Software Engineering

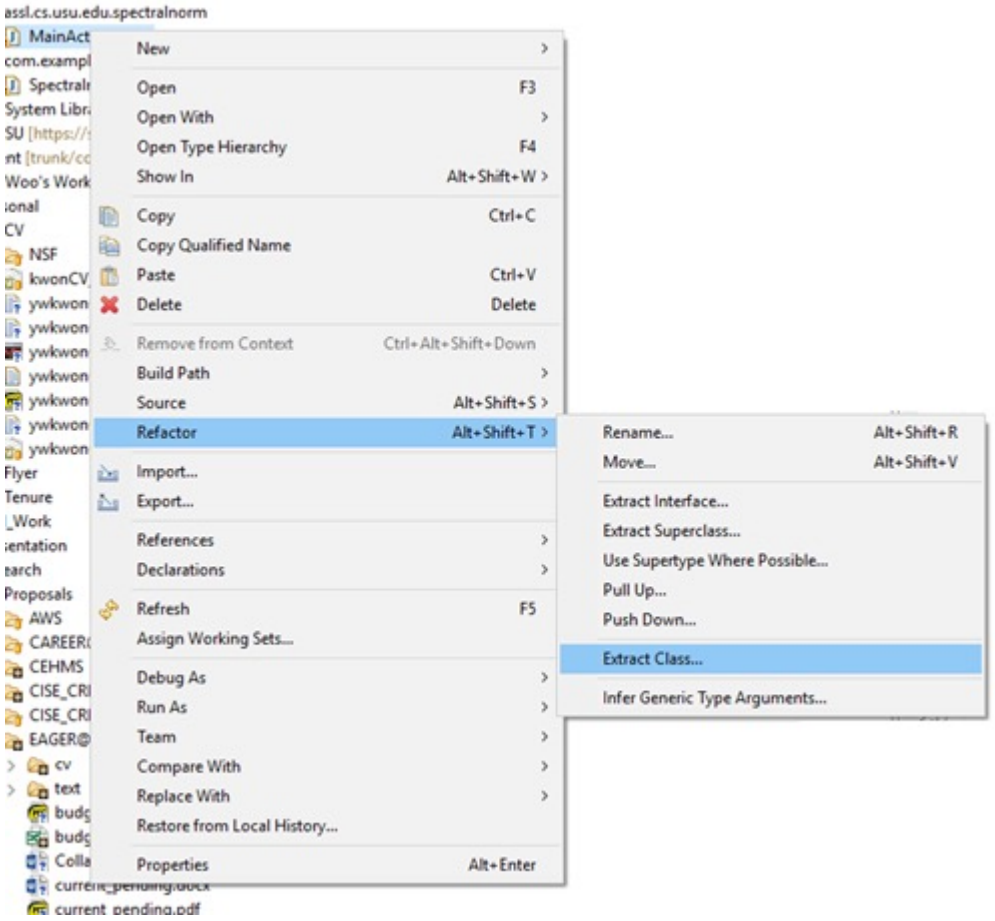
Dr. Young-Woo Kwon

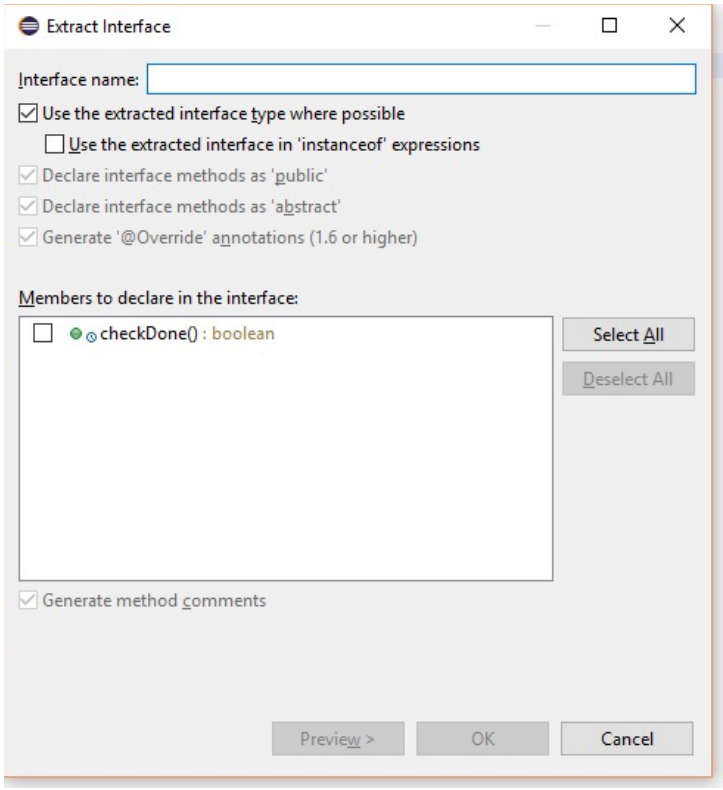
REFACTORING

Refactoring

- Refactoring is:
  - restructuring (rearranging) code...
  - ...in a series of small, **semantics-preserving transformations** (i.e. the code keeps working)...
  - ...in order to make the code easier to maintain and modify
- Refactoring is *not* just any old restructuring
  - You need to keep the code working
  - You need small steps that preserve semantics
  - You need to have unit tests to prove the code works
- There are numerous well-known refactoring techniques
  - You should be at least somewhat familiar with these before inventing your own

Refactoring Example (Eclipse)





When to refactor

- When should you refactor?
  - *Any* time you find that you can improve the design of existing code
  - You detect a “bad smell” (an indication that something is wrong) in the code
- When *can* you refactor?
  - You should be in a supportive environment (agile programming team, or doing your own work)
  - You should have an adequate set of unit tests

Why Refactoring

- Refactoring improves the design of the software
  - Removes duplicated code:
    - reduces the amount of code
- Refactoring makes software easier to understand
  - helps make your code more readable
  - improves program comprehension
- Refactoring helps you program faster
  - this sounds counterintuitive
  - less bugs, no patches
  - helps correct bugs

What to refactor?

- Code Smell (or bad smell)
  - You should refactor any time you detect a “bad smell” in the code
- Examples of bad smells include:
  - Duplicate Code
  - Long Methods
  - Large Classes
  - Long Parameter Lists
  - Data Clumps
  - Switch Statements
  - Lazy Classes
  - Temporary Fields
  - Comments

How to refactor?
Example: *Rename Method*

- Manual approach:
  - Check that no method with the new name already exists in any subclass or superclass.
  - Browse all the implementers (method definitions)
  - Browse all the callers (method invocations)
  - Edit and rename all implementers
  - Edit and rename all callers
  - Test
- Automated (or automatic) refactoring is better !

