Test Automation Patterns: Best Practices and Common Pitfalls

Gerard Meszaros

StarEast2008@gerardm.com

Instructor Biography

Gerard Meszaros is a Calgary-based consultant specializing is agile development processes. Gerard built his first unit testing framework in 1996 and has been doing automated unit testing ever since. He is an expert in test automation patterns, refactoring of software and tests, and design for testability.

Gerard has applied automated unit and acceptance testing on projects ranging from full-on eXtreme Programming to traditional waterfall development and technologies ranging from Java, Smalltalk and Ruby to PLSQL stored procedures and SAP's ABAP.

He is the author of the Jolt Productivity Award winning book xUnit Test Patterns – Refactoring Test Code.

Gerard Meszaros
StarEast2008@gerardm.com

Teaching Method

Most adults learn best by doing

1. **Introduction to topic**
   - PowerPoint presentation describing principles, symptoms, root cause, possible solution patterns, etc.
   - Sample code or other concrete examples
   - Max 20 minutes
2. **Short Exercise**
   - Work in small groups
   - Given a list of symptoms
   - Propose root cause, solution
   - About 5-15 minutes
3. **Short Discussion**
   - Someone from each group provides one answer to one question
   - Round-robin so every group is heard (eventually)

Objectives of Tutorial

- **Understand why Test Smells are important**
- **Be able to recognize key Code Smells**
- **Be aware of test design patterns that can address or prevent these Code Smells**
- **Be able to recognize Behavior Smells and be aware of patterns to address them**
- **Be able to recognize Project Smells how they are related to Code and Behavior Smells**

Agenda

- **Introduction**
  - Metaphors
- **Test Strategy**
  - Why test?
  - Why Automate?
  - Which Automation Strategy?
- **Test Behavior Smells**
  - Patterns to address them
- **Test Code Smells**
  - Patterns to address them
- **Wrap Up**

Metaphors

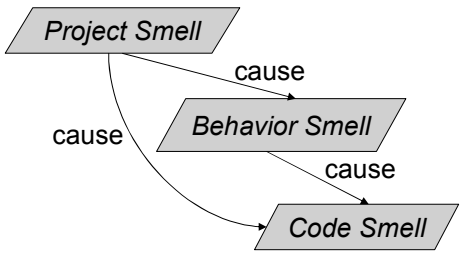
- **Why are metaphors useful?**
  - Gives us a way to understand new concepts based on prior experience
- **Three Metaphors:**
  - Smells
  - Debt
  - Patterns

What's a "Smell"?

- **A set of symptoms of an underlying problem in code**
- **Introduced by Martin Fowler in:**
  - Refactoring – Improving the Design of Existing Code
  - Term originally attributed to Kent Beck
- **The "Sniff Test":**
  - A smell should be obvious
  - It should "grab you by the nose"
- **Not necessarily the actual cause**
  - There may be many possible causes for the symptom
  - Some root causes may contribute to several different smells

What's a "Test Smell"?

- **Three common kinds of Test Smells:**
  - Code Smells – Visible Problems in Test Code
  - Behavior Smells – Tests Behaving Badly
  - Project Smells – Testing-related problems visible to a Project Manager
- **Code Smells may be root cause of Behavior and Project Smells**



What's a "Debt"?

- **Debt is generally considered to be bad**
- **Left alone, debt generally grows due to "Interest"**
  - Therefore, the more in debt you are,
  - the harder it is to get out
- **It takes a lot of effort to get out of debt**
  - Paying down the Interest on the debt vs paying down the capital (the debt itself)
  - The longer we leave it, the worse it gets

What's A Project Debt?

- **Anything we**
  - don't do enough of or do too late
  - that gets worse the longer we don't deal with it
- **Examples:**
  - Test enough (quantity)
  - Test early (timing)
  - Integrate software (e.g. Big Bang)
  - Test Automation (too manual)

Kinds of Test Debt

- **Test Debt**
  - Most projects are deep in Test Debt; they test too little, too infrequently and much too late to build quality in
- **Automation Debt**
  - Even more projects are in automation debt; they have very little in the way of automated testing and therefore have to spend a lot of effort doing manual testing.
- **Technical Test Debt**
  - Those projects that do have significant test automation are often deep in Automated Test Maintenance Debt because their tests require a lot of effort to keep current

What's a "Pattern"?

- **A "pattern" is a "recurring solution to a recurring problem"**
  - E.g. A "Decorator" object lets us add behavior to a system dynamically by adding one or more decorators to an existing object.
- **Must have been "invented" by three independent sources**
  - That's what makes it a "pattern" as in: "I see a pattern here!"
- **The pattern exists whether or not is has been written up in the "pattern form"**
  - Includes info on when (not) to use it

What's a "Test Pattern"?

- **A "test pattern" is a recurring solution to a test automation problem**
  - E.g. A "Mock Object" solves the problem of verifying the behavior of an object that should delegate behavior to other objects
- **Test Patterns occur at many levels:**
  - Test Automation Strategy Patterns
    - » **Recorded Test vs Scripted Test**
  - Test Design Patterns
    - » **Implicit SetUp vs Delegated SetUp**
  - Test Coding Patterns
    - » **Assertion Method, Creation Method**
  - Language-specific Test Coding Idioms
    - » **Expected Exception Test, Constructor Test**

Agenda

- **Introduction**
  - Metaphors
- **Test Strategy**
  - Why test?
  - Why Automate?
  - Which Automation Strategy?
- **Test Behavior Smells**
  - Patterns to address them
- **Test Code Smells**
  - Patterns to address them
- **Wrap Up**

What Could Possibly Go Wrong?

- **Software is an inherently detail-oriented activity**
  - Detailed data structures
  - Detailed algorithms
  - Detailed control structures
- **Building Software is all about communication**
  - What does the customer want (req'ts)
  - What does the developer plan to build (design docs)
  - How should the application behave? (test specs)
  - What did the customer really want (defect reports)

Lots, technically:

- **Out by one errors**
  - ">" instead of ">="
  - Array indexing errors
- **Missed cases**
- **Backwards logic**
  - "=" instead of "!=" (or ^= or <>)
  - ">" instead of "<"
- **Pure typos**
  - Misspelled variable names
  - "=" instead of "=="

Lots, Communication-wise:

- **Unspecified requirements**
- **Misunderstood requirements**
  - By developers
  - By testers
- **Misunderstood design**
- **Uncoded requirements**
- **etc.**

How Do We Keep it on the Rails?

- **We've developed a lot of techniques over the years:**
  - Requirements capture tools
  - Design modeling tools
  - Higher level programming languages
  - Syntax checking compilers
  - Static type checkers
- **While they all help …**

> … none of them is a substitute for actually trying the software!

"Trying" Software, the traditional way

- **Unit testing involves trying all the code**
  - After it is built
    - » **Sometimes called "debugging"**
- **Two basic strategies:**
  - Testing individual units before integration – Unit Testing
  - Testing units as part of larger whole – Integration Testing
- **Either approach can be done either**
  - Manually, by developers, testers or end users
  - Automated, by developers, testers or end users

Project Smell

Production Bugs

- **Symptoms:**
  - Bugs are being found in production
- **Impact:**
  - Expensive trouble-shooting
  - Development team's reputation is in jeopardy
- **Possible Causes:**
  - Lost/Missing Tests
  - Slow Tests
  - Untested Code
  - Hard-to-Test Code
  - Developers Not Unit Testing

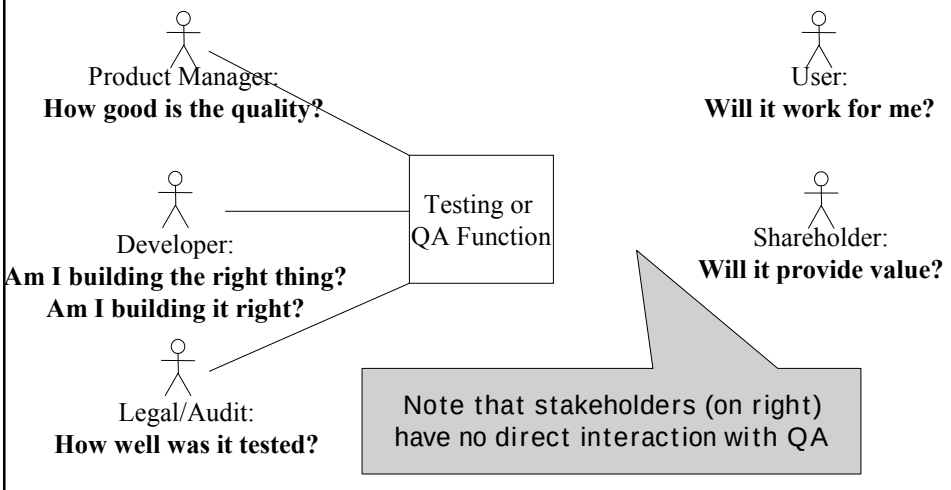
Purpose of Tests

Business Facing	Acceptance Tests	Usability Tests Exploratory Tests
Technology Facing	Unit Tests	Property Tests (Response Time, Security, Scalability)

Support Development **Critique Product**

Support Product Management

Testing as a Service:
Stakeholders vs. Customers



How Tests Support Product Management

- **Help ProdMgt understand quality**
  - Provide a "Scorecard" for the Product
  - Whenever they need it
    - » **Not just at Release Milestones**
    - » **Scorecard can improve visibility into status of development**
- **Help ProdMgt understand business impact of defects**
  - Risk = Probability * Consequence
  - Consequence must be expressed in business terms
    - » **Direct Support costs**
    - » **Lost opportunity cost (of people diverted to support)**
    - » **Cost of damage to reputation**

How Tests Support S/W Development

- **Before code is written**
  - Tests as Specification
    - » **What the code *should* do**
    - » **Clarification of the Requirements**
- **After code is written**
  - Tests as Documentation
    - » **What the code *does***
  - Tests as Safety Net
    - » **Provide Rapid feedback on quality (A.K.A. Bug Repellent)**
  - Defect Localization
    - » **Reduce cost to fix by Minimizing Debugging**

Agenda

- **Introduction**
  - Metaphors
- **Test Strategy**
  - Why test?
  - Why Automate?
  - Which Automation Strategy?
- **Test Behavior Smells**
  - Patterns to address them
- **Test Code Smells**
  - Patterns to address them
- **Wrap Up**

How Automated Tests Support Dev't

- **Minimize Cost of Running Tests**
  - Fully Automated Tests
  - Repeatable Tests
  - Robust Tests
- **Encourage Frequent Regression Testing**
  - By removing obstacles to running them frequently
  - Avoid inserting defects
  - Reduce cost of removing newly inserted defects

Both Prod. Management and Development want to know the score at all times

How Can QA Support Development?

- **Help Development understand requirements**
  - By clarifying them via concrete examples (tests)
- **Help Development assess quality**
  - As soon as possible after software is built
    - » **By testing the code immediately**
  - Whenever changes are made
    - » **By providing automated regression tests**

How Can Development Help QC Help Dev't?

