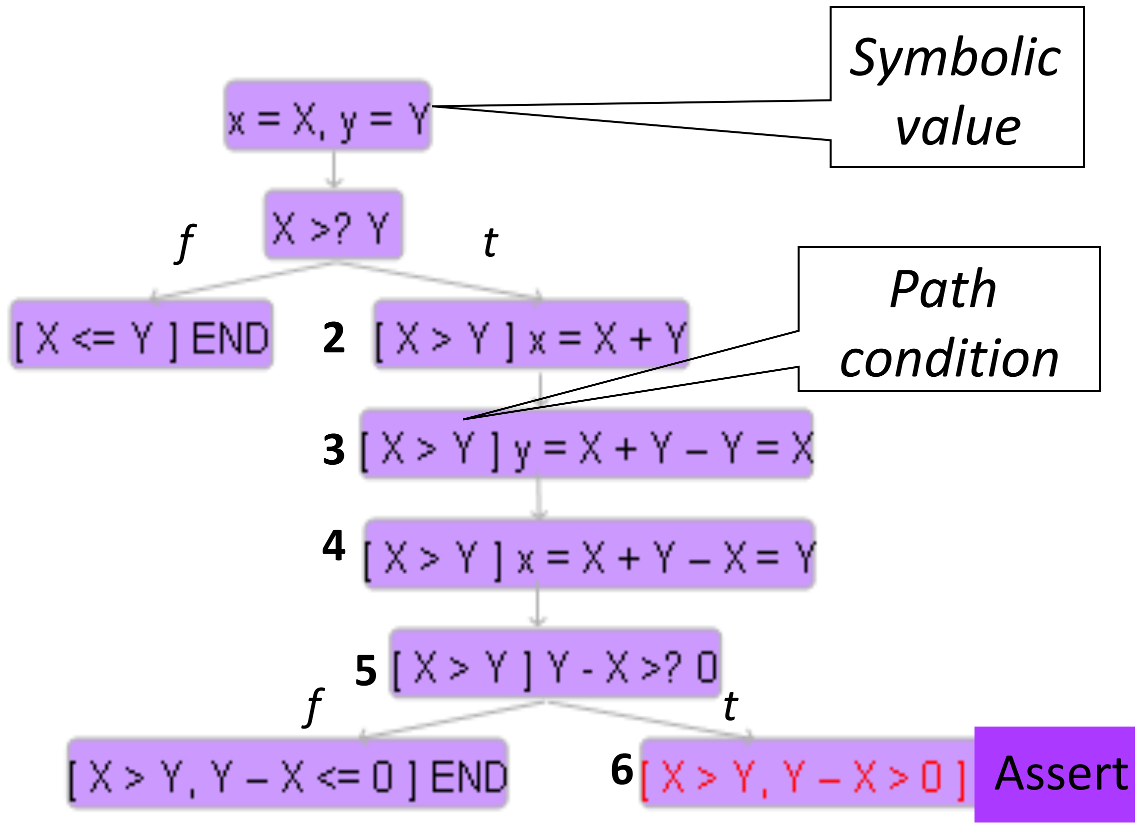


Is the assert (line 6) reachable?

4: x = Y, y=X

Symbolic Execution by Example

```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:         assert(false);
   }
```