Well-Known Refactoring Techniques

- Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts, *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 1999.

Class Refactorings	Method Refactorings	Attribute Refactorings
add (sub)class to hierarchy	add method to class	add variable to class
rename class	rename method	rename variable
remove class	remove method	remove variable
	push method down	push variable down
	push method up	pull variable up
	add parameter to method	create accessors
	move method to component	abstract variable
	extract code in new method	

Code Smell I: Duplicated code

- Martin Fowler refers to duplicated code as “Number one in the stink parade”
  - The usual solution is to apply *Extract Method:* Create a single method from the repeated code, and use it wherever needed
- This adds the overhead of method calls, thus the code gets a bit slower
  - Is this a problem?

Code Smell II: Long methods

- Another “bad smell” is the overly long method
- Almost always, you can fix long methods by applying *Extract Method*
  - Find parts of the method that seem to perform a single task, and make them into a new method
- Potential problem: You may end up with lots of parameters and temporary variables
  - Temporaries: Consider *Replace Temp With Query*
  - Parameters: Try *Introduce Parameter Object* and *Preserve Whole Object*
  - If all else fails, use *Replace Method With Method Object*

Replace Temp With Query

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
  return basePrice * 0.95;
else
  return basePrice * 0.98;
```

```
if (basePrice() > 1000)
  return basePrice() * 0.95;
else
  return basePrice() * 0.98;
...
double basePrice() {
  return _quantity * _itemPrice;
}
```

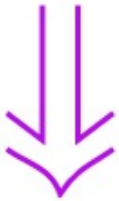
Preserve Whole Object

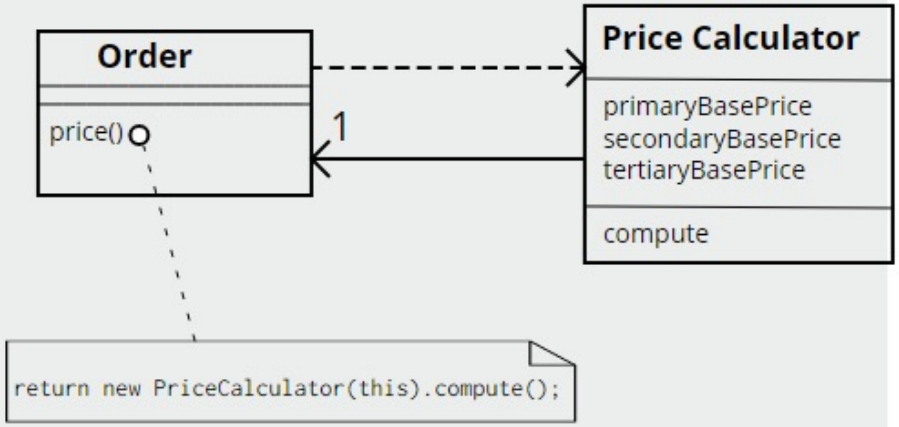
```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

```
withinPlan = plan.withinRange(daysTempRange());
```

Replace Method With Method Object

```
class Order...
double price() {
  double primaryBasePrice;
  double secondaryBasePrice;
  double tertiaryBasePrice;
  // long computation;
  ...
}
```





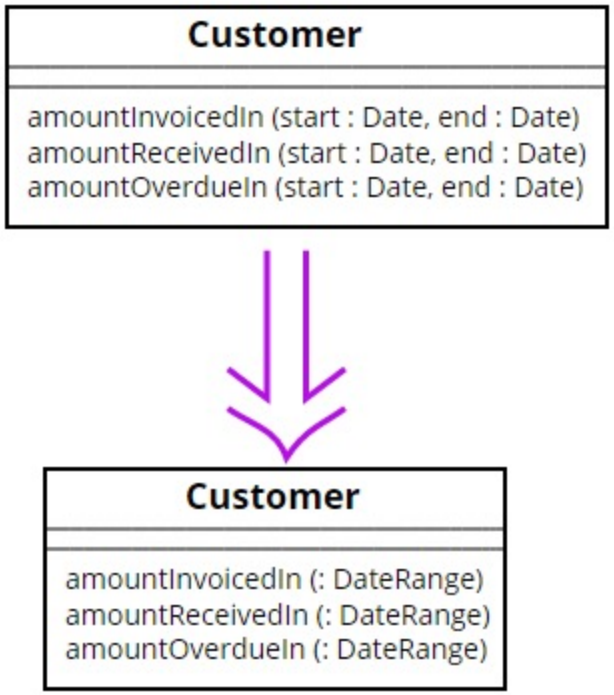
Code Smell III: Long parameters

- Problem: You have a method that requires a long parameter list
  - You may have a group of parameters that go naturally together
  - If so, make them into an object

Introduce Parameter Object

```
public void marry(String name, int age,
                  boolean gender, String name2,
                  int age2, boolean gender2) {...}
```

→ `public void marry(Person person1, Person person2) {...}`



Code Smell IV: Large class

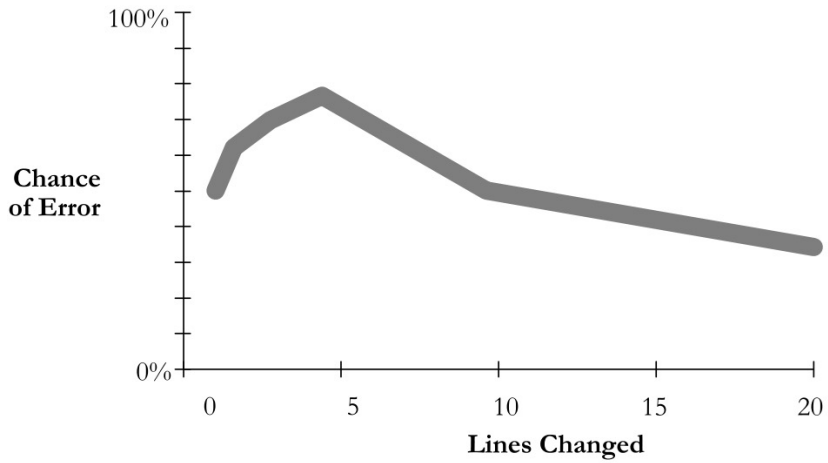
- Classes can get overly large
  - Too many instance variables
  - More than a couple dozen methods
  - Seemingly unrelated methods in the same class
- Possible refactorings are *Extract Class* and *Extract Subclass*
- A related refactoring, *Extract Interface,* can be helpful in determining how to break up a large class