- **By Building Software Incrementally**
  - In small, testable increments (User Stories)
- **By Making Designs Testable**
  - Include testability requirements in every User Story

The Problem with Manual Testing

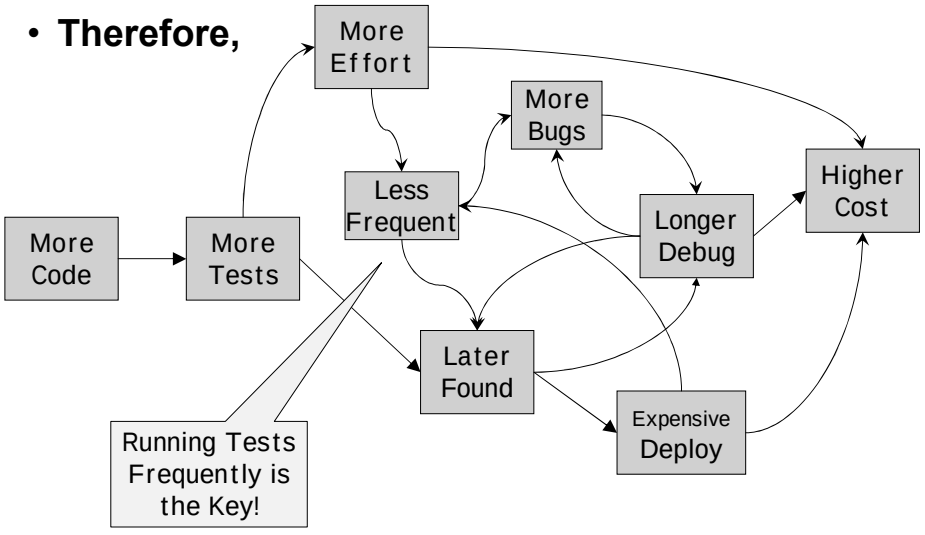
- **Not very repeatable**
  - Either cannot remember exactly how it was last tested,
  - Or, too expensive to document detailed procedures
- **Very effort-intensive**
  - How long will it take to rerun all the tests?
  - How often will you actually do it?
- **Doesn't solve the communication problem**
  - Tests typically prepared and executed once application is built

Automation Debt

Test Automation Debt

Testing & the Cost of Change

- **Therefore,**



Why Automate Tests?

- **In two words: Rapid Feedback!**
- **How do we know where are bugs are?**
  - **We still remember where we put them!**

Recap: Kinds of Test Debt

- **Test Debt**
  - Test too little, too infrequently and too late to prevent defects from being introduced
- **Test Automation Debt**
  - Relying primarily on manual testing to detect problems. Every test run takes a lot of time and effort.
- **Test Maintainability Debt**
  - Having automated tests that require a lot of effort to keep current

We want to avoid incurring the latter kinds of debt as we dig ourselves out of the former kinds of debt!

Agenda

- **Introduction**
  - Metaphors
- **Test Strategy**
  - Why test?
  - Why Automate?
  - Which Automation Strategy?
- **Test Behavior Smells**
  - Patterns to address them
- **Test Code Smells**
  - Patterns to address them
- **Wrap Up**

Project Smell

High Test Cost

- **Symptoms:**
  - It takes a lot of effort to test the system
  - There is a reluctance to test/deploy very often
- **Impact:**
  - Longer release cycles
  - Expanding feature backlog
- **Possible Causes:**
  - Test Debt
  - Test Automation Debt
  - Test Maintainability Debt

Project Smell

High Test Maintenance Cost

- **Symptoms:**
  - A lot of effort is going into maintaining the automated tests
  - Technical Test Debt
- **Impact:**
  - Cost of building functionality is increasing
  - People are agitating to abandon the automated tests
- **Possible Causes:**
  - Erratic Test
  - Fragile Test
  - Test Maintainability Debt
    - » **Wrong Test Automation Strategy**
  - –Buggy Automated Tests
  - –Obscure Test
  - –Hard to Test Code

Project Smell

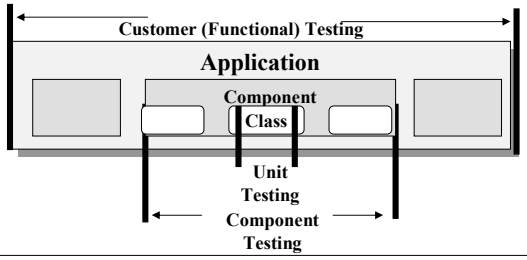
Buggy Automated Tests

- **Symptoms:**
  - Tests are failing when they shouldn't (the SUT works fine)
- **Impact:**
  - No one trusts the tests any more
- **Possible Causes:**
  - Erratic Tests
  - Fragile Tests
    - » **Both can be caused by using wrong test automation strategy**
  - Untested *Test* Code

Pattern

Layered Defence

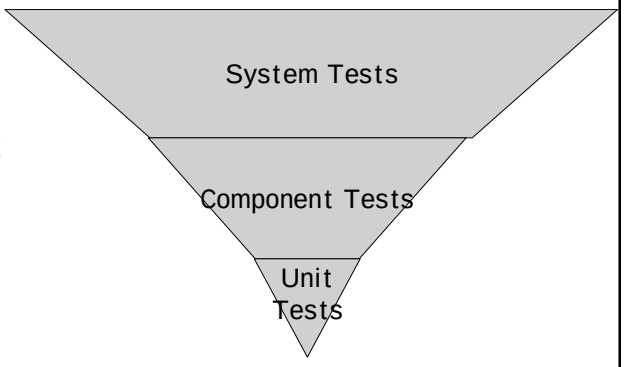
- **Functional Tests will tell us which features of the product doesn't work**
- **Component tests will tell us which component is at fault**
- **Unit tests tell us exactly which class/method is broken**



Unit Tests also let us test code we cannot hit in functional tests

Test Automation Pyramid

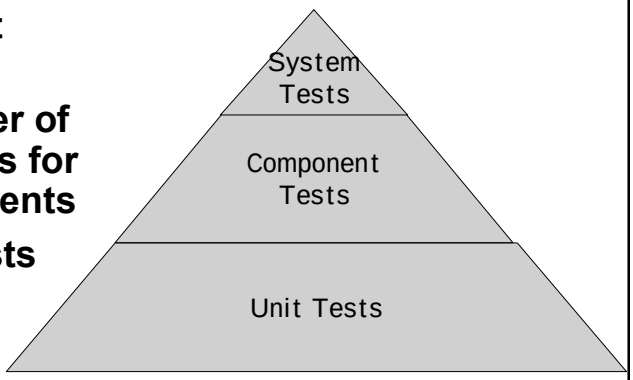
- **Large numbers of automated functional test**
- **Very few if any automated unit tests**



Typical when testing is "QA's job"

Test Automation Pyramid as Intended

- **Large numbers of very small unit tests**
- **Smaller number of functional tests for major components**
- **Even fewer tests for the entire application & workflow**



Automated System Tests

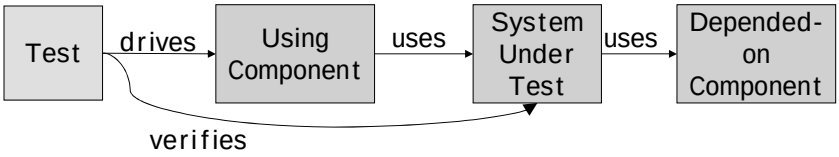
- **Defines what "Done looks like"**
- **Executible, self-verifying examples**
- **Verifies that units deliver useful functionality**

Automated Unit Tests

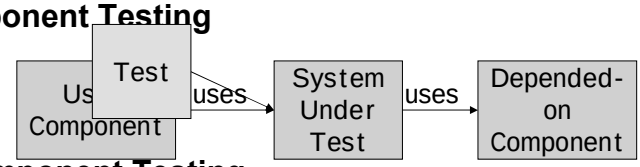
- **The equivalent of Lean Production's "In-process Inspection"**
- **"Inspection for defects is waste; inspection to prevent defects is essential."**
  - » **Shigeo Shingo**
  - – If type-checking compiler is good, behavior-checking compiler must be better. Automated unit tests!
- **Verify that individual units are built correctly.**
- **Does *not* verify that units deliver useful functionality**

Automated Component Testing

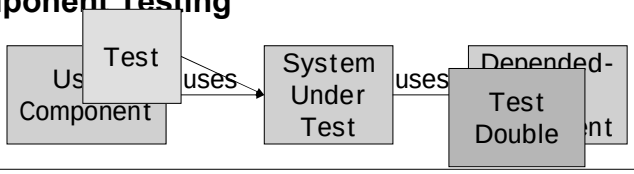
- **Indirect Component Testing**



- **Direct Component Testing**



- **Isolated Component Testing**



Project Smell

Developers Not Unit Testing

- **Symptoms:**
  - No tests can be found when you ask to see the
    - » **unit tests for a task,**
    - » **customer tests for a User Story,**
  - Lack of clarity about what a feature, user story or task really means
- **Impact:**
  - Lack of safety net
  - Lack of focus
- **Possible Causes:**
  - Hard to Test Code?
  - Not enough time?
  - Don't have the skills?
  - Have been told not to?
  - Don't see the value?

Recap of Common Project Smells

- **Production Bugs**
- **High Test Cost**
- **High Test Maintenance Cost**
- **Buggy Tests**
- **Developers Not Writing Tests**

Exercise – PS1

- **Symptoms:**
  - Our project adopted automated testing over a year ago. We now have a large number of automated tests. Every week we spend several person-days trouble-shooting broken tests. More often than not, the production code is found to be correct. At the same time, beta users are reporting a fair number of problems.
- **Discussion Questions:**
  - What forms of debt do we have?
  - Which Project Smells are we having?
  - What questions do we need to ask to find out why they are occurring?
  - What can we do about them?

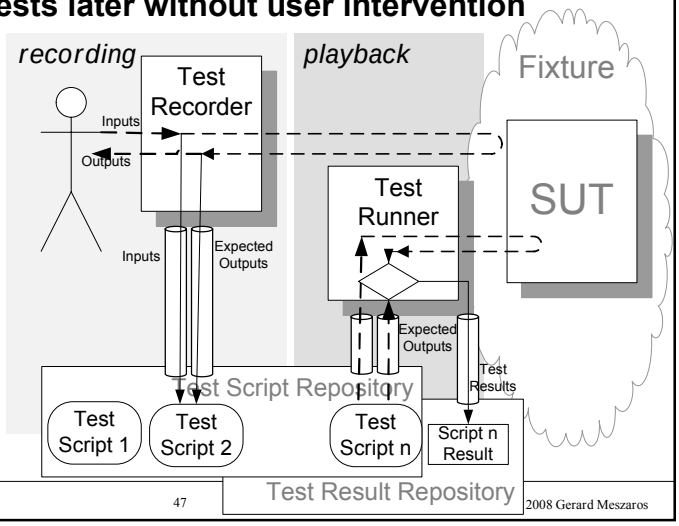
Agenda

- **Introduction**
  - Metaphors
- **Test Strategy**
  - Why test?
  - Why Automate?
  - Which Automation Strategy? Continued!
- **Test Behavior Smells**
  - Patterns to address them
- **Test Code Smells**
  - Patterns to address them
- **Wrap Up**

Recorded Tests

- **User executes tests manually; tool records as tests**
- **Tool replays tests later without user intervention**

The tests are data interpreted by the test tool.



