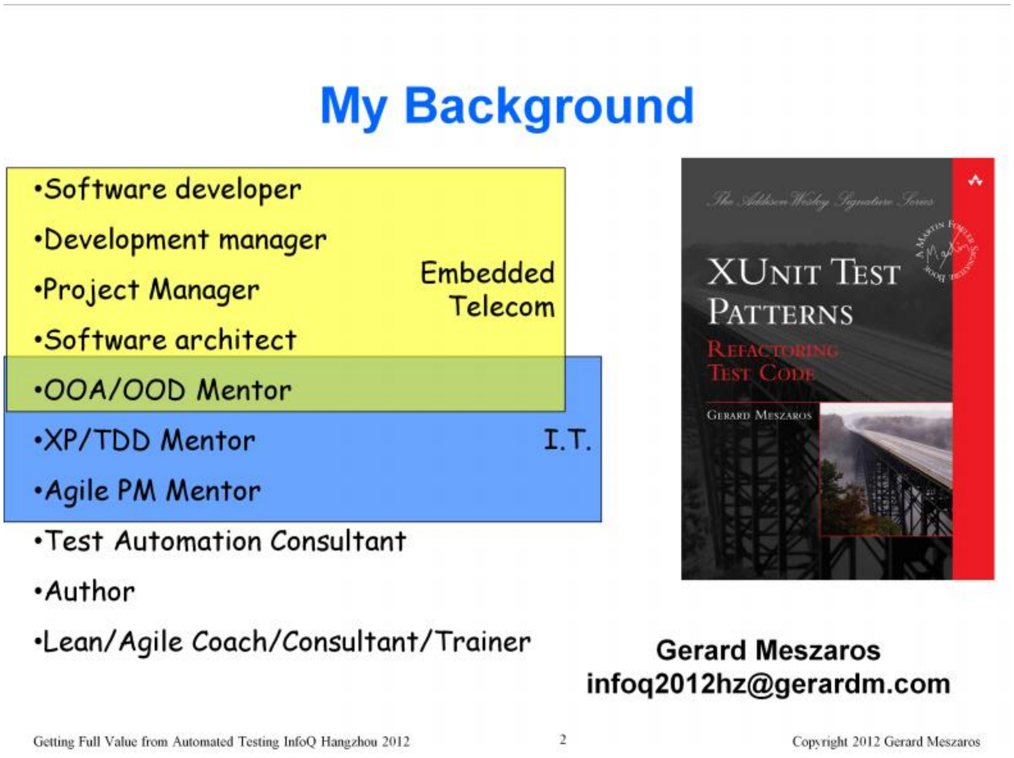
# Getting Full Value from Automated Testing

**Gerard Meszaros**
infoQ2012hz@gerardm.com



## My Background

- Software developer
- Development manager
- Project Manager
- Software architect
- OOA/OOD Mentor
- XP/TDD Mentor
- Agile PM Mentor
- Test Automation Consultant
- Author
- Lean/Agile Coach/Consultant/Trainer

Embedded Telecom

I.T.

The Addison Wesley Signature Series
XUnit Test Patterns
Refactoring Test Code
Gerard Meszaros
A Martin Fowler Signature Book

Gerard Meszaros
infoq2012hz@gerardm.com

My original background is Telecom Switching Software.

In my 14 years at Nortel, we put out 45 releases of the DMS-100 switching system software.