Extract Class

- *Extract Class* is used when you decide to break one class into two classes
  - Classes tend to grow, and get more and more data
  - Good signs:
    - A subset of the data and a subset of the methods seem to go together
    - A subset of the data seems to be interdependent
  - Useful tests:
    - If you removed a method or a particular piece of data, what other fields and methods would become nonsense?
    - How is the class subtyped?
  - The actual refactoring technique involves creating a new (empty) class, and repeatedly moving fields and methods into it, compiling and testing after each move

Dangers of refactoring

- code that used to be ...
  - well commented, now (maybe) isn't
  - fully tested, now (maybe) isn't
  - fully code reviewed, now (maybe) isn't
- easy to insert a bug into previously working code (regression!)
  - a small initial change can have a large chance of error



Why refactoring hard?

- Culture:
  - "refactoring is an overhead activity. I'm paid to write new, revenue-generating features".
  - "What do I tell my manager?"
    - When it is not part of XP: Don't tell!! Treat it as part of the profession to produce a better code (this is how you construct code, it is not viewed by you as an additional work).

Why refactoring hard?

- Internal resistance: Why are developers reluctant to refactor? (Opdyke, in Fowler's book, p. 313)
  - it should be executed when the code runs and all the tests pass.
  - developers might not understand how to refactor.
  - if the benefits are long-term, why exert the effort now? In the long term, they might not be with the project to reap the benefits. [satisfaction chasm]
  - it seems that time is wasted now.
  - refactoring might break the existing program.

Refactoring at Google

- "At Google, refactoring is very important and necessary/inevitable for any code base. If you're writing a new app quickly and adding lots of features, your initial design will not be perfect. Ideally, do *small* refactoring tasks early and often, as soon as there is a sign of a problem."
- "Refactoring is unglamorous because it does not add features. At many companies, people don't refactor because you don't get promoted for it, and their code turns into hacky beasts."
- "Google feels refactoring is so important that there are company-wide initiatives to make sure it is encouraged and rewarded."
- "Common reasons not to do it are incorrect:
  - a) *'Don't have time; features more important'* -- You will pay more cost, time in adding features (because it's painful in current design), fixing bugs (because bad code is easy to add bugs into), ramping up others on code base (because bad code is hard to read), and adding tests (because bad code is hard to test), etc.
  - b) *'We might break something'* -- Sign of a poor design from the beginning, where you didn't have good tests. For same reasons as above, you should fix your testing situation and code.
  - c) *'I want to get promoted and companies don't recognize refactoring work'* -- This is a common problem. Solution varies depending on company. Seek buy-in from your team, gather data about regressions and flaws in the design, and encourage them to buy-in to code quality."
- "An important line of defense against introducing new bugs in a refactor is having solid unit tests (*before* the refactor)."

-- Victoria Kirst,
Software Engineer, Google

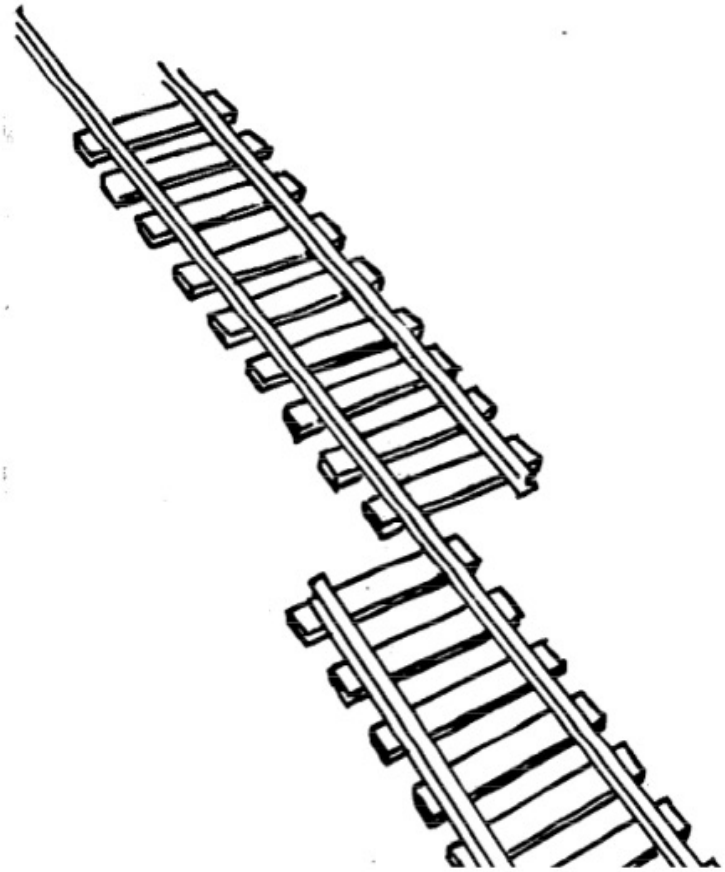
TESTING

What is this?

A fault?

An error?

A failure?