Common Pitfalls of Recorded Tests

- **The tests cannot be rerun reliably (Erratic Tests)**
  - Synchronization problems with the UI
  - Differences in system/database state
  - Differences in how system behaves at different times
- **The tests cannot be easily understood (Obscure Tests)**
  - because they are focussed on UI details
- **The tests are very sensitive to changes in the application (Fragile Tests)**
  - They need to be rerecorded very often
- **The tests only verify a subset of what the tester verified**
  - Tester has a brain and eyes that look for things
  - Adding ***checkpoints*** increases cost/complexity of recording tests

Recorded Tests Can Work Well …

… in the right circumstances:

- **Record, Refactor, Playback**
- **Built-In Test Recording**

More often than not, however,

Scripted Tests are a better solution

Built-In Test Recording

A Satisfied Customer Says:

"… Not only did this save 10's of man-years of testing effort, but it even uncovered before unknown bugs in the legacy system which we considered to be the gold standard. …" Peter Arato, CPR on LinkedIn

How:

- We retrofitted the capability to record and playback right into the application.
- Test recording vocabulary was domain-specific,
  - » **not generic UI interactions**

Requires Work By Development;
Must be Made Priority by Prod Mgnt

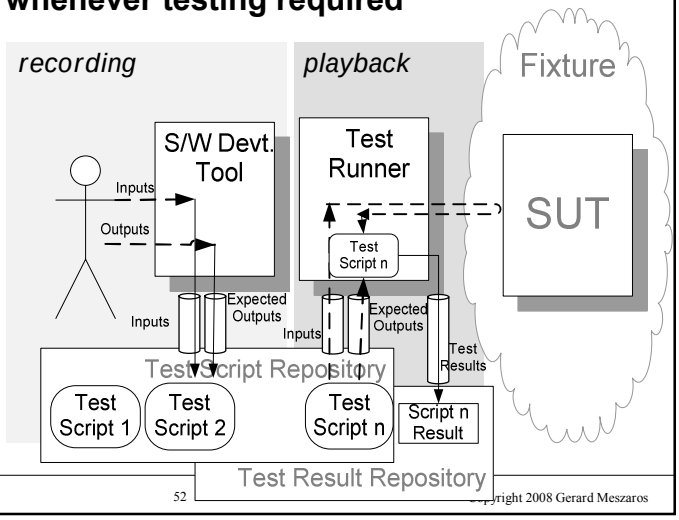
Record, Refactor, Playback

- **Use Test Recording as a way to capture tests**
- **Replace details with calls to domain-specific Test Utility Methods**
  - using Extract Method refactorings
- **Make Test Utility Methods reusable**
  - Replace Hard-Coded Literal Values with variables/parameters

Scripted Tests

- **Tester writes test code to exercise the software**
- **Test code run whenever testing required**

The tests are a program that tests the program



Scripted Tests

- **Developers or testers write test scripts to verify behavior of system**
- **Can be applied at Unit or System level**
- **Built using development language**
  - xUnit (JUnit, NUnit, CppUnit, PyUnit, etc.)
- **Built in dynamic scripting language**
  - Watir framework for Ruby
  - IeUnit framework for VbScript
  - A host of others
- **Can even be done w/commercial R&PB tools**
  - e.g. QuickTest

Common Pitfalls of Scripted Tests

- **More work (initially) than Recorded Tests**
- **May also suffer from Test Smells if not designed well:**
  - Obscure Tests are hard to understand and maintain
  - Erratic Tests have different results every time they are run
  - Fragile Tests fail unnecessarily after changes to SUT

Overall, a more maintainable strategy than Recorded Test

Pros & Cons of Scripted Tests

Pros

- **Not limited to UI-based testing**
  - **Can test via any programming API**
- **A more maintainable strategy than Recorded Test**
  - **Easier to understand**
  - **Less duplication**
  - **Less fragile**

Cons

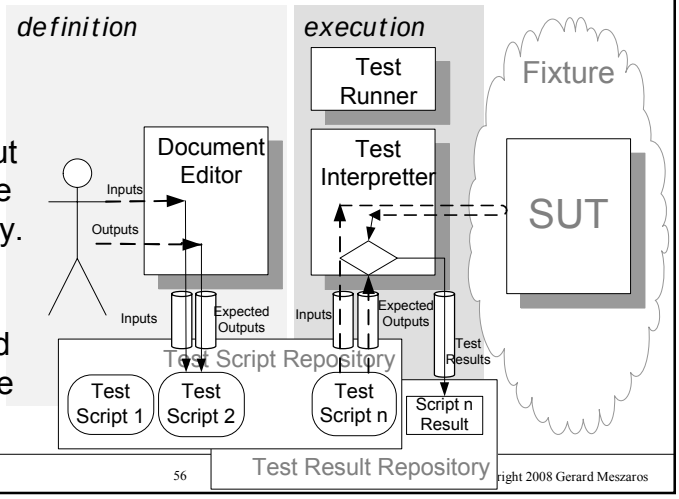
- **More work (initially) than Recorded Tests**
- **May also suffer from Test Smells if not designed well:**
  - **Obscure Tests**
  - **Erratic Tests**
  - **Fragile Tests**

Data-Driven Tests

- **The tests are expressed as tabular data by users.**
- **The tests are read & executed by a test interpreter written by techies.**

Prepared like Scripted Tests but with a much more limited vocabulary.

"Keyword"-based testing is just one example.



Pros & Cons of Data-Driven Tests

Pros

- **Easier for business people to understand**
- **Can be written by business people**

Cons

- **Need to build interpreter**
  - For each keyword
- **Less general than either Scripted or Recorded**

Agenda

- **Introduction**
  - Metaphors
- **Test Strategy**
  - Why test?
  - Why Automate?
  - Which Automation Strategy?
- **Test Behavior Smells**
  - Patterns to address them
- **Test Code Smells**
  - Patterns to address them
- **Wrap Up**

A Sobering Thought

Plan on having as much test code as production code!

The Challenge: How To Prevent Doubling Cost of Software Maintenance?

Why is Test Maintainability so Crucial?

- **Tests need to be maintained along with rest of the software.**
- **Testware must be much easier to maintain than the software, otherwise:**
  - It will slow you down
  - It will get left behind
  - Value drops to zero
  - You'll go back to manual testing

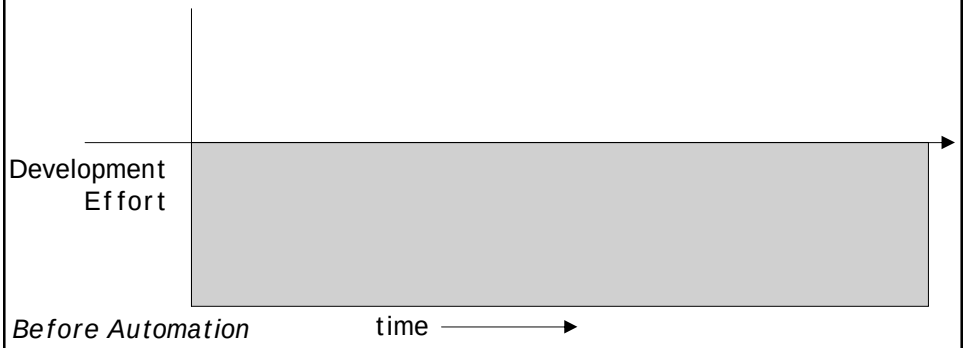
Critical Success Factor:

Writing tests in a maintainable style

Economics of Maintainability

Test Automation is a lot easier to sell on

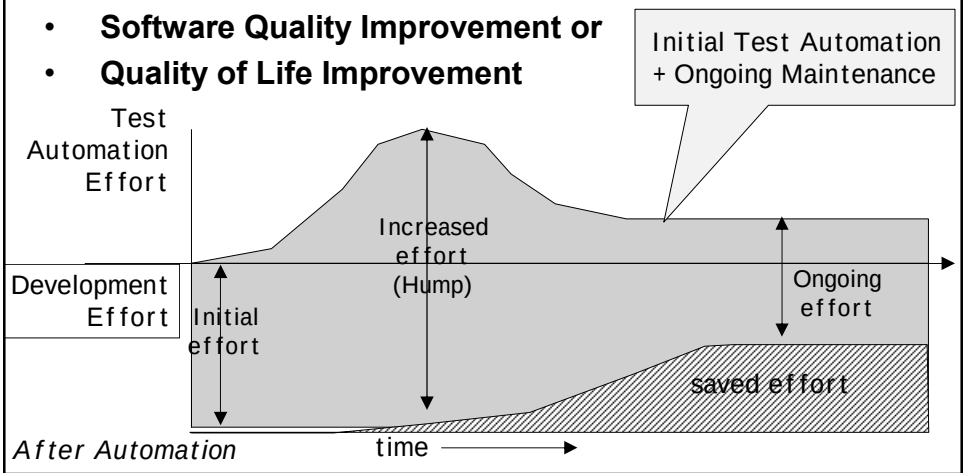
- **Cost reduction than**
- **Software Quality Improvement or**
- **Quality of Life Improvement**



Economics of Maintainability

Test Automation is a lot easier to sell on

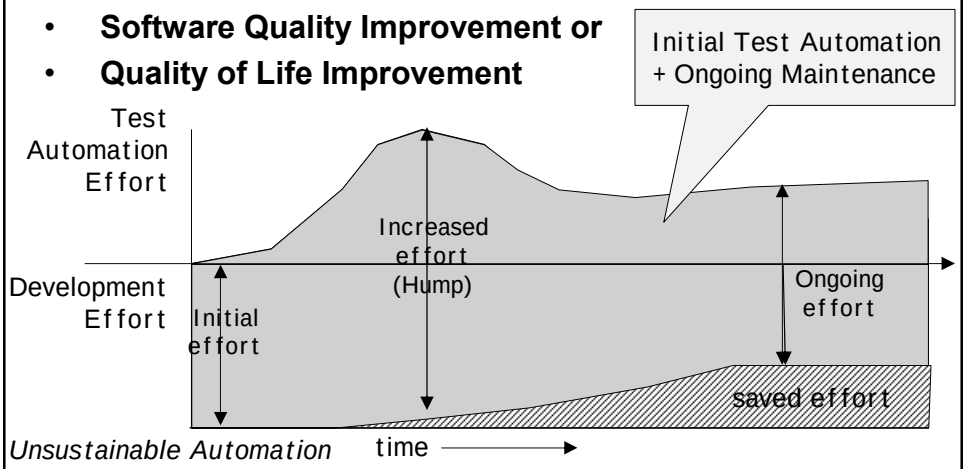
- **Cost reduction than**
- **Software Quality Improvement or**
- **Quality of Life Improvement**



Economics of Maintainability

Test Automation is a lot easier to sell on

- **Cost reduction than**
- **Software Quality Improvement or**
- **Quality of Life Improvement**



Sources of Savings

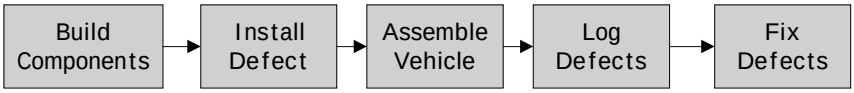
- **Preventing Defects rather than Finding/Fixing them**
  - Less debugging
  - Less rework

- **Need to find ways to measure "Defects Prevented"**
  - Possibly implied from "Defects Logged By Customer"
    - » **Historical – Current = Prevented**

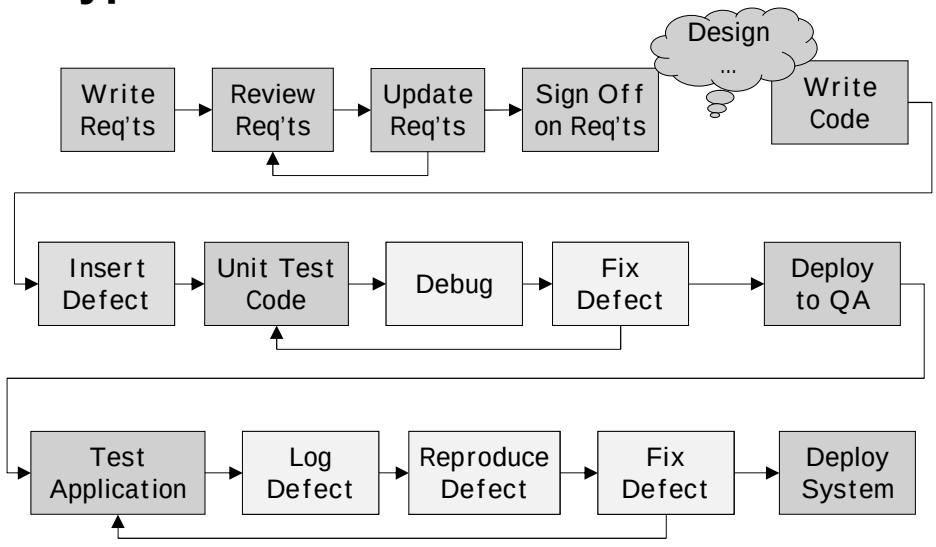
Understanding Behaviour - Value Stream Mapping