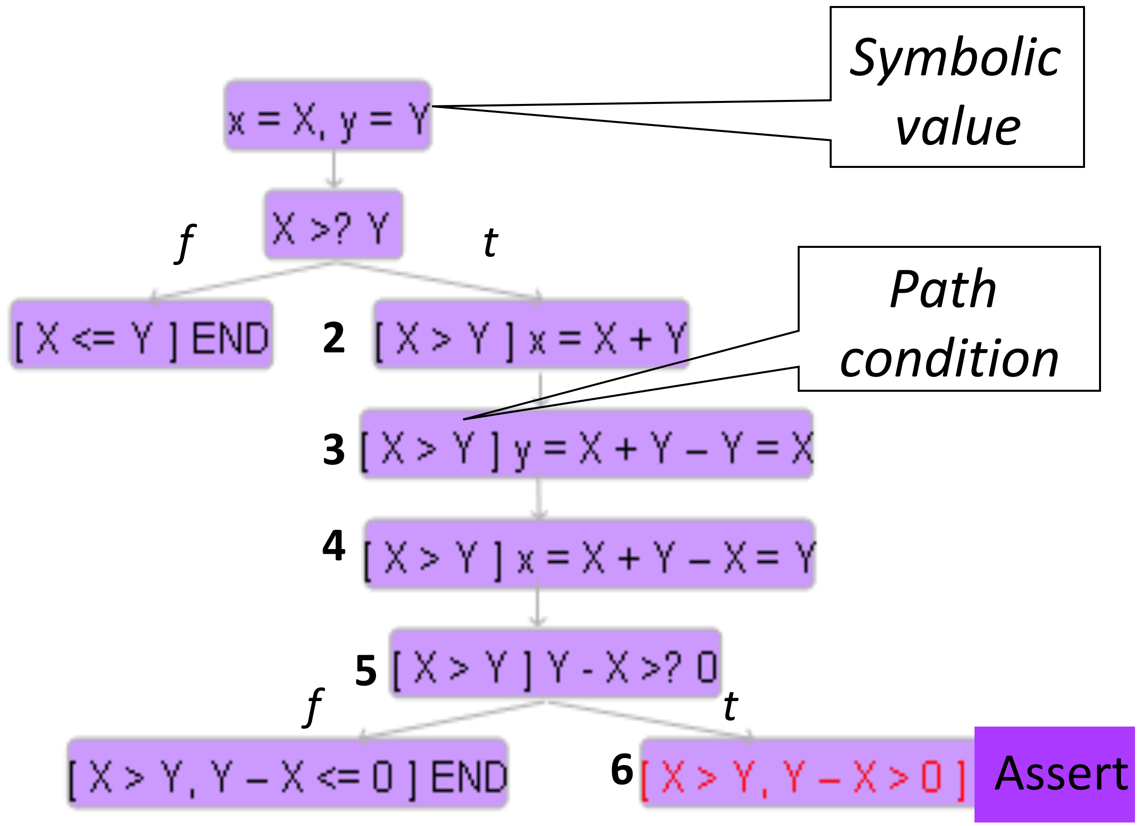


Is the assert (line 6) reachable?

5: Y-X > 0?

Symbolic Execution by Example

```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:         assert(false);
   }
```



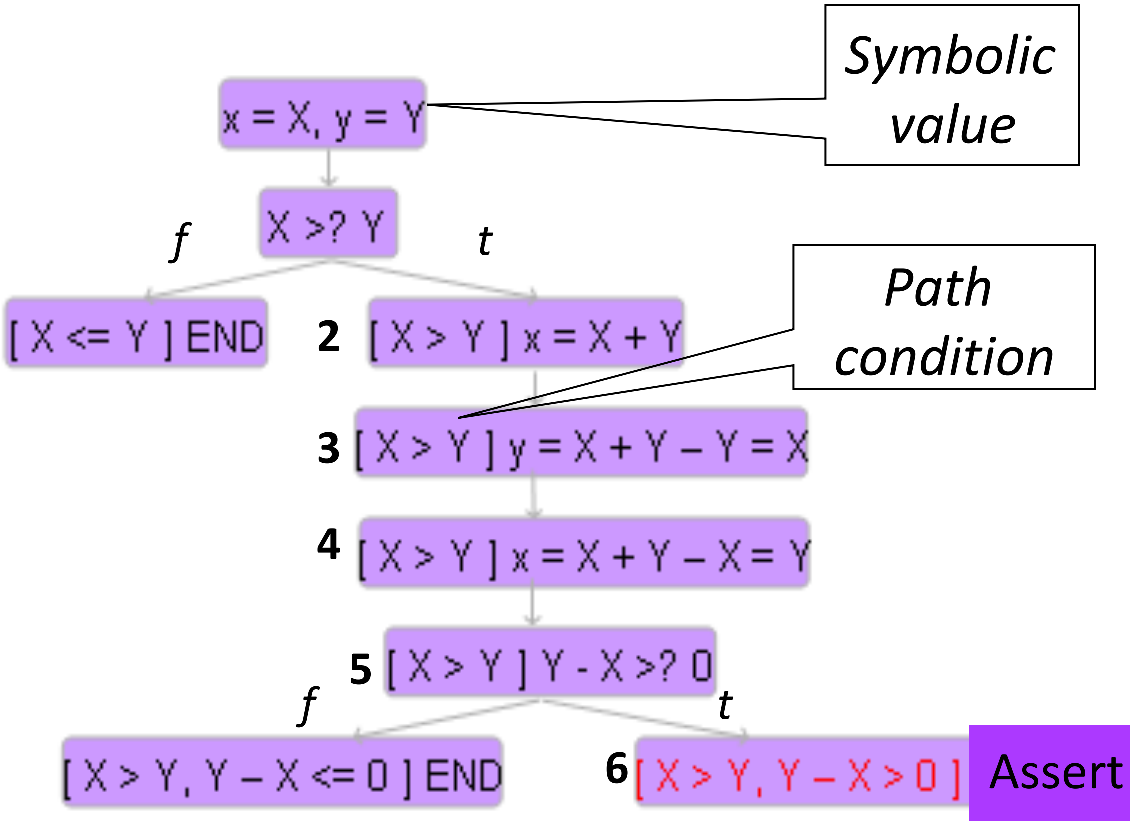
Is the assert (line 6) reachable?