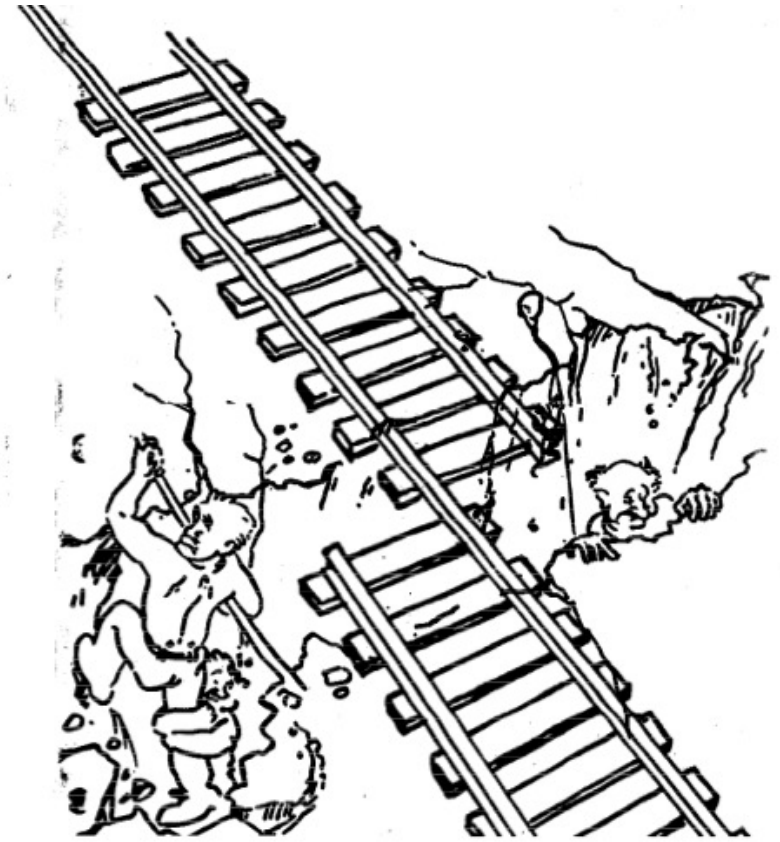


Fault vs. Error vs. Failure

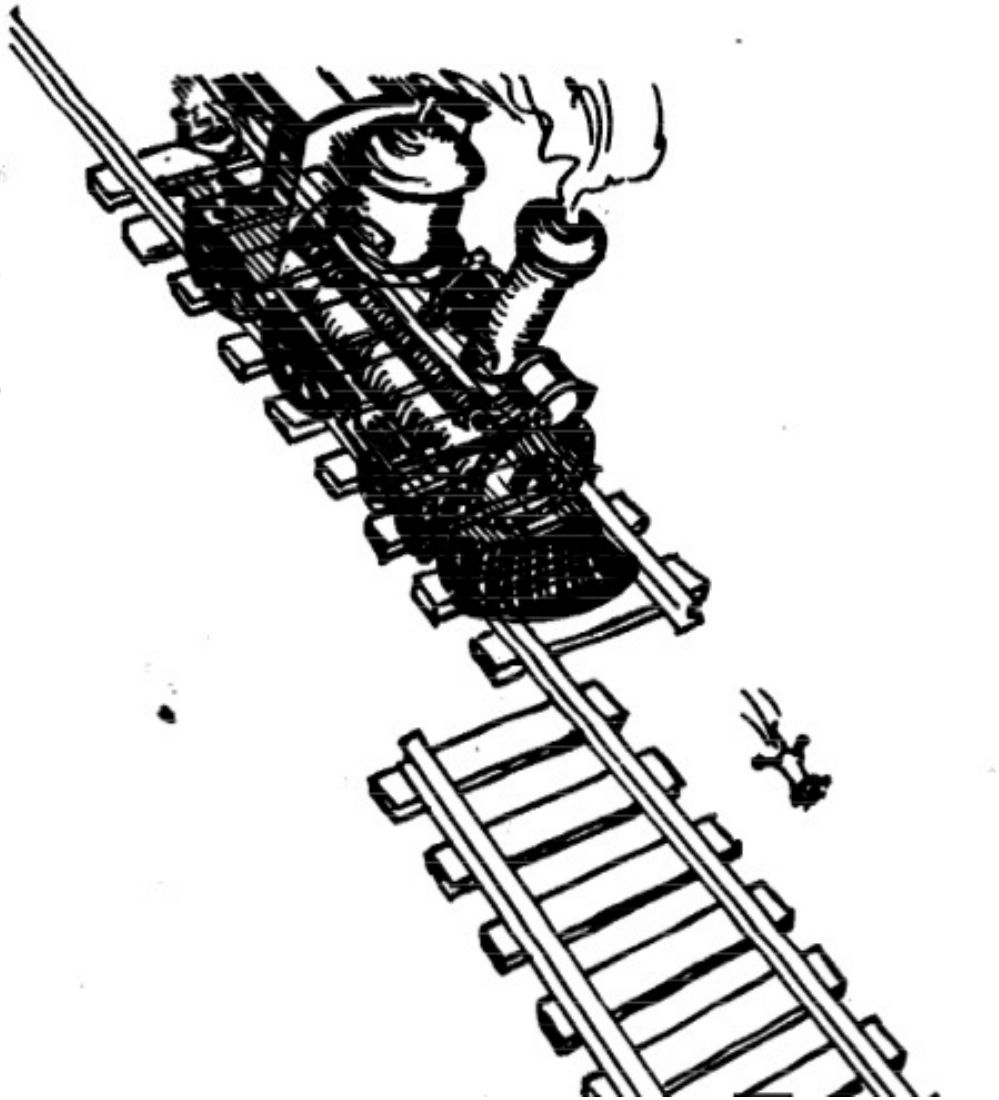
- **Fault (bug)**
  - Defect in the system that may lead to an error
  - Algorithmic and mechanical fault
- **Error**
  - The part of the total state of the system that may lead to a failure
- **Failure**
  - Deviation of the execution of a program from its intended behavior
- **Fault -> Error -> Failure**

Algorithmic Fault

Mechanical Fault



Erroneous State (“Error”)



Examples

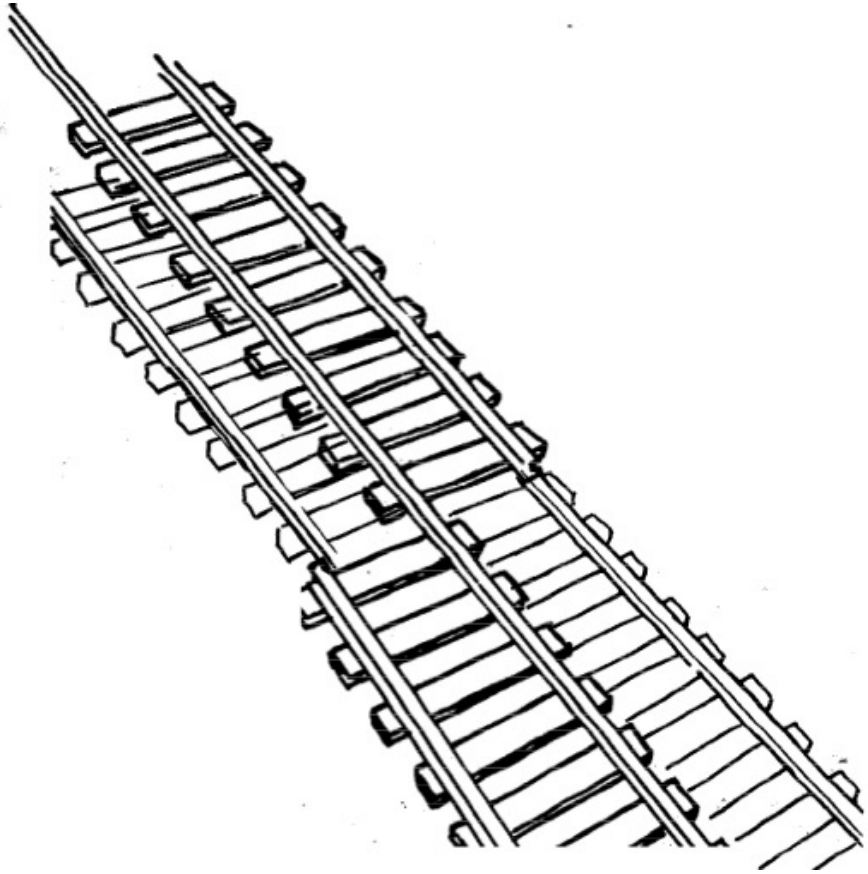
- Algorithmic Faults
  - Missing initialization
  - Incorrect branching condition
  - Missing test for null