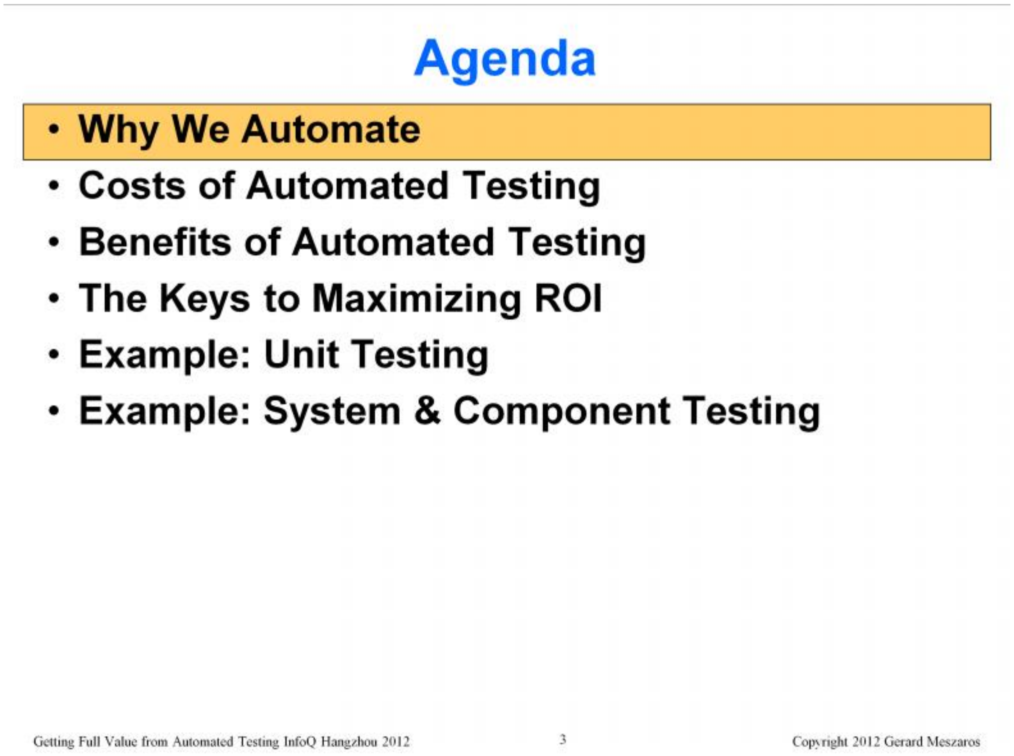
I became an IT consultant in 1995

I started doing automated unit testing in 1996

I started doing XP in 2000

My book was published in 2007 after a 3.5 year gestation period.

Currently, I teach courses on best practices in test automation and agile development and coach agile teams.

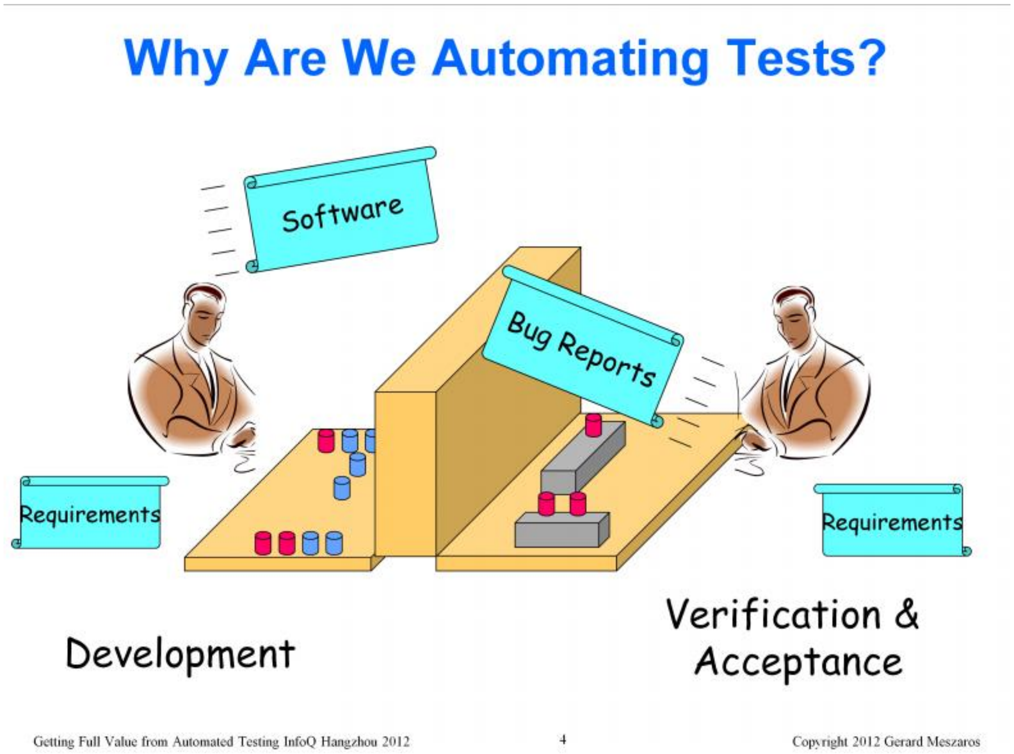


## Agenda

- Why We Automate
- Costs of Automated Testing
- Benefits of Automated Testing
- The Keys to Maximizing ROI
- Example: Unit Testing
- Example: System & Component Testing

This talk is about how to ensure you get the most bang for your Yuan (or Chaio?) What I hope you'll get out of this is some ideas about how to get the most out of your tests and if necessary, how to sell the idea of automating tests to your management.