- **A way to understand all the work required to produce a product or deliver a service**
- **Often used to identify waste in the delivery process**
- **Example: "Waterfall" Software Development**

- **Example: American Car Assembly Line**



Typical "Waterfall" S/W Value Stream



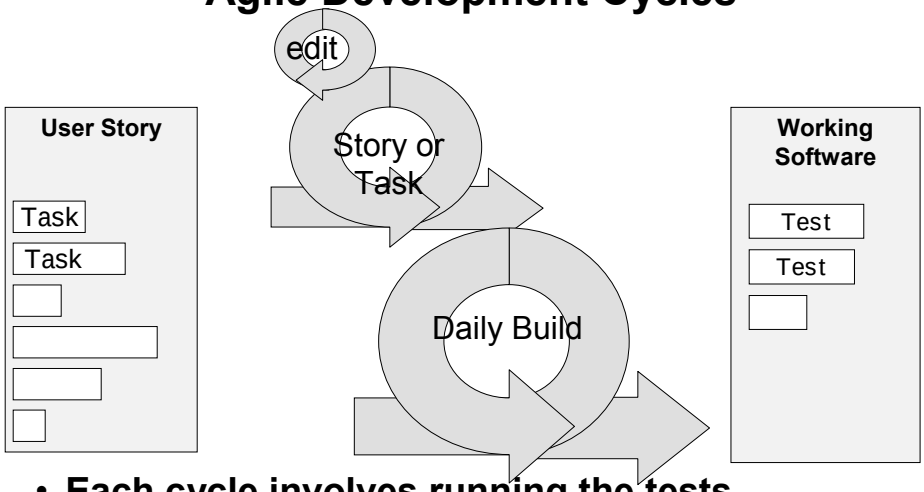
What's a Behavior Smell

- **A problem seen when running tests.**
- **Tests fail when they should pass**
  - or pass when they should fail (rarer)
- **The problem is with how tests are coded;**
  - not a problem in the SUT
- **Sniff Test:**
  - Detectable via compile or execution behavior of tests

Common Behavior Smells

- **Slow Tests**
- **Erratic Tests**
  - Too many variants to list here
- **Fragile Tests**
  - The 4 sensitivities
- **Assertion Roulette**
- **Frequent Debugging**
- **Manual Intervention**

Agile Development Cycles



- **Each cycle involves running the tests**
- **The tests must run "quickly enough"**

Behavior Smell

Slow Tests

- **Slow Tests**
  - It takes several minutes to hours to run all the tests
- **Impact**
  - Lost productivity caused by waiting for tests
  - Lost quality due to running tests less frequently
- **Causes:**
  - Slow Component Usage
    - » **e.g. Database**
  - Asynchronous Test
    - » **e.g. Delays or Waits**
  - General Fixture
    - » **too much fixture being setup**

Exercise: BS1 - Avoiding Slow Tests

- ***Symptoms:***
  - *Your tests are taking 10 minutes to run. Everyone is getting frustrated because it is taking an hour to get the "commit token".*
- ***Instructions:***
  - *Brainstorm at least 5 ways to make the tests run faster.*
- ***Discussion Questions:***
  - What might be the root causes of slow test?
  - What can we do to address each possible cause?

Avoiding Slow Tests – Slow SUT

- **Run Tests Faster**
  - Get faster hardware
    - » **E.g. Quad-processor test execution box**
- **Avoid Slow Code**
  - Avoid Fixture Persistence
    - » **Use a Fresh Fixture with Fake Database**
  - Avoid slow components
    - » **Replace with Test Double (or Test Stub)**
- **Run Fewer Tests**
  - Run subsets of tests when possible (e.g. pre-checkin)
  - Run all the tests sometime, somewhere! (e.g. overnight)

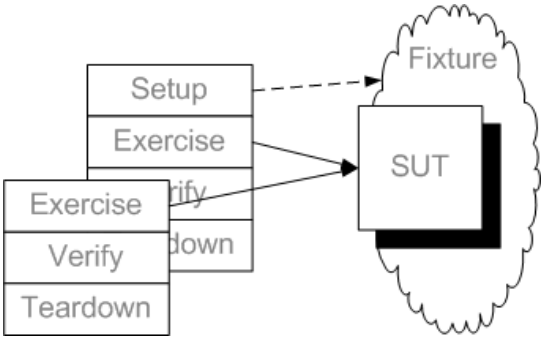
Avoiding Slow Tests – Slow Test Code

- **Avoid Waits**
  - Use Humble Object to avoid Asynchronous Test by testing logic directly
- **Test Less Code**
  - Reduce Test Overlap
- **Set Up Less Fixture**
  - Use a Minimal Fixture
- **Set Up Fixture Less Often**
  - Reuse a Shared Fixture

Pattern

Shared Test Fixture

- **What it is:**
  - Improves test run times by reducing setup overhead.
  - A "standard" test environment applicable to all tests is built and the tests reuse the same fixture instance.



Shared Test Fixture

- **Variations:**
  - Fixture is shared between some/all the tests in a single test run
  - Fixture may be shared across many TestRunners (Global Text Fixture)
- **Examples:**
  - Standard Database contents
  - Standard Set of Directories and Files
  - Standard set of objects

Bad Smell Alert:
•Erratic Tests

Setting Up the Shared Test Fixture

To share the same fixture *instance* between tests:

- **Prebuilt Fixture** — Unrepeatable Tests
  - Fixture is built ahead of time and reused by many test runs
- **Lazy Setup** — Use only when don't need to clean up the fixture
  - First reference causes it to be initialized
  - How do you know when to clean up?
- **SuiteFixture Setup** — Only supported by NUnit, VbUnit, JUnit 4.0
  - Use Static variables to hold the fixture
  - Initialize one before first test; destroy after last
- **Setup Decorator** — Tests that depend on the decorator cannot be run without it.
  - Define a Test Decorator that implements Test
  - Wrap the test suite with an instance of the decorator

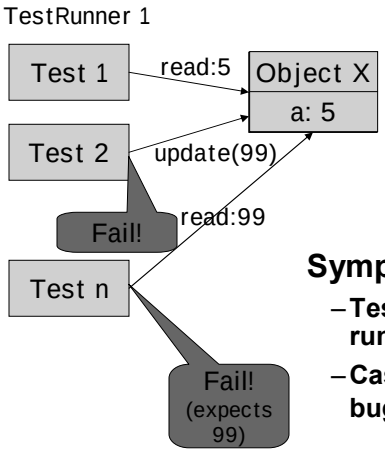
Behavior Smell

Erratic Tests

- **Interacting Tests**
  - When one test fails, a bunch of other tests fail for no apparent reason because they depend on other tests' side effects
- **Unrepeatable Tests**
  - Tests can't be run repeatedly without intervention
- **Test Run War**
  - Seemingly random, transient test failures
  - Only occurs when several people testing simultaneously
- **Resource Optimism**
  - Tests depend on something in the environment that isn't avaiable
- **Non-Deterministic Tests**
  - Tests depend on non-deterministic inputs

Behavior Smell

Erratic Tests – Interacting Tests



If many tests use same objects, tests can affect each other's results.

– Test 2 failure may leave Object X in state that causes Test n to fail.

Symptoms:

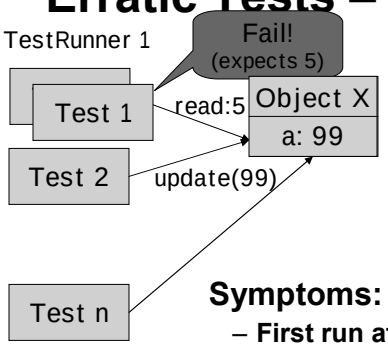
– **Tests that work by themselves fail when run in a suite.**

– **Cascading errors caused by a single bug failing a single test.**

» **Bug need not affect other tests directly but leaves fixture in wrong state for subsequent tests to succeed.**

Behavior Smell

Erratic Tests – Unrepeatable Tests



If many test runs use same objects, test runs can affect each other's results.

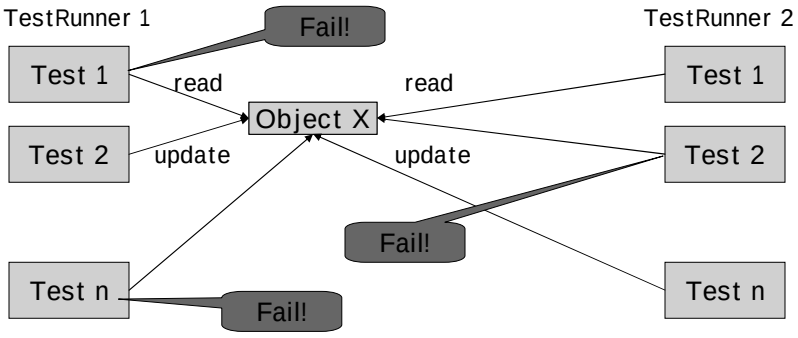
- Test 2 update may leave Object X in state that causes Test 1 to fail on next run.

Symptoms:

- **First run after opening the TestRunner or re-initializing Shared Fixture behaves differently**
  - » **Succeed, Fail, Fail, Fail**
  - » **Fail, Succeed, Succeed, Succeed**
- **Resetting the fixture may "reset" things to square 1 (restarting the cycle)**
  - » **Closing and reopening the test runner for in-memory fixture**
  - » **Reinitializing the database**

Behavior Smell

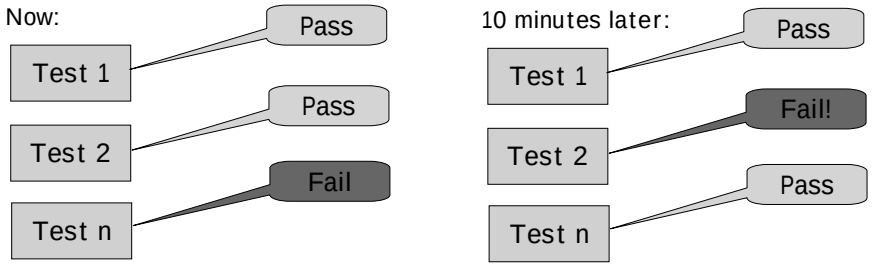
Erratic Tests – Test Run War



- **If many test runners use the same objects (from Global Fixture), random results can occur.**
  - Interleaving of tests from parallel runners makes determining cause very difficult

Behavior Smell

Erratic Tests – Non Deterministic Test



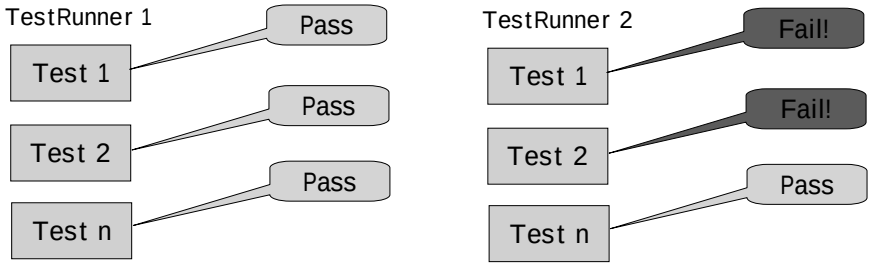
Tests depend on non-deterministic inputs.