- Errors
  - Null reference errors
  - Concurrency errors
  - Exceptions.

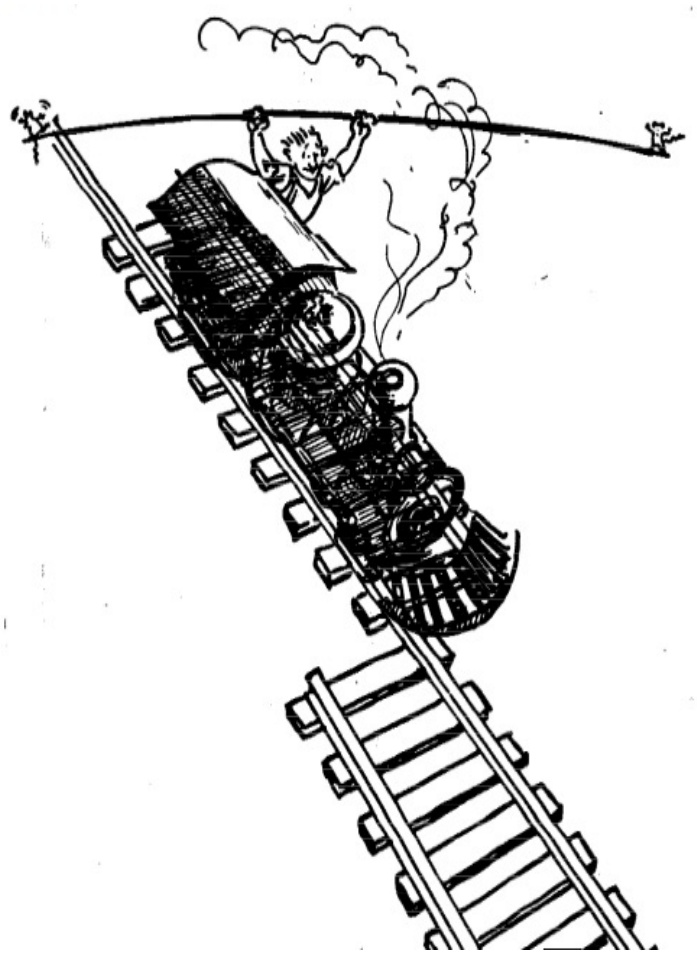
- Failure
  - Mismatch between requirements and implementation

How do we deal with Errors, Failures and Faults?