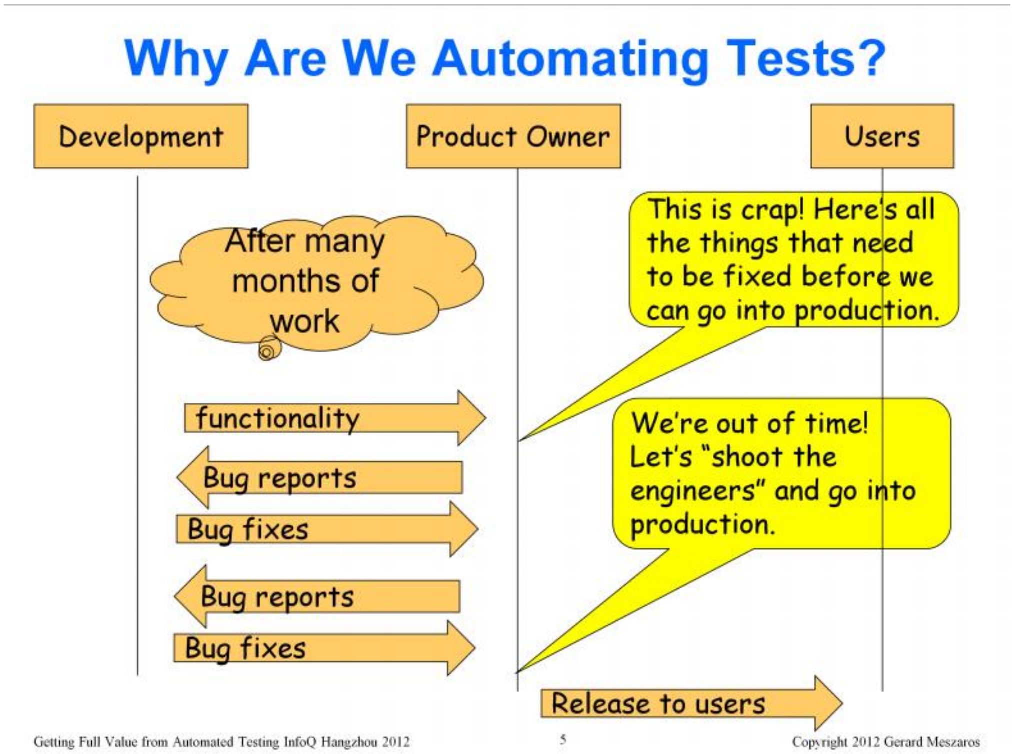
I'll start out by defining the ROI or Return on Investment of automated testing. I will provide an overview of the various kinds of costs we need to be aware of and the ways we benefit from the tests. Then I will focus in on some of the most tangible ways we can affect the ROI. I'll conclude with two examples of how to apply these ideas, first to unit-level tests, and second, to business-level tests including system and component tests.



## Why Are We Automating Tests?

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 4 Copyright 2012 Gerard Meszaros

The traditional waterfall or phased approach to testing involves developers tossing the finished product “over the wall” to the independent test team. The testers then create bug reports for each problem they find and toss those back over the wall to the developers.



## Why Are We Automating Tests?

Development

Product Owner

Users

After many months of work

functionality

This is crap! Here's all the things that need to be fixed before we can go into production.

Bug reports

Bug fixes

We're out of time! Let's "shoot the engineers" and go into production.