Symptoms:

- **Tests pass at some times; fail at other times**
  - Lack of control over time/date when system contains time/date logic (addressed by getting control of indirect input via a stub)
  - Tests use different values in different runs

Behavior Smell

Erratic Tests – Resource Optimism



Tests depend on non-ubiquitous external resources.

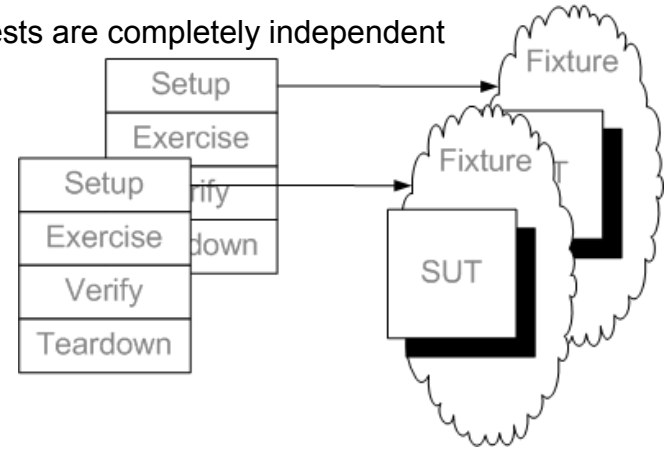
Symptoms:

- **Tests pass in some environments; fail in others**
  - SUT depends on something in the environment that is not always present.
  - Addressed by creating it during the fixture setup phase

Pattern

Avoiding Erratic Tests - Fresh Fixture

- **What it is:**
  - "Brand new" fixture built for each test
  - Tests are completely independent



Pattern

Fresh Fixture

- **Variations:**
  - Transient Fresh Fixture
    - » **Fixture automatically disappears at end of each test**
    - » **e.g. Garbage-collected TearDown**
  - Persistent Fresh Fixture
    - » **Fixture naturally "hangs around" after test**
    - » **Requires extra effort to ensure it is fresh**

Pattern

Persistent Fresh Fixture

Two Options:

1. **Rebuild fixture for each test and tear it down**
   - When
     - » **At end of this test (just in case)**
     - » **At start of next test that uses it (just in time)**
   - How
     - » **Hand-coded Tear Down**
     - » **Automated Tear Down**
2. **Build different fixture for each test**
   - Use a Distinct Generated Value for any unique Id's
   - Makes tear down necessary

Reducing Erratic Tests - Shared Fixture

- **Avoid Interactions between Test Runners**
  - Give each developer their own Database Sandbox.
    - » Avoids Test Run Wars but not Interacting Tests, etc,
- **Don't Change Shared Fixture**
  - Immutable Shared Fixture avoids Interacting Tests
  - Create Fresh Fixture for objects to be changed
    - » (See Persistent Fresh Fixture)
  - Challenge: What constitutes a "change" to a fixture?
    - » Change existing objects / rows -> YES!
    - » Add new objects related to existing objects -> SOMETIMES!

Reducing Erratic Tests - Shared Fixture

- **Build new Shared Fixture for each run**
  - Avoids Unrepeatable Tests
  - When:
    - » Lazy Setup
    - » Setup Decorator
    - » SuiteFixture Setup

Exercise: BS2 – Test Debugging

- ***Symptoms:***
  - *Earlier today, you ran all the tests after making some code changes and the tests ran green. You then went to lunch. When you came back you re-ran the tests "just to make sure" before committing your changes. Now, several tests are failing.*
- ***Instructions:***
  - Diagnose the problem using only the console output.
- ***Discussion Questions:***
  - Which Behavior Smells are we having?
  - What questions do we need to ask to find out why they are occurring?
  - What are the underlying root causes?
  - What can we do about them?

Behavior Smell

Fragile Tests

Causes:

- **Interface Sensitivity**
  - Every time you change the SUT, tests won't compile or start failing
  - You need to modify lots of tests to get things "Green" again
  - Greatly increases the cost of maintaining the system
- **Behavior Sensitivity**
  - Behavior of the SUT changes but it should not affect test outcome
  - Caused by being dependent on too much of the SUT's behavior.

Behavior Smell

Fragile Tests (2)

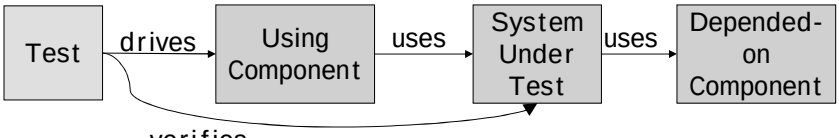
Causes (continued):

- **Data Sensitivity**
  - Aliase: Fragile Fixture
  - Tests start failing when a shared fixture is modified
    - » **e.g. New records are put into the database**
- **Context Sensitivity**
  - Something outside the SUT changes
    - » **e.g. System time/date, contents of another application**

Avoiding Interface Sensitivity

- **Use Stable Interfaces**
  - Bypass Presentation Layer (UI)
  - Backwards compatibility of changes to used interface
    - » **e.g. Facade**
- **SUT API Encapsulation**
  - Hide non-essential parts of SUT API from Test Methods via
    - » **Creation Method**
    - » **Finder Method**
    - » **Verification Method**

Testing Components as part of Product Stack



Advantages:

- **Doesn't require understand component usage patterns**
- **Doesn't require extra automation effort**

Disadvantages:

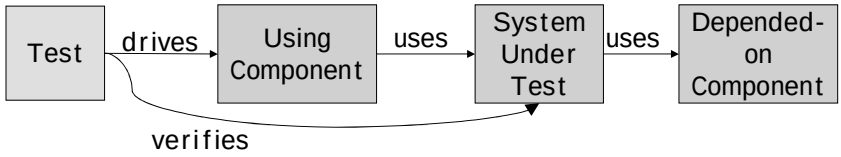
- **Cannot test components independently**
  - Need using components to drive it
  - Need depended-on components to support it.
- **Cannot exercise some scenarios**
  - Indirect testing via users cannot force some scenarios
  - Indirect inputs may not be controllable

Implication:
Cannot tell which component is at fault when Application Test fails.

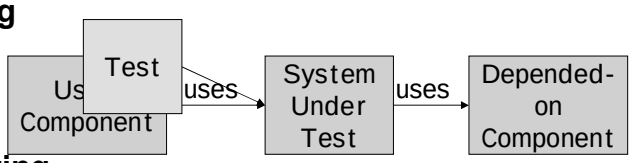
Implication:
Don't have an accurate assessment of quality.

Better Component Testing

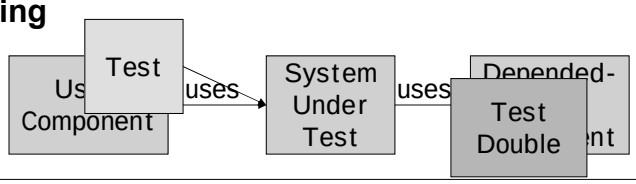
- **Indirect Testing**



- **Direct Testing**



- **Isolated Testing**



Test Double Patterns

- **Replace depended-on components with test-specific ones to isolate SUT**
- **Kinds of Test Doubles**
  - Test Stubs return test-specific values
  - Test Spies record method calls and arguments for verification by Test Method
  - Mock Objects verify the method calls and arguments themselves
  - Fake Objects provide (apparently) same services in a "lighter" way

Test Double Patterns (more)

- **Test Doubles need to be "installed"**
  - Dependency Injection
  - Dependency Lookup
- **… therefore, Design for Testability is key.**
- **Configurable Test Doubles are reusable but need to be configure with test-specific values**
  - return values
  - expected method calls & arguments

Development & Test Need to Collaborate with Each Other to Do Effective Component Testing

Avoiding Data/Context Sensitivity

- **Minimal Fresh Fixture**
  - Use a Fresh Fixture
  - Custom design it for each test.
  - Avoid a Standard Fixture that could become a Fragile Fixture
- **Test Stubs**
  - Replace the need for real fixture by using a Test Stub to provide indirect inputs

Exercise: BS3 – Build Debugging

- ***Symptoms:***
  - *Last week, all the tests ran clean. Since then, 10 new tests have been added (green) but several existing tests are failing.*
- ***Instructions:***
  - Diagnose the problem using only the console output.
- ***Discussion Questions:***
  - Which Behavior Smells are we having?
  - What questions do we need to ask to find out why they are occurring?
  - What are the underlying root causes?
  - What can we do about them?

Behavior Smell

Assertion Roulette

- **Symptom:**
  - One or more unit tests are failing in the automated build and you cannot tell why without rerunning the tests in your IDE. When you cannot reproduce the problem in your IDE you have no idea what is going wrong.
- **Impact:**
  - It takes longer to determine what is wrong with the code.
  - Bugs that cannot be reproduced cannot be fixed.
- **Root Cause:**
  - Missing/Unclear Assertion Messages
- **Solution:**
  - Use the right Assertion Method.
  - Add Assertion Messages to all Assertion Method calls
  - Write Diagnostic Custom Assertion

Pattern

Diagnostic Custom Assertion

- **Variation of Custom Assertion**
- **Compares its inputs in a way that provides useful diagnostic messages.**
- **e.g. assertEquals does this:**
  - expected <nil> but was <abc>
  - strings differ starting at position 247; expected <..abcdefghi..> but was <...abcxyzghi..>

Behavior Smell

Frequent Debugging

- **Symptom:**
  - One or more tests are failing and you cannot tell why without resorting to the debugger. This seems to be happening a lot lately!
- **Impact:**
  - Debugging is a very time-intensive activity.
  - While it may help you find the bug, it won't keep it from coming back.
- **Root Causes:**
  - Missing Unit Tests
  - Poor Assertion Messages
- **Solution:**
  - Better unit test coverage of the code
  - More/Better Assertion Messages

Exercise: BS4 – Build Debugging

- ***Symptoms***:
  - *You are the first one into the office this morning. You check the builds logs from the overnight build and discover a test failure. When you run the test on your machine it passes. Now what?*
- ***Instructions***:
  - Given the following test code and the corresponding TestRunner output, how can you change the test to provide more diagnostic output?
- ***Discussion Questions:***
  - Based on the these symptoms, which Behavior Smells are we having?
  - What questions do we need to ask to find out why they are occurring?
  - What might be the underlying root causes?
  - What can we do about them?

Exercise – Frequent Debugging

- ***Symptoms:***
  - This customer test is failing because of a bug in the attached code.
- ***Instructions:***
  - Identify the unit tests that you could write to help you find the bug (without using the debugger) and keep it from coming back.
  - Just list the test conditions you would write tests for.
    - » **No need to write the actual tests**
- ***Discussion Questions:***
  - What part of the software is not tested thoroughly?

Recap of Behavior Smells

- **Slow Tests**
- **Erratic Tests**
  - Too many variants to list here
- **Fragile Tests**
  - The 4 sensitivities
- **Assertion Roulette**
- **Frequent Debugging**
- **Manual Intervention**

Recap of Patterns

- **Standard Fixture**
- **Shared Fixture**
- **Minimal Fixture**
- **Fresh Fixture**
- **Prebuilt Fixture**
- **Lazy Setup**
- **Setup Decorator**
- **SuiteFixture Setup**

What's a Code Smell?