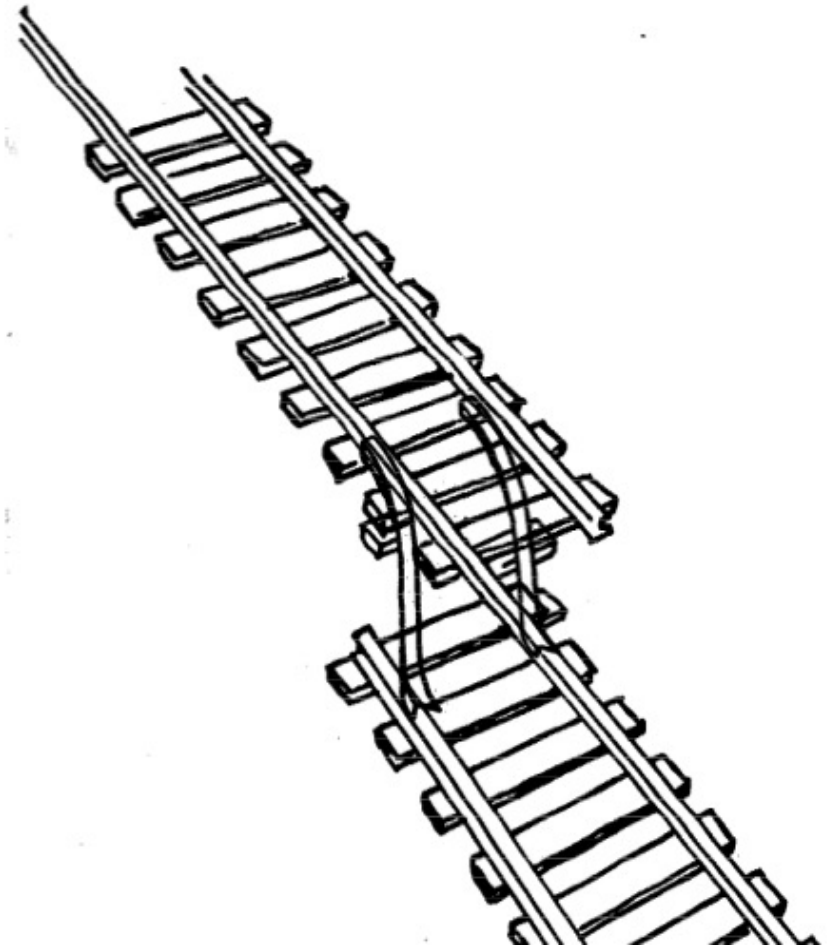
Modular Redundancy



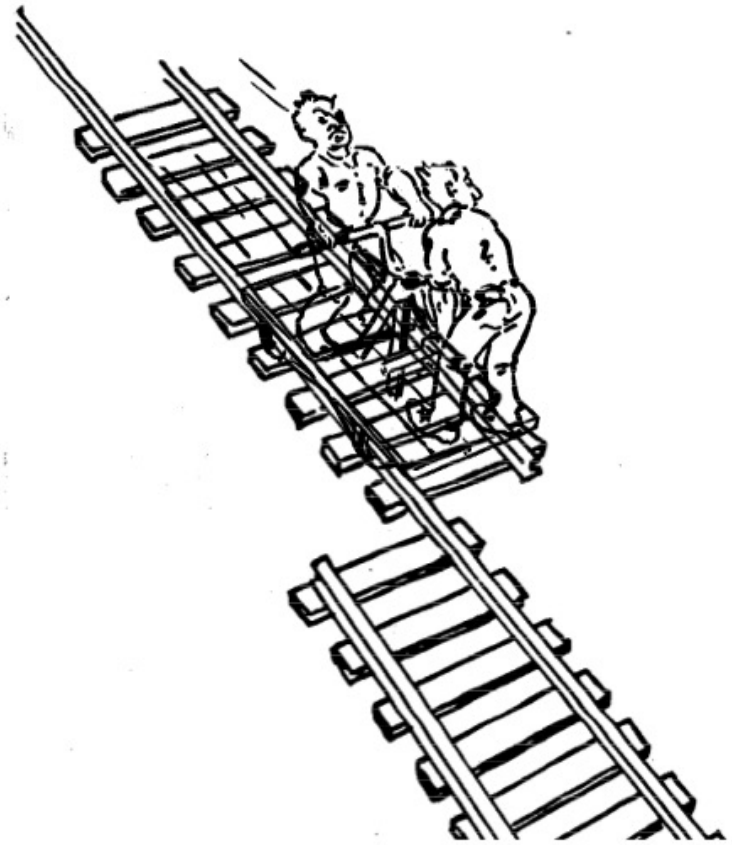
Declaring the Bug as a Feature



Patching



Testing



How to Deal with Faults?

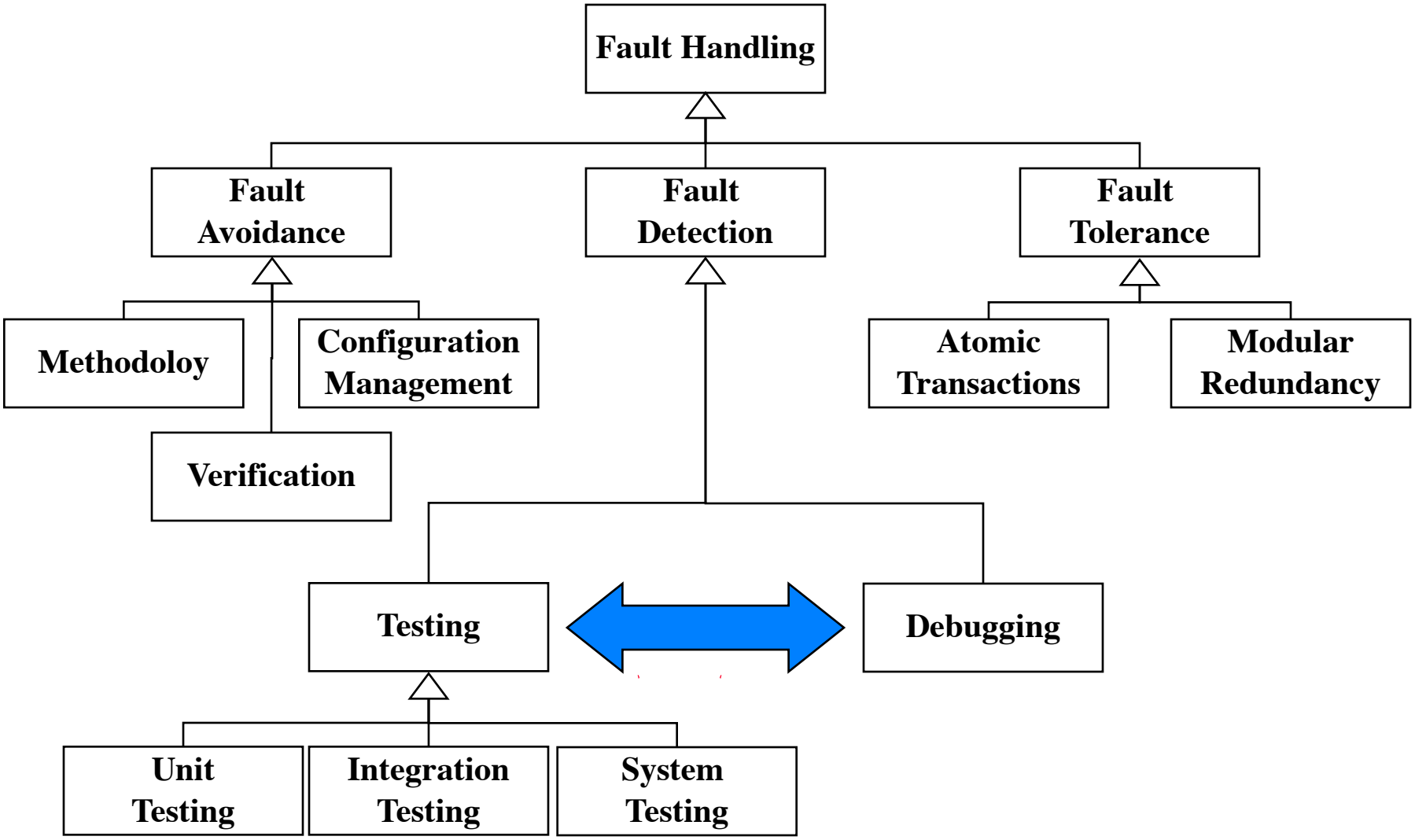
Any large system is bound to have faults.
What are some *quality control techniques* we can use to deal with them?

- **fault avoidance**: prevent errors by finding faults before the system is released
- **fault detection**: find existing faults without recovering from the errors
- **fault tolerance**: when system can recover from failure by itself

How to Deal with Faults?