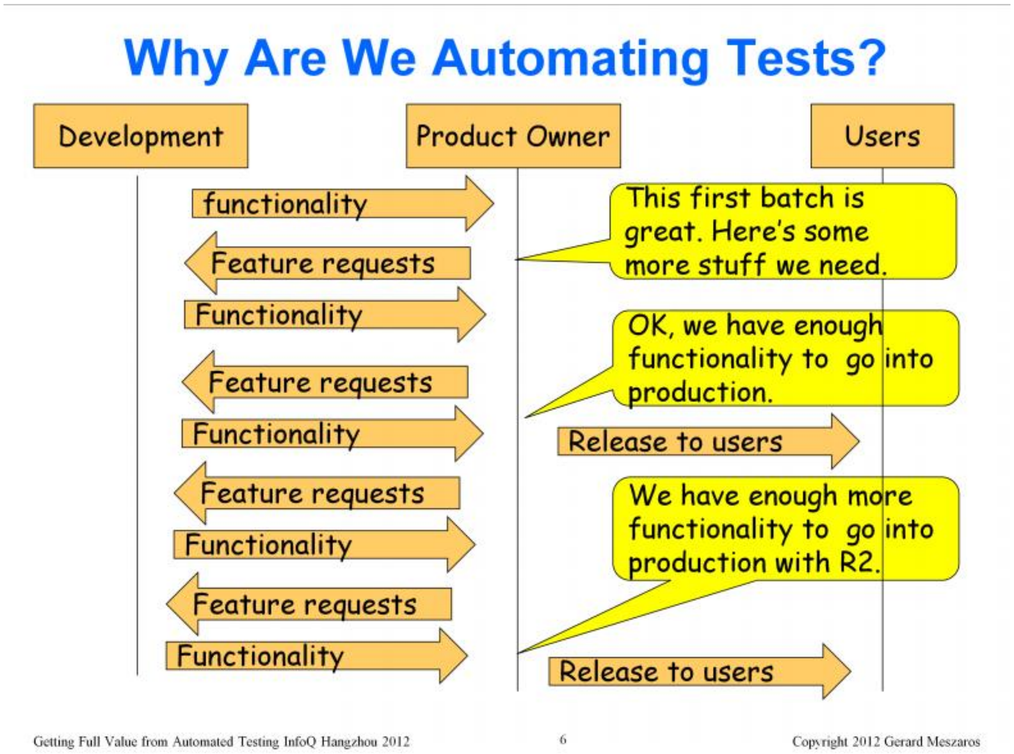
Bug reports

Bug fixes

Release to users

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 5 Copyright 2012 Gerard Meszaros

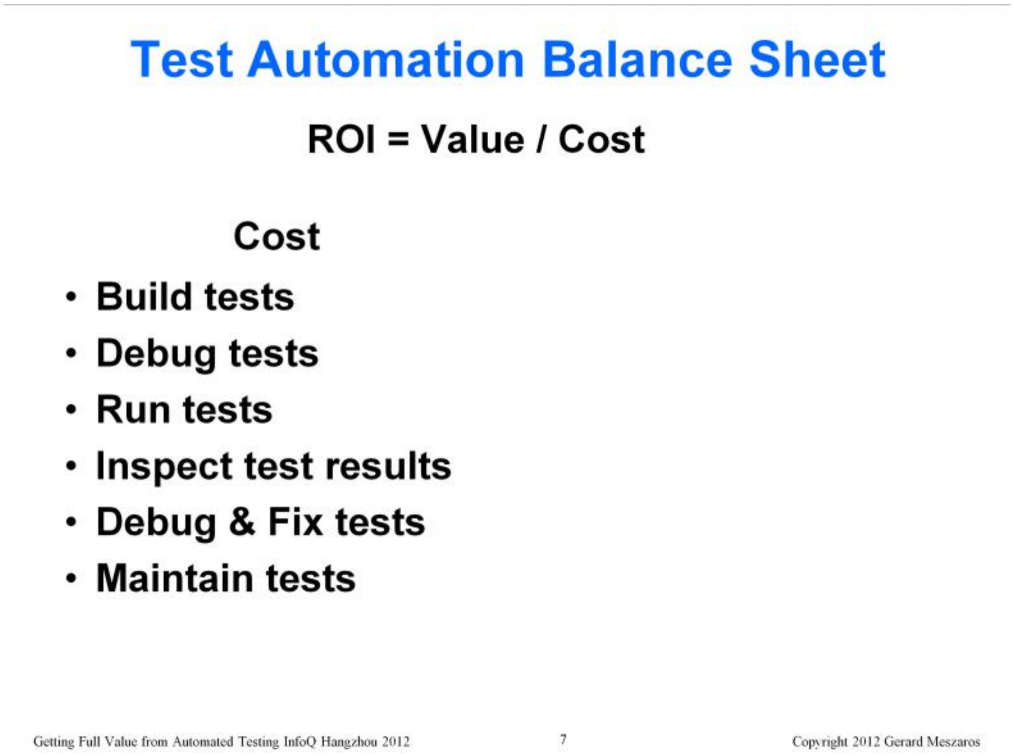
This process would consume the last quarter of the typical product schedule despite the fact that many of the bugs are never fixed. At some point, the product manager would decide it was time to “Shoot the engineers and put the product into production.”



## Why Are We Automating Tests?

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 6 Copyright 2012 Gerard Meszaros

Agile projects try to deliver working code to the product owner every few weeks or more frequently. This avoids the "big bang integration" and "test phase" but it creates a new problem: The software needs to be retested every few weeks or even more often. This has motivated agile teams to invest heavily in test automation so that the cost of rerunning the tests is greatly reduced. But are we getting enough value to justify the investment in the tests?