A problem visible when looking at test code:

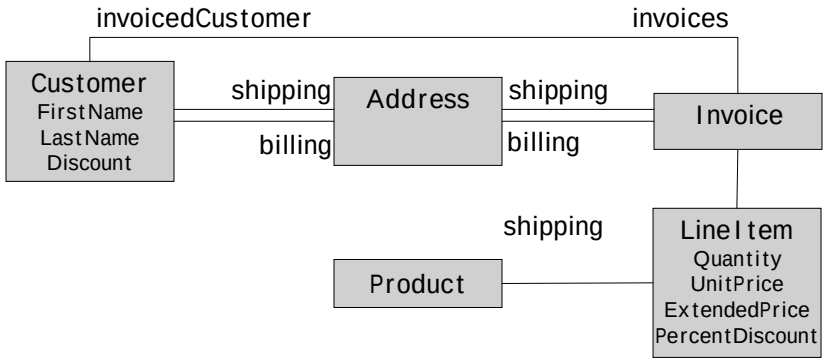
- **Tests are hard to understand**
- **Tests contain coding errors that may result in**
  - Missed bugs
  - Erratic Tests
- **Tests are difficult or impossible to write**
  - No test API on SUT
  - Cannot control initial state of SUT
  - Cannot observe final state of SUT
- **Sniff Test:**
  - Problem must be visible (in their face) to test automater or test reader

Common Code Smells

- **Conditional Test Logic**
- **Hard to Test Code**
- **Obscure Test**
- **Test Code Duplication**
- **Test Logic in Production**

Example

- **Test addItemQuantity and removeLineItem methods of Invoice**



The Whole Test

```
public void testAddItemQuantity_severalQuantity() throws Exception {
  try {
     // Setup Fixture
     final int QUANTITY = 5;
     Address billingAddress = new Address("1222 1st St SW", "Calgary", "Alberta", "T2N
           2V2", "Canada");
     Address shippingAddress = new Address("1333 1st St SW", "Calgary", "Alberta", "T2N
           2V2", "Canada");
     Customer customer = new Customer(99, "John", "Doe", new BigDecimal("30"),
           billingAddress, shippingAddress);
     Product product = new Product(88, "SomeWidget", new BigDecimal("19.99"));
     Invoice invoice = new Invoice(customer);
     // Exercise SUT
     invoice.addItemQuantity(product, QUANTITY);
     // Verify Outcome
     List lineItems = invoice.getLineItems();
     if (lineItems.size() == 1) {
       LineItem actualLineItem = (LineItem)lineItems.get(0);
       assertEquals(invoice, actualLineItem.getInvoice());
       assertEquals(product, actualLineItem.getProduct());
       assertEquals(quantity, actualLineItem.getQuantity());
       assertEquals(new BigDecimal("30"), actualLineItem.getPercentDiscount());
       assertEquals(new BigDecimal("19.99"), actualLineItem.getUnitPrice());
       assertEquals(new BigDecimal("69.96"), actualLineItem.getExtendedPrice());
     } else {
       assertTrue("Invoice should have exactly one line item", false);
     }
  } finally {
     deleteObject(expectedLineItem);
     deleteObject(invoice);
     deleteObject(product);
     deleteObject(customer);
     deleteObject(billingAddress);
     deleteObject(shippingAddress);
  }
}
```


Verifying the Outcome

```
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
  LineItem actualLineItem = (LineItem)lineItems.get(0);
  assertEquals(invoice, actualLineItem.getInvoice());
  assertEquals(product, actualLineItem.getProduct());
  assertEquals(quantity, actualLineItem.getQuantity());
  assertEquals(new BigDecimal("30"),
       actualLineItem.getPercentDiscount());
  assertEquals(new BigDecimal("19.99"),
       actualLineItem.getUnitPrice());
  assertEquals(new BigDecimal("69.96"),
       actualLineItem.getExtendedPrice());
} else {
  assertTrue("Invoice should have exactly one line item",
                        false);
}
```

Obtuse Assertion

Refactoring

Use Better Assertion

```
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
  LineItem actualLineItem = (LineItem)lineItems.get(0);
  assertEquals(invoice, actualLineItem.getInvoice());
  assertEquals(product, actualLineItem.getProduct());
  assertEquals(quantity, actualLineItem.getQuantity());
  assertEquals(new BigDecimal("30"),
       actualLineItem.getPercentDiscount());
  assertEquals(new BigDecimal("19.99"),
       actualLineItem.getUnitPrice());
  assertEquals(new BigDecimal("69.96"),
       actualLineItem.getExtendedPrice());
} else {
  fail("invoice should have exactly one line item");
}}
```


Refactoring

Use Better Assertion

```
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
  LineItem actualLineItem = (LineItem)lineItems.get(0);
  assertEquals(invoice, actualLineItem.getInvoice());
  assertEquals(product, actualLineItem.getProduct());
  assertEquals(quantity, actualLineItem.getQuantity());
  assertEquals(new BigDecimal("30"),
       actualLineItem.getPercentDiscount());
  assertEquals(new BigDecimal("19.99"),
       actualLineItem.getUnitPrice());
  assertEquals(new BigDecimal("69.96"),
       actualLineItem.getExtendedPrice());
} else {
  fail("invoice should have exactly one line item");
}}
```

Hard-Wired Test Data

Fragile Tests

Pattern

Expected Object

```
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
  LineItem actualLineItem = (LineItem)lineItems.get(0);
  LineItem expectedLineItem =
          newLineItem(invoice, product, QUANTITY);
  assertEquals(expectedLineItem.getInvoice(),
         actualLineItem.getInvoice());
  assertEquals(expectedLineItem.getProduct(),
         actualLineItem.getProduct());
  assertEquals(expectedLineItem.getQuantity(),
         actualLineItem.getQuantity());
  assertEquals(expectedLineItem.getPercentDiscount(),
         actualLineItem.getPercentDiscount());
  assertEquals(expectedLineItem.getUnitPrice(),
         actualLineItem.getUnitPrice());
  assertEquals(expectedLineItem.getExtendedPrice(),
         actualLineItem.getExtendedPrice());
} else {
  fail("invoice should have exactly one line item");
}}
```


Pattern

Expected Object

```
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
  LineItem actualLineItem = (LineItem)lineItems.get(0);
  LineItem expectedLineItem =  newLineItem(invoice,
        product, QUANTITY, product.getPrice()*QUANTITY );
  assertEquals(expectedLineItem.getInvoice(),
        actualLineItem.getInvoice());
  assertEquals(expectedLineItem.getProduct(),
        actualLineItem.getProduct());
  assertEquals(expectedLineItem.getQuantity(),
        actualLineItem.getQuantity());
  assertEquals(expectedLineItem.getPercentDiscount(),
        actualLineItem.getPercentDiscount());
  assertEquals(expectedLineItem.getUnitPrice(),
        actualLineItem.getUnitPrice());
  assertEquals(expectedLineItem.getExtendedPrice(),
        actualLineItem.getExtendedPrice());
} else {
  fail("invoice should have exactly one line item");
}
```

Verbose Test

Refactoring

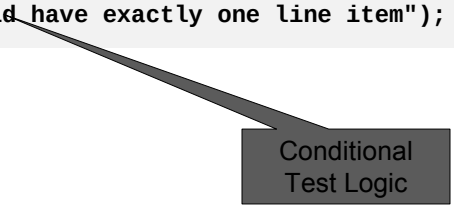
Introduce Custom Assert

```
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
  LineItem actualLineItem = (LineItem)lineItems.get(0);
  LineItem expectedLineItem =  newLineItem(invoice,
        product, QUANTITY, product.getPrice()*QUANTITY );
  assertLineItemsEqual(expectedLineItem, actualLineItem);
} else {
  fail("invoice should have exactly one line item");
}
```

Refactoring

Introduce Custom Assert

```
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
  LineItem actualLineItem = (LineItem)lineItems.get(0);
  LineItem expectedLineItem =  newLineItem(invoice,
        product, QUANTITY, product.getPrice()*QUANTITY );
  assertLineItemsEqual(expectedLineItem, actualLineItem);
} else {
  fail("invoice should have exactly one line item");
}
```



Refactoring

Replace Conditional Logic with Guard Assertion

```
List lineItems = invoice.getLineItems();
assertEquals("number of items",lineItems.size(),1);
LineItem actualLineItem = (LineItem)lineItems.get(0);
LineItem expectedLineItem =  newLineItem(invoice,
  product, QUANTITY, product.getPrice()*QUANTITY );
assertLineItemsEqual(expectedLineItem, actualLineItem);
```


The Whole Test

```
public void testAddItemQuantity_severalQuantity () throws Exception {
  try {
     // Setup Fixture
     final int QUANTITY = 5;
     Address billingAddress = new Address("1222 1st St SW", "Calgary", "Alberta", "T2N
          2V2", "Canada");
     Address shippingAddress = new Address("1333 1st St SW", "Calgary", "Alberta", "T2N
          2V2", "Canada");
     Customer customer = new Customer(99, "John", "Doe", new BigDecimal("30"),
          billingAddress, shippingAddress);
     Product product = new Product(88, "SomeWidget", new BigDecimal("19.99"));
     Invoice invoice = new Invoice(customer);
     // Exercise SUT
     invoice.addItemQuantity(product, QUANTITY);
     // Verify Outcome
     List lineItems = invoice.getLineItems();
     assertEquals("number of items",lineItems.size(),1);
     LineItem actualLineItem = (LineItem)lineItems.get(0);
     LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY);
     assertLineItemsEqual(expectedLineItem, actualLineItem);
  } finally {
     deleteObject(expectedLineItem);
     deleteObject(invoice);
     deleteObject(product);
     deleteObject(customer);
     deleteObject(billingAddress);
     deleteObject(shippingAddress);                                    :
}
```


The Smells Seen Thus Far (1)

- **Obscure Test**
  - Test is hard to understand.
- **Common Causes:**
  - Verbose Test
    - » **So much test code that it obscures the test intent**
  - Eager Test
    - » **Too many test conditions verified in one test case**
  - General Fixture
    - » **Fixture contains objects irrelevant for this test**
  - Obtuse Assertion
    - » **Using the wrong kind of assertion**
  - Hard-Coded Test Data
    - » **Lots of "Magic Numbers" or Strings used when creating objects.**
    - » **More likely to result in unrepeatable tests**

The Smells Seen Thus Far (2)

- **Other Obscure Test Causes:**
  - Indirect Testing
    - » **Interacting with the SUT via other software**
    - » **A cause of Fragile Tests (Behavior Smell)**
  - Mystery Guest
    - » **Lots of "Magic Numbers" or Strings used as keys to database.**
    - » **"Lopsided" feel to tests (either Setup or Verification of outcome is external to test)**

The Smells Seen Thus Far (3)

- **Conditional Test Logic**
  - Tests containing conditional logic (IF statements or loops)
  - Hard to verify correctness. Does it always test the same thing?
  - A cause of Buggy Tests (Project Smell)
- **Test Code Duplication**
  - Same code sequences appear many times in many tests
  - More code to modify when something changes
  - A cause of Fragile Tests (Behavior Smell)

The Patterns Used So Far

- **Expected Objects**
  - Use AssertEquals on whole objects rather than comparing individual fields
- **Guard Assertions**
  - Remove conditional logic associated with avoiding assertions when they would fail
- **Custom Asserts**
  - Remove Test Code Duplication by factoring out common code
  - Remove conditional logic associated with complex verification logic

Pattern

Expected Object

- **Replace a series of assertEquals on individual fields:**
  - assertEquals (expectedXvalue, actualPoint .getx());
  - assertEquals (expectedYvalue, actualPoint .getY());
- **with a single assertion of the whole object:**
  - assertEquals (expectedPoint, actualPoint);
- **Ways to construct the Expected Object:**
  - Point expectedPoint = new Point(17.0, 9.0)

  Or:
  - expectedPoint.setX(17.0);
  - expectedPoint.sety(9.0);

Pattern

Guard Assertion

Conditional Test Logic creates multiple execution paths thru test:

```
if (actualCollection == null)
    fail("collection is null");
else {
    assertTrue( actualCollection.includes(expectedElement) );
}
```

This makes tests hard to verify.