- Fault avoidance
  - Use methodology to reduce complexity
  - Apply verification to prevent algorithmic faults
  - Use reviews
- Fault detection
  - Testing: Activity to provoke failures in a planned way
  - Debugging: Find and remove the cause (Faults) of an observed failure
  - Monitoring: Deliver information about state => Used during debugging
- Fault tolerance
  - Exception handling
    - Atomic transaction
  - Modular redundancy

Taxonomy for Fault Handling Techniques



Observations

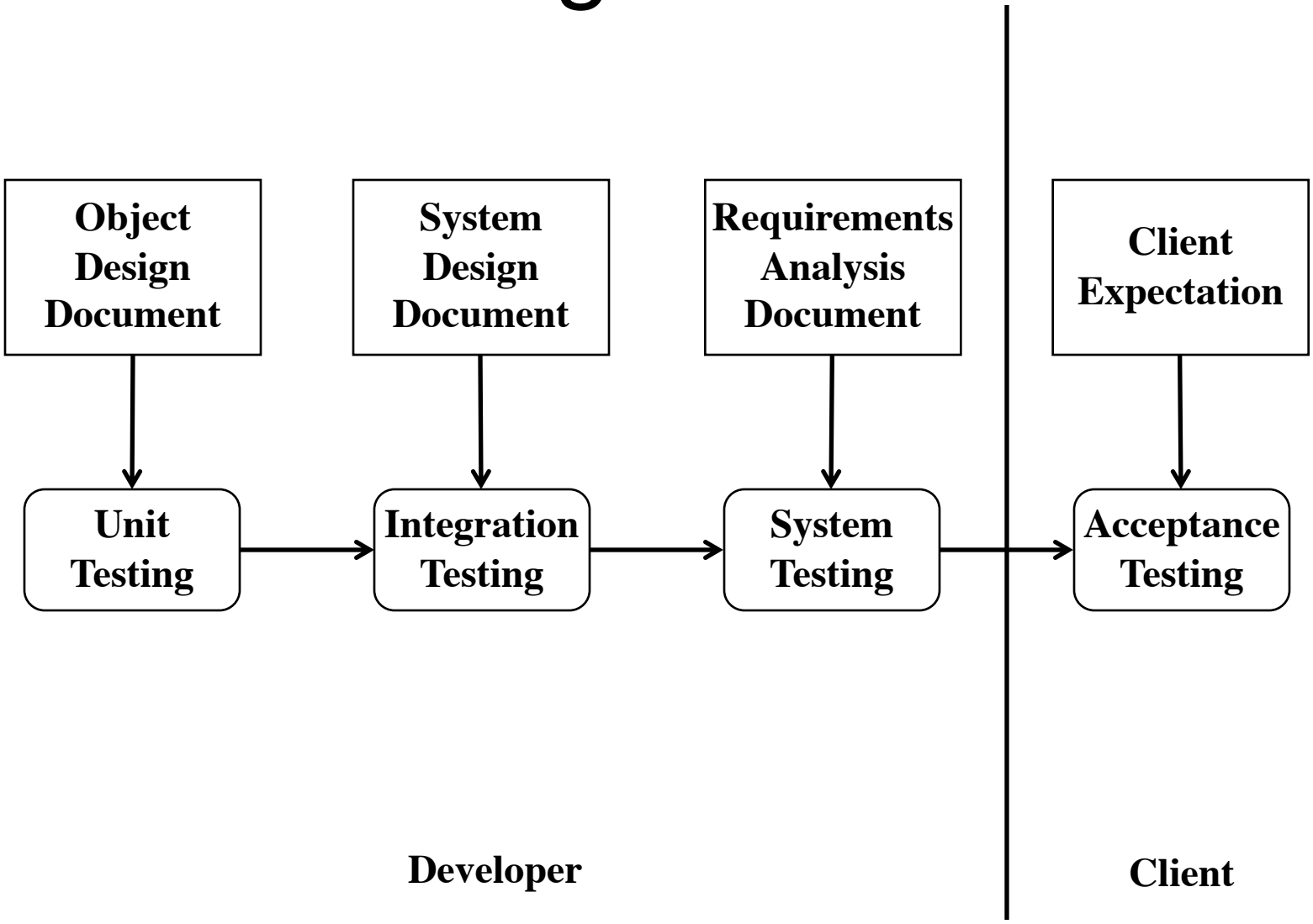
- It is impossible to completely test any nontrivial module or system
  - Practical limitations: Complete testing is prohibitive in time and cost
  - Theoretical limitations: e.g. Halting problem
- “Testing can only show the presence of bugs, not their absence” (Dijkstra).
- Testing is not for free

=> Define your goals and priorities

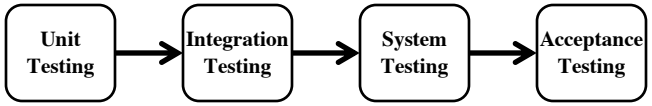
Testing takes creativity

- To develop an effective test, one must have:
  - Detailed understanding of the system
  - Application and solution domain knowledge
  - Knowledge of the testing techniques
  - Skill to apply these techniques
- Testing needs to be done by independent testers
  - We often develop a certain mental attitude that the program should in a certain way when in fact it does not
  - Programmers often stick to the data set that makes the program work
  - A program often does not work when tried by somebody else.

Testing Activities

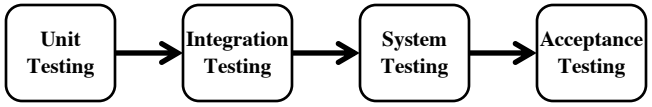


Types of Testing



- Unit Testing
  - Individual component (class or subsystem)
  - Carried out by developers
  - Goal: Confirm that the component or subsystem is correctly coded and carries out the intended functionality
- Integration Testing
  - Groups of subsystems (collection of subsystems) and eventually the entire system
  - Carried out by developers
  - Goal: Test the interfaces among the subsystems.

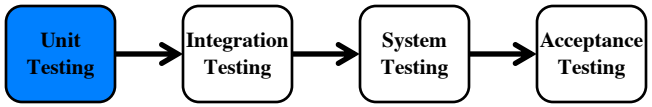
Types of Testing



- System Testing
  - The entire system
  - Carried out by developers
  - Goal: Determine if the system meets the requirements (functional and nonfunctional)
- Acceptance Testing
  - Evaluates the system delivered by developers
  - Carried out by the client.
  - Goal: Demonstrate that the system meets the requirements and is ready to use.

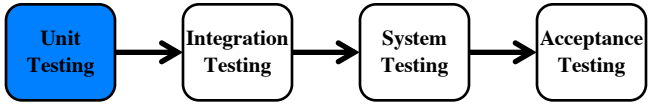
Unit Testing

- Static Testing (at compile time)
  - Static Analysis
  - Review
    - Walk-through (informal)
    - Code inspection (formal)



- Dynamic Testing (at run time)
  - Black-box testing
  - White-box testing.