## Test Automation Balance Sheet

**ROI = Value / Cost**

**Cost**

- **Build tests**
- **Debug tests**
- **Run tests**
- **Inspect test results**
- **Debug & Fix tests**
- **Maintain tests**

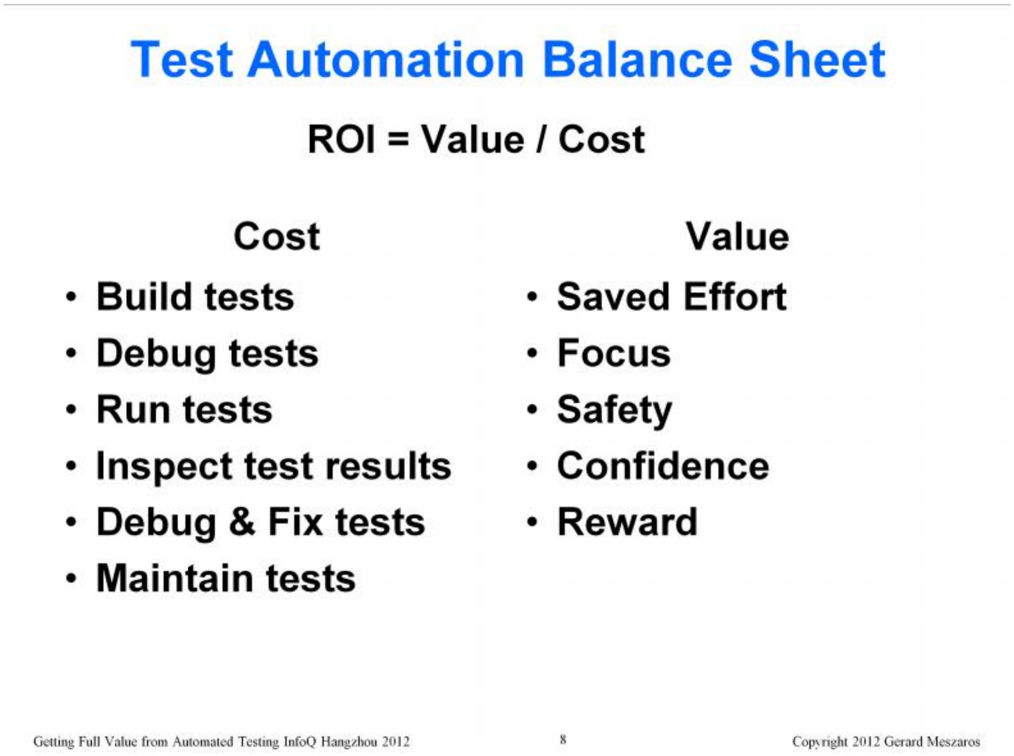
Getting Full Value from Automated Testing InfoQ Hangzhou 2012 7 Copyright 2012 Gerard Meszaros

The return on investment is the value received over the cost incurred. If the value exceeds the cost, then we are ahead. If the value is less than the cost, then we aren’t earning back our investment

The costs are relatively easy to enumerate. They include the cost of building the tests, the cost of running the tests, the cost of inspecting the test results to decide whether or not they passed, and the cost to fix bugs in the tests and to maintain the tests when the software or environment changes.

The value may be a bit harder to quantify but let’s try anyway. Of course, we expect them to save us considerable effort while testing but is this by itself enough to offset the costs? We also get value from the focus and safety the tests provide, and these improve our confidence. And the tests make our work more rewarding.

To get the best ROI, we need to minimize the costs and maximize the value. Let’s look at how we can do this.



## Test Automation Balance Sheet

**ROI = Value / Cost**

**Cost**

- **Build tests**
- **Debug tests**
- **Run tests**
- **Inspect test results**
- **Debug & Fix tests**
- **Maintain tests**