Condition for 6:

X>Y & Y-X > 0

Symbolic Execution by Example

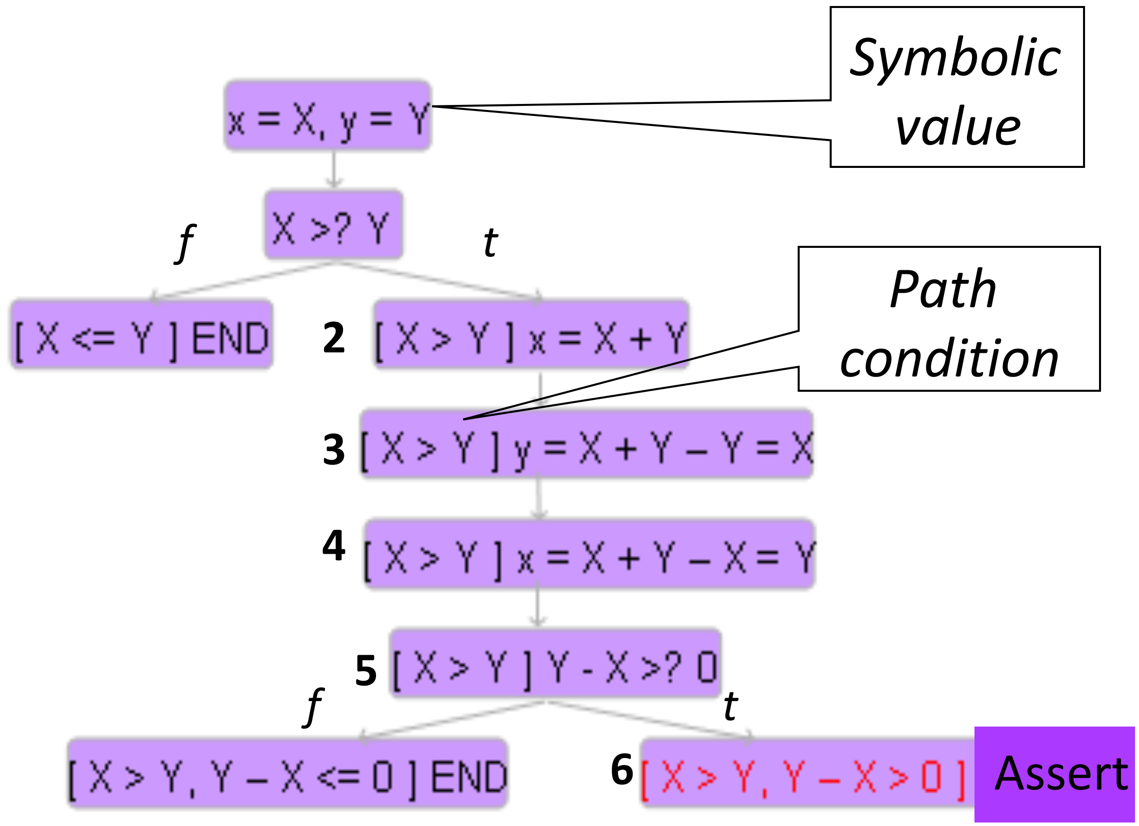
```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:         assert(false);
   }
```



Is the assert (line 6) reachable? NO!

Symbolic Execution by Example

```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:        assert(false);
   }
```



Two equivalence classes (a) X<=Y (b) X>Y

Applications of Symbolic Execution

- Guiding the test input generation to cover all branches
- Identifying infeasible program paths
- Security testing
- …

Limitations of Symbolic Execution

- Expensive
  - Executing all feasible program paths is exponential in the number of branches
  - Does not scale to large programs
- Problems with function calls
- Problem with handling loops
  - often *unroll* them up to a certain depth rather than dealing with termination or loop invariants

Is code coverage a good metric?

```
// Effects: If x==null throw Null Exception
// else return the number of positive elements in x

public int countPositive(int[] x) {
  int count = 0;
  for (int i=0; i<x.length; i++) {
    if (x[i] >= 0) {
      count++;
    }
  }
  return count;
}
```

Limitations of Coverage

- Coverage is just a heuristic.
- 100% coverage may not be achievable.
- 100% is not sufficient!

- Common practice: statement-level coverage + clever test selection + test case for all found bugs + regression

- More advanced techniques: input space partitioning, combinatorial testing, etc.

Next Class

- Best practices and main types of testing activities
  - Unit Testing
  - Integration Testing
  - System-level testing
- GUI Testing
- Expectations for the Testing Milestone (M5, November 5)