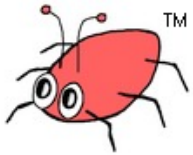
Static Analysis



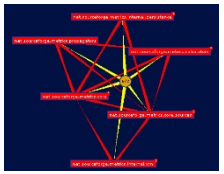
- Compiler Warnings and Errors
  - *Possibly uninitialized Variable*
  - *Undocumented empty block*
  - *Assignment has no effect*

- Checkstyle
  - Check for code guideline violations
  - http://checkstyle.sourceforge.net

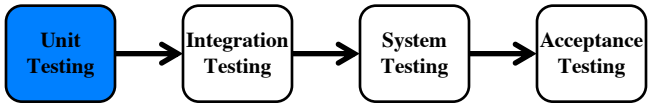
- FindBugs
  - Check for code anomalies
  - http://findbugs.sourceforge.net



- Metrics
  - Check for structural anomalies
  - http://metrics.sourceforge.net

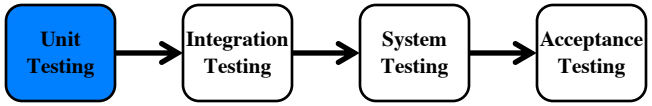


Black-box testing



- Focus: I/O behavior
  - If for any given input, we can predict the output, then the component passes the test
  - Requires test oracle
- Goal: Reduce number of test cases by equivalence partitioning:
  - Divide input conditions into equivalence classes
  - Choose test cases for each equivalence class

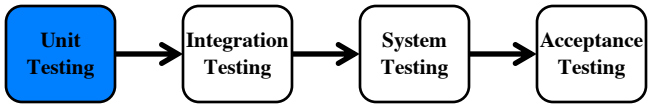
White-box testing overview



- Code coverage
- Branch coverage
- Condition coverage
- Path coverage

JUnit: Overview

- A Java framework for writing and running unit tests
  - Test cases and fixtures
  - Test suites
  - Test runner



- Written by Kent Beck and Erich Gamma
- Written with “test first” and pattern-based development in mind
  - Tests written before code
  - Facilitates refactoring
- JUnit is Open Source
  - www.junit.org

An example: Testing MyList

- Unit to be tested
  - MyList
- Methods under test
  - add()
  - remove()
  - contains()
  - size()
- Concrete Test case
  - MyListTestCase

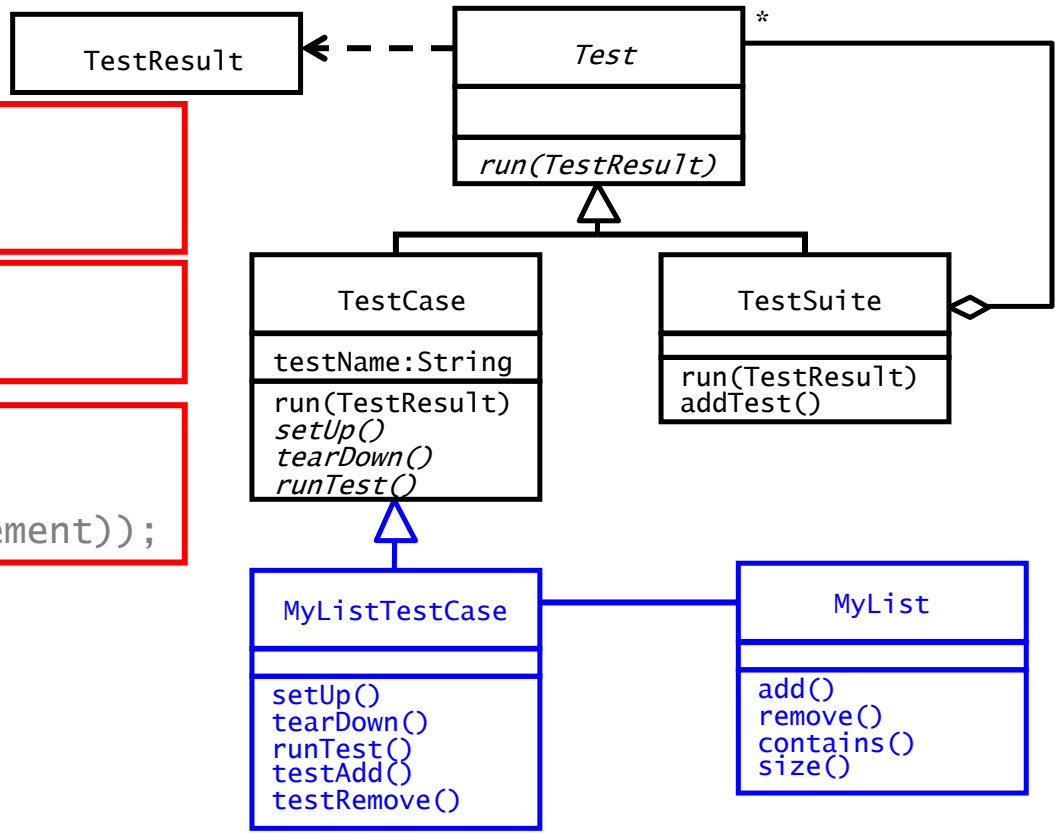
Writing TestCases in JUnit

```
public class MyListTestCase extends TestCase {

public MyListTestCase(String name) {
    super(name);
}
public void testAdd() {
    // Set up the test
    List aList = new MyList();
    String anElement = "a string";

    // Perform the test
    aList.add(anElement);

    // Check if test succeeded
    assertTrue(aList.size() == 1);
    assertTrue(aList.contains(anElement));
}
protected void runTest() {
    testAdd();
}
}
```



Writing Fixtures and Test Cases

```
public class MyListTestCase extends TestCase {
// …
private MyList aList;
private String anElement;
public void setUp() {
    aList = new MyList();
    anElement = “a string”;
}
```

Test Fixture

```
public void testAdd() {
    aList.add(anElement);
    assertTrue(aList.size() == 1);
    assertTrue(aList.contains(anElement));
}
```

Test Case

```
public void testRemove() {
    aList.add(anElement);
    aList.remove(anElement);
    assertTrue(aList.size() == 0);
    assertFalse(aList.contains(anElement));
}
```

Test Case

Collecting TestCases into TestSuites

```
public static Test suite() {
   TestSuite suite = new TestSuite();
   suite.addTest(new MyListTest("testAdd"));
   suite.addTest(new MyListTest("testRemove"));
   return suite;
}
```

Composite Pattern!