Better to replace with a Guard Assertion:

```
assertNotNull( "collection is null", actualCollection);
assertTrue( actualCollection.includes(expectedElement)
  );
```


Pattern

Custom Assertion

Remove duplicated assertion logic by creating your own Assertion Methods to:

- **Improve readability**
  - Intent-revealing methods that verify expected outcome
- **Simplify troubleshooting**
  - Make xUnit failure reports easier to understand
- **Define test-specific equality**
  - Ignore "don't care" fields when comparing objects
  - "Foreign Method" specific to testing
- **Can be defined using Extract Method refactoring.**

Exercise: CS1 – Result Verification

- ***Situation:***
  - *You have just inherited maintenance of the Flight Management System. The good news is that there are automated unit tests. The bad news is that most of the tests look something like these tests.*
- ***Instructions:***
  - Examine the code in the handout and determine what code smells you are seeing.
- ***Discussion Questions:***
  - Which Code Smells are we having?
  - What are the underlying root causes?
  - Which XUnit Patterns can we apply to alleviate them?

The Whole Test

```
public void testAddItemQuantity_severalQuantity () throws Exception {
  try {
     // Setup Fixture
     final int QUANTITY = 5;
     Address billingAddress = new Address("1222 1st St SW", "Calgary", "Alberta", "T2N
           2V2", "Canada");
     Address shippingAddress = new Address("1333 1st St SW", "Calgary", "Alberta", "T2N
           2V2", "Canada");
     Customer customer = new Customer(99, "John", "Doe", new BigDecimal("30"),
           billingAddress, shippingAddress);
     Product product = new Product(88, "SomeWidget", new BigDecimal("19.99"));
     Invoice invoice = new Invoice(customer);
     // Exercise SUT
     invoice.addItemQuantity(product, QUANTITY);
     // Verify Outcome
     List lineItems = invoice.getLineItems();
     assertEquals("number of items",lineItems.size(),1);
     LineItem actualLineItem = (LineItem)lineItems.get(0);
     LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY);
     assertLineItemsEqual(expectedLineItem, actualLineItem);
  } finally {
     deleteObject(expectedLineItem);
     deleteObject(invoice);
     deleteObject(product);
     deleteObject(customer);
     deleteObject(billingAddress);
     deleteObject(shippingAddress);                                    :
  }
```


Pattern

Inline Fixture Teardown - Naive

```
try {
   // Setup Fixture
   // Exercise SUT
   // Verify Outcome
} finally {
   deleteObject(expectedLineItem);
   deleteObject(invoice);
   deleteObject(product);
   deleteObject(customer);
   deleteObject(billingAddress);
   deleteObject(shippingAddress);
}
}
```


Pattern

Inline Fixture Teardown - Robust

```
try {
   // Setup Fixture
   // Exercise SUT
   // Verify Outcome
} finally {
   try {
      deleteObject(expectedLineItem);
   } finally {
      try {
         deleteObject(invoice);
      } finally {
         try {
           deleteObject(product);
         } finally {
               :
```


Pattern

Implicit Fixture Teardown - Naive

```
public void testAddItemQuantity_severalQuantity ()
  throws Exception {
   // Setup Fixture
   // Exercise SUT
   // Verify Outcome
}

public void tearDown() {
   deleteObject(expectedLineItem);
   deleteObject(invoice);
   deleteObject(product);
   deleteObject(customer);
   deleteObject(billingAddress);
   deleteObject(shippingAddress);
}
```

Even better: Avoid need to tear down

Pattern

Automated Fixture Teardown

```
public void testAddItemQuantity_severalQuantity () {
   final int QUANTITY = 5;
   Address billingAddress = new Address("1222 1st St SW",
             "Calgary", "Alberta", "T2N 2V2", "Canada");
   addTestObject( billingAddress );
   Address shippingAddress = new Address("1333 1st St SW",
             "Calgary", "Alberta", "T2N 2V2", "Canada");
   addTestObject(shippingAddress );

            :
}

public void tearDown() {
   deleteAllTestObjects();
}
```


Pattern

Automated Fixture Teardown

```
public void deleteAllTestObjects() {
    Iterator i = testObjects.iterator();
    while (i.hasNext()) {
       try {
          Deletable object = (Deletable)
       i.next();
          object.delete();
       } catch (Exception e) {
          // do nothing if the remove failed
       }
    }
}
```


Pattern

Transaction Rollback Teardown

```
public void setUp() {
   TransactionManager.beginTransaction();
}

public void tearDown() {
   TransactionManager.abortTransaction();
}
```

Important: SUT must not commit transaction

- DFT Pattern: Humble Transaction Controller

Only works with unit/component testing

Avoiding Tear Down Entirely

Options:

1. **Test Business Logic Units/Components separately from database**
   - Preferred approach in xUnit
2. **Fake out persistence mechanism so that it forgets the objects/data we create**
   - Possible for component testing
3. **Build brand new objects/data each time tests are run**
   - May be only option when testing entire system thru user interface

The Whole Test

```
public void testAddItemQuantity_severalQuantity() throws Exception {
    // Setup Fixture
    final int QUANTITY = 5;
    Address billingAddress = new Address("1222 1st St SW", "Calgary", "Alberta",
         "T2N 2V2", "Canada");
    addTestObject(billingAddress);
    Address shippingAddress = new Address("1333 1st St SW", "Calgary", "Alberta",
         "T2N 2V2", "Canada");
    addTestObject(billingAddress);
    Customer customer = new Customer(99, "John", "Doe", new BigDecimal("30"),
         billingAddress, shippingAddress);
    addTestObject(billingAddress);
    Product product = new Product(88, "SomeWidget", new BigDecimal("19.99"));
    addTestObject(billingAddress);
    Invoice invoice = new Invoice(customer);
    addTestObject(billingAddress);
    // Exercise SUT
    invoice.addItemQuantity(product, QUANTITY);
    // Verify Outcome
    assertEquals("number of items",lineItems.size(),1);
    LineItem actualLineItem = (LineItem)lineItems.get(0);
    LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY);
    assertLineItemsEqual(expectedLineItem, actualLineItem);
}
```

// No Visible Fixture Tear Down!

The Smells Seen Thus Far

- **Complex Undo Logic**
  - Complex fixture teardown code
  - More likely to leave test environment corrupted leading to Erratic Tests (Causes: Unrepeatable Tests or Interacting Tests)

The Patterns Used So Far

- **Inline Teardown**
  - Hand-coded tear down logic within the Test Method
- **Implicit Teardown**
  - Hand-coded tear down logic in a tearDown method
- **Automated Teardown**
  - Tear down all registered test objects programatically
- **Transaction Rollback Teardown**
  - Get the database to undo all the changes made by test
  - SUT must not commit transaction

The Whole Test

```
public void testAddItemQuantity_severalQuantity() throws Exception {
    // Setup Fixture
    final int QUANTITY = 5;
    Address billingAddress = new Address("1222 1st St SW", "Calgary",
        "Alberta", "T2N 2V2", "Canada");
    addTestObject(billingAddress);
    Address shippingAddress = new Address("1333 1st St SW", "Calgary",
        "Alberta", "T2N 2V2", "Canada");
    addTestObject(billingAddress);
    Customer customer = new Customer(99, "John", "Doe", new BigDecimal("30"),
        billingAddress, shippingAddress);
    addTestObject(billingAddress);
    Product product = new Product(88, "SomeWidget", new BigDecimal("19.99"));
    addTestObject(billingAddress);
    Invoice invoice = new Invoice(customer);
    addTestObject(billingAddress);
    // Exercise SUT
    invoice.addItemQuantity(product, QUANTITY);
    // Verify Outcome
    assertEquals("number of items",lineItems.size(),1);
    LineItem actualLineItem = (LineItem)lineItems.get(0);
    LineItem expectedLineItem = newLineItem(invoice, product, QUANTITY);
    assertLineItemsEqual(expectedLineItem, actualLineItem);
}
```


Smell

Hard-Coded Test Data

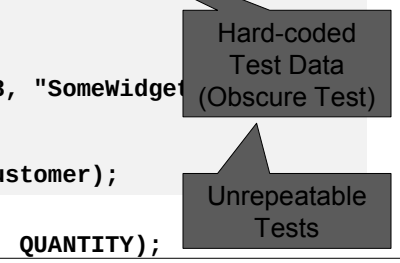
```
public void testAddItemQuantity_severalQuantity() {
    final int QUANTITY = 5;
    Address billingAddress = new Address("1222 1st St SW",
        "Calgary", "Alberta", "T2N 2V2", "Canada");

    Address shippingAddress = new Address("1333 1st St SW",
        "Calgary", "Alberta", "T2N 2V2", "Canada");

    Customer customer = new Customer(99, "John", "Doe", new
        BigDecimal("30"), billingAddress, shippingAddress);

    Product product = new Product(88, "SomeWidget",
        BigDecimal("19.99"));

    Invoice invoice = new Invoice(customer);
    // Exercise SUT
    invoice.addItemQuantity(product, QUANTITY);
```



Pattern

Distinct Generated Values

```
public void testAddItemQuantity_severalQuantity() {
    final int QUANTITY = 5 ;
    Address billingAddress = new Address(getUniqueString(),
        getUniqueString(), getUniqueString(),
        getUniqueString(), getUniqueString());
    Address shippingAddress = new Address(getUniqueString(),
        getUniqueString(), getUniqueString(),
        getUniqueString(), getUniqueString());
    Customer customer = new Customer(
        getUniqueInt(), getUniqueString(),
        getUniqueString(), getUniqueDiscount(),
        billingAddress, shippingAddress);
    Product product = new Product(
        getUniqueInt(), getUniqueString(),
        getUniqueNumber());
    Invoice invoice = new Invoice(customer);
```


Pattern

Distinct Generated Values

```
public void testAddItemQuantity_severalQuantity() {
    final int QUANTITY = 5 ;
    Address billingAddress = new Address(getUniqueString(),
        getUniqueString(), getUniqueString(),
        getUniqueString(), getUniqueString());
    Address shippingAddress = new Address(getUniqueString(),
        getUniqueString(), getUniqueString(),
        getUniqueString(), getUniqueString());
    Customer customer1 = new Customer(
        getUniqueInt(), getUniqueString(),
        getUniqueString(), getUniqueDiscount(),
        billingAddress, shippingAddress);
    Product product = new Product(
        getUniqueInt(), getUniqueString(),
        getUniqueNumber());
    Invoice invoice = new Invoice(customer);
```

Irrelevant Information (Obscure Test)

Pattern

Creation Method

```
public void testAddItemQuantity_severalQuantity() {
   final int QUANTITY = 5;
   Address billingAddress = createAnonymousAddress();

   Address shippingAddress = createAnonymousAddress();

   Customer customer = createCustomer( billingAddress,
       shippingAddress);

   Product product = createAnonymousProduct();

   Invoice invoice = new Invoice(customer);
```


Smell

Obscure Test - Irrelevant Information

```
public void testAddItemQuantity_severalQuantity() {
   final int QUANTITY = 5;
   Address billingAddress = createAnonymousAddress();
   Address shippingAddress = createAnonymousAddress();
   Customer customer = createCustomer(
       billingAddress, shippingAddress);
   Product product = createAnonymousProduct();
   Invoice invoice = new Invoice(customer);
   // Exercise
   invoice.addItemQuantity(product, QUANTITY);
   // Verify
   LineItem expectedLineItem =   newLineItem(invoice,
       product, QUANTITY, product.getPrice()*QUANTITY );
   List lineItems = invoice.getLineItems();
   assertEquals("number of items", lineItems.size(), 1);
   LineItem actualLineItem = (LineItem)lineItems.get(0);
   assertLineItemsEqual(expectedLineItem, actualLineItem);
}
```