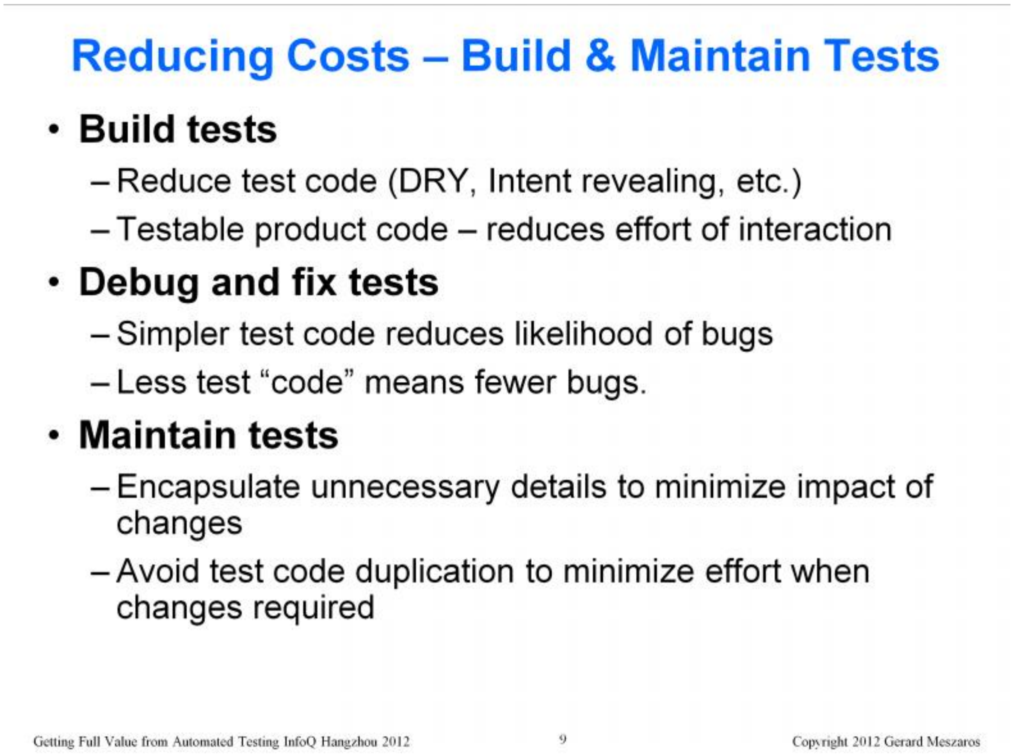
**Value**

- **Saved Effort**
- **Focus**
- **Safety**
- **Confidence**
- **Reward**

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 8 Copyright 2012 Gerard Meszaros

The value may be a bit harder to quantify but let’s try anyway. Of course, we expect them to save us considerable effort while testing but is this by itself enough to offset the costs? We also get value from the focus and safety the tests provide, and these improve our confidence. And the tests make our work more rewarding.

To get the best ROI, we need to minimize the costs and maximize the value. Let’s look at how we can do this.



## Reducing Costs – Build & Maintain Tests

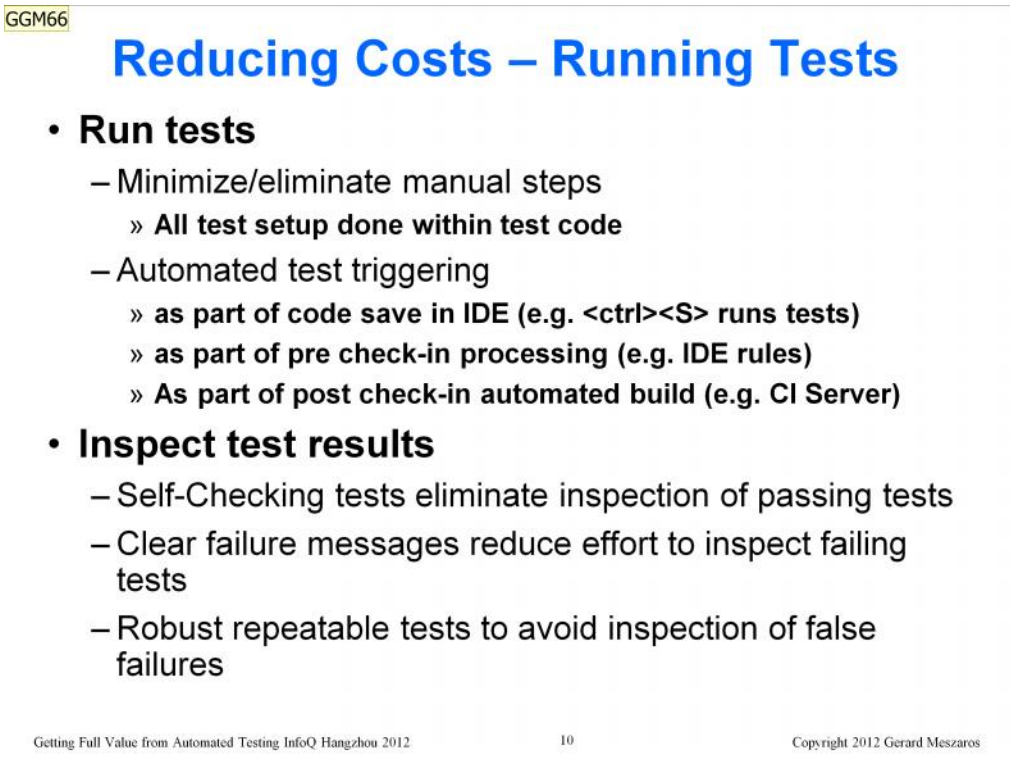
- **Build tests**
  - Reduce test code (DRY, Intent revealing, etc.)
  - Testable product code – reduces effort of interaction
- **Debug and fix tests**
  - Simpler test code reduces likelihood of bugs
  - Less test “code” means fewer bugs.
- **Maintain tests**
  - Encapsulate unnecessary details to minimize impact of changes
  - Avoid test code duplication to minimize effort when changes required

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 9 Copyright 2012 Gerard Meszaros

We can minimize the cost of building the tests by reducing the amount of test code we write. We do this by elevating the level of the language we use while writing the tests to avoid unnecessary details. We avoid repetition between and within tests. And the simpler we make the test code, the less likely we are to introduce bugs while we write it and the less test debugging we’ll need.

Not only does this reduce the effort to write the tests, it also reduces the number of places we need to change when something in our product changes. This will reduce our maintenance costs when we change the functionality of our product.

I’ll show you some examples of how we can do this a bit later in this talk.



GGM66

## Reducing Costs – Running Tests

- **Run tests**
  - Minimize/eliminate manual steps
    - » **All test setup done within test code**
  - Automated test triggering
    - » **as part of code save in IDE (e.g. <ctrl><S> runs tests)**
    - » **as part of pre check-in processing (e.g. IDE rules)**
    - » **As part of post check-in automated build (e.g. CI Server)**
- **Inspect test results**
  - Self-Checking tests eliminate inspection of passing tests
  - Clear failure messages reduce effort to inspect failing tests
  - Robust repeatable tests to avoid inspection of false failures

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 10 Copyright 2012 Gerard Meszaros

We can minimize the cost of running our tests by ensuring everything is automated. Tests should set up their own starting points and check their own results so that no manual intervention is ever required. The tests should be invoked automatically, triggered by the appropriate events:

•Every time we save and compile code in our IDE, the tests should run automatically, without any effort involved.

•As we initiate a check-in operation, all the tests should run automatically, before the check-in proceeds

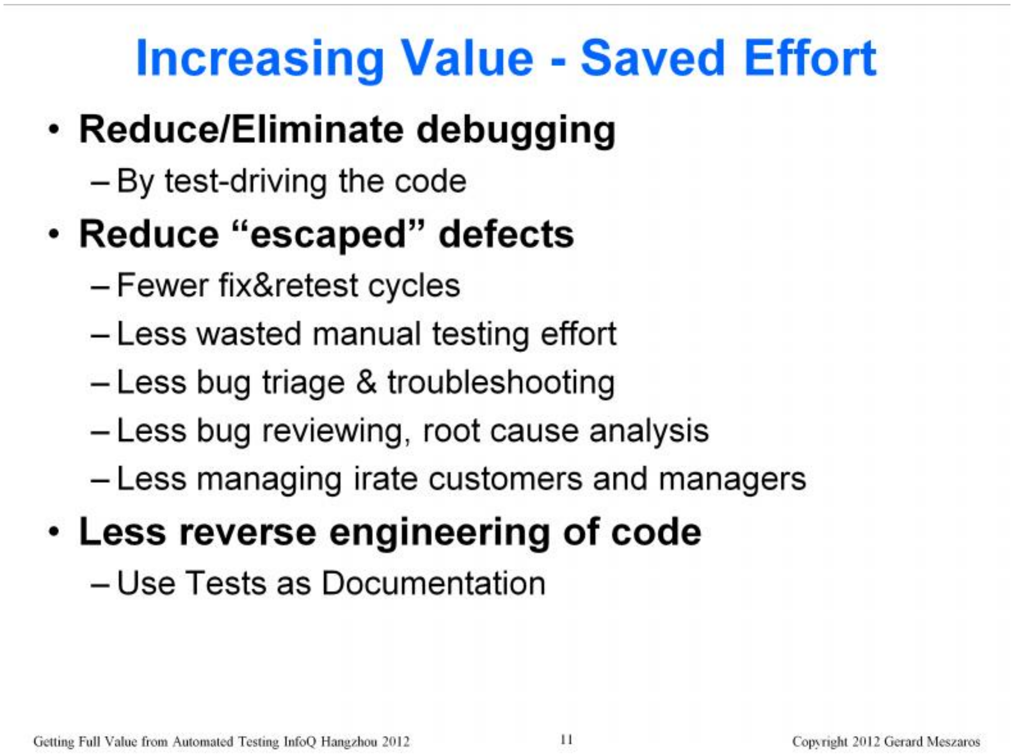
•And the newly checked-in code should be built and tested automatically on our build server. Here, we can run a more extensive suite of tests that takes longer or requires resources not available on the developer desktop

•When our self-checking tests fail, we want to spend as little time looking at them as possible. The failure messages should be clear and describe exactly what deviated from the expected results. And we need to ensure our tests never fail when the code works correctly; they need to control everything on which the code being tested depends.