Irrelevant Information (Obscure Test)

Refactoring

Remove Irrelevant Information

```
public void testAddItemQuantity_severalQuantity() {
    final int QUANTITY = 5 ;

    Customer customer = createAnonymousCustomer();

    Product product = createAnonymousProduct();
    Invoice invoice = new Invoice(customer);
    // Exercise
    invoice.addItemQuantity(product, QUANTITY);
    // Verify
    LineItem expectedLineItem =   newLineItem(invoice,
        product, QUANTITY, product.getPrice()*QUANTITY );
    List lineItems = invoice.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actualLineItem = (LineItem)lineItems.get(0);
    assertLineItemsEqual(expectedLineItem, actualLineItem);
}
```

Irrelevant Information (Obscure Test)

Refactoring

Remove Irrelevant Information

```
public void testAddItemQuantity_severalQuantity() {
    final int QUANTITY = 5 ;

    Product product = createAnonymousProduct();
    Invoice invoice = createAnonymousInvoice()
    // Exercise
    invoice.addItemQuantity(product, QUANTITY);
    // Verify
    LineItem expectedLineItem =   newLineItem(invoice,
        product, QUANTITY, product.getPrice()*QUANTITY );
    List lineItems = invoice.getLineItems();
    assertEquals("number of items", lineItems.size(), 1);
    LineItem actualLineItem = (LineItem)lineItems.get(0);
    assertLineItemsEqual(expectedLineItem, actualLineItem);
}
```


Refactoring

Introduce Custom Assertion

```
public void testAddItemQuantity_severalQuantity() {
   final int QUANTITY = 5 ;

   Product product = createAnonymousProduct();
   Invoice invoice = createAnonymousInvoice()
   // Exercise
   invoice.addItemQuantity(product, QUANTITY);
   // Verify
   LineItem expectedLineItem =   newLineItem(invoice,
      product, QUANTITY, product.getPrice()*QUANTITY );
   List lineItems = invoice.getLineItems();
   assertEquals("number of items", lineItems.size(), 1);
   LineItem actualLineItem = (LineItem)lineItems.get(0);
   assertLineItemsEqual(expectedLineItem, actualLineItem);
}
```

Mechanics hides Intent

Refactoring

Introduce Custom Assertion

```
public void testAddItemQuantity_severalQuantity() {
   final int QUANTITY = 5 ;

   Product product = createAnonymousProduct();
   Invoice invoice = createAnonymousInvoice()
   // Exercise
   invoice.addItemQuantity(product, QUANTITY);
   // Verify
   LineItem expectedLineItem =
      newLineItem(invoice, product, QUANTITY,
      product.getPrice()*QUANTITY );
   assertExactlyOneLineItem(invoice, expectedLineItem
      );
}
```


The Whole Test – Done

```
public void testAddItemQuantity_severalQuantity() {
   final int QUANTITY = 5 ;
   Product product = createAnonymousProduct();
   Invoice invoice = createAnonymousInvoice();
   // Exercise
   invoice.addItemQuantity(product, QUANTITY);
   // Verify
   LineItem expectedLineItem =   newLineItem(invoice,
       product, QUANTITY, product.getPrice()*QUANTITY );
   assertExactlyOneLineItem(invoice, expectedLineItem );
}
```


Test Coverage

TestInvoiceLineItems extends TestCase {
- **TestAddItemQuantity_oneItem {..}**
- **TestAddItemQuantity_severalItems {..}**
- **TestAddItemQuantity_duplicateProduct {..}**
- **TestAddItemQuantity_zeroQuantity {..}**
- **TestAddItemQuantity_severalQuantity {..}**
- **TestAddItemQuantity_discountedPrice {..}**
- **TestRemoveItem_noItemsLeft {..}**
- **TestRemoveItem_oneItemLeft {..}**
- **TestRemoveItem_ severalItemsLeft {..}**

}

Pattern: Testcase Class per Feature

Rapid Test Writing

```
public void testAddItemQuantity_severalItems() {
   final int QUANTITY = 1 ;
   Product product1 = createAnonymousProduct();
   Product product2 = createAnonymousProduct();
   Invoice invoice = createAnonymousInvoice();
   // Exercise
   invoice.addItemQuantity(product1, QUANTITY);
   invoice.addItemQuantity(product2, QUANTITY);
   // Verify
   LineItem expectedLineItem1 =  newLineItem(invoice,
       product, QUANTITY, product.getPrice()*QUANTITY );
   LineItem expectedLineItem2 =  newLineItem(invoice,
       product, QUANTITY, product.getPrice()*QUANTITY );
   assertExactlyTwoLineItems(invoice,
                    expectedLineItem1, expectedLineItem2 );
}
```


The Smells Seen Thus Far

- **Obscure Test**

  The test is hard to understand. Specific causes:

  – Hard-Coded Test Data
    - » **Literal Constants**
  – Irrelevant Information
    - » **Information in test unrelated to SUT behavior**

The Patterns Used so Far

- **Generated Value**
  - Variation: Distinct Generated Value
    - » **Generate a unique value for each test run**
- **Creation Method**
  - Anonymous Creation Method
    - » **Sets all attributes/references to default values**
  - Parameterized Creation Method
    - » **Tests specifies relevant values only**
- **Testcase Class per Feature**
  - Group all Test Methods for a feature or concept on a single class
  - Alternatives: Testcase Class per Class, Testcase Class per Fixture
- **Custom Assertion**
  - » **(again)**

Exercise: CS2 – Fixture Set Up

- ***Situation:***
  - *You have just inherited maintenance of the Flight Management System. The good news is that there are automated unit tests. The bad news is that most of the tests look something like these tests.*
- ***Instructions:***
  - Examine the code in the handout and determine what code smells you are seeing.
- ***Discussion Questions:***
  - Which Code Smells are we having?
  - What are the underlying root causes?
  - Which Patterns can we apply to alleviate them?

Smell

Hard to Test Code

- **Code can be hard to test for a number of reasons:**
  - Too closely coupled to other software
  - No interface provided to set state, observe state
  - Only asynchronous interfaces provided
- **Root Cause is lack of Design for Testability**
  - Comes naturally with Test-Driven Development
  - Must be retrofitted to legacy (test-less) software
- **Temporary Workaround is Test Hook**
  - Becomes Test Logic in Production (code smell) if not removed

Smell

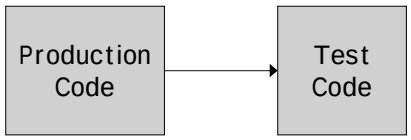
Test Logic in Production

Test Hook:

```
If (testing) then
  // test-specific logic
else
  // production logic
endif
```

Test Dependency:

```
Production Module
import TestBlahBlahBlah;
```



A Recipe for Success

1. **Write some tests**
   - start with the easy ones!
2. **Note the Test Smells that show up**
3. **Refactor to remove obvious Test Smells**
   - Apply appropriate xUnit Test Patterns
4. **Write some more tests**
   - possibly more complex
5. **Repeat from Step 2 until:**
   - All necessary tests written
   - No smells remain

Recap of Code Smells

- **Conditional Test Logic**
- **Hard to Test Code**
- **Obscure Test**
- **Test Code Duplication**
- **Test Logic in Production**

Recap of Patterns Used:

- **Expected Object**
- **Custom Assertion**
- **Guard Assertion**
- **Inline Teardown**
- **Implicit Teardown**
- **Automated Teardown**
- **Transaction Rollback Teardown**
- **(*Anonymous/Parameterized*) Creation Method**
- **(*Distinct*) Generated Value**
- **Humble Object**
- **Dependency Injection / Lookup**
- **Test Double / Test Stub / Mock Object / Fake Object**
- **Test-Specific Subclass**
- **Testcase Class per Feature/Fixture/Class**

Agenda

- **Introduction**
  - Metaphors
- **Test Strategy**
  - Why test?
  - Why Automate?
  - Which Automation Strategy?
- **Test Behavior Smells**
  - Patterns to address them
- **Test Code Smells**
  - Patterns to address them
- **Wrap Up**

What Next?

- **You have a better idea of:**
  - What can be achieved with good automated tests
  - Three kinds of Test Smells to look for
    - » **Code Smells, Behavior Smells and Project Smells**
  - Symptoms (Smells) vs root causes of them
- **You have an initial list of patterns to address root causes**
  - Test automation strategy patterns
  - Test design patterns
  - Test coding patterns
- **More at the web site and in the book**

Be Pragmatic!

- **Not all Smells can (or should) be eliminated**
  - Cost of having smell vs. cost of removing it
  - Cost to remove it now vs. cost of removing it later
- **Catalog of Smells and Causes gives us the tools to make the decision intelligently**
  - Trouble-shooting flow chart
  - Suggested Patterns for removing cause
- **Catalog of Patterns gives us the tools to eliminate the Smells *when we choose to do so***
  - How it Works
  - When to Use It
  - Before/After Code samples
  - Refactoring notes

What ***Does*** it Take To be Successful?

Testing Experience
+ Programing Experience
+Test Automation Tool Experience
+ Design for Testability
- Test Smells
+ Test Automation Patterns
- Test Debt
+ Fanatical Attention to Test Maintainability

= Robust, maintainable automated tests, rapid data on quality, and defect *prevention*

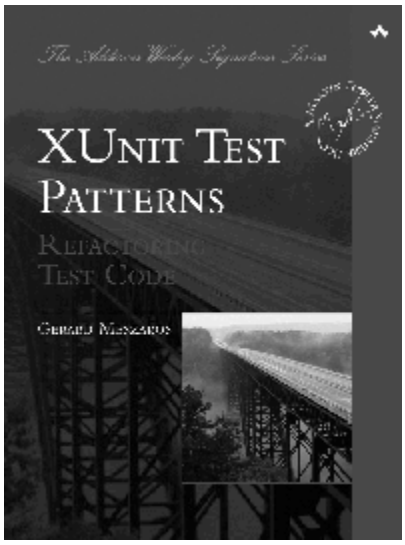
What Does it Take To be Successful?

Collaboration between Development & Test

- **Test organization sharing tests with developers before development**
- **Development organization helping Test with:**
  - Automation of tests
  - Design for Testability
- **Commitment to *preventing* defects**
  - Not insert, find & fix!

More on xUnit Patterns & Smells

- **Book:**
  xUnit Test Patterns
  Refactoring Test Code
  by: Gerard Meszaros
  – published by Addison Wesley
  – available Now!
- **Website:**
  http://xunitpatterns.com
  With handy links to purchase



Thank You!
Gerard

Questions & Comments?

Books on xUnit Test Automation

- **xUnit Test Patterns – Refactoring Test Code**
  - Gerard Meszaros
- **Test-driven Development - A Practical Guide**
  - David Astels
- **Test-driven Development - By Example**
  - Kent Beck
- **Test-Driven Development in Microsoft .NET**
  - James Newkirk, Alexei Vorontsov
- **Unit Testing With Java - How tests drive the code**
  - Johannes Link
- **JUnit Recipes**
  - J.B. Rainsberger

Other Useful Books

- **Working Effectively with Legacy Code**
  - Michael Feathers
- **Fit for Software Development**
  - Rick Mugridge, Ward Cunningham
- **Refactoring - Improving the Design of Existing Code**
  - Martin Fowler plus contributors
- **Design Patterns: Reusable Elements of Design**
  - Erich Gamma, Richard Helm, Ralph Johnson, John Vlisides