## Slide 10

GGM66 Order of notes and bullets don't line up. Either fancy animation or reorganize one or the other.

Gerard Meszaros, 10/19/2012



## Increasing Value - Saved Effort

- **Reduce/Eliminate debugging**
  - By test-driving the code
- **Reduce “escaped” defects**
  - Fewer fix&retest cycles
  - Less wasted manual testing effort
  - Less bug triage & troubleshooting
  - Less bug reviewing, root cause analysis
  - Less managing irate customers and managers
- **Less reverse engineering of code**
  - Use Tests as Documentation

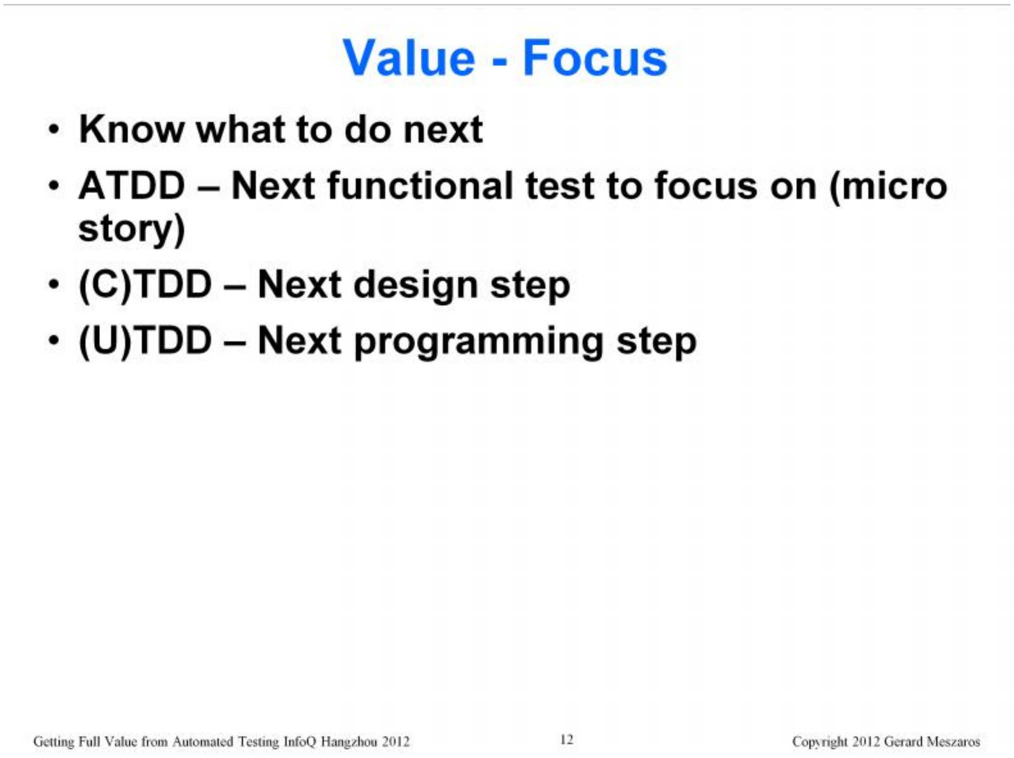
Getting Full Value from Automated Testing InfoQ Hangzhou 2012 11 Copyright 2012 Gerard Meszaros

Now lets look at the value side of the ROI equation. These are the things we want to maximize.

We’d like to hope that our tests save us a lot of effort testing and retesting our software. But that’s not the only way we can leverage our tests to save effort.

When we test-drive our code, that is the tests are written before the product code and run while coding, we can greatly reduce or even eliminate debugging. We’ll have fewer bugs slipping through to the test team or the customer and that saves manual test effort. We have fewer bugs to troubleshoot and prioritize and do root-cause analysis on and fix and apologize for; that saves us effort.

Furthermore, if the the tests are written clearly, they describe what the code should do in varous situations. This means we don’t have to read the code to reverse engineer what it does; we can just read the tests. People new to the code can learn it much more quickly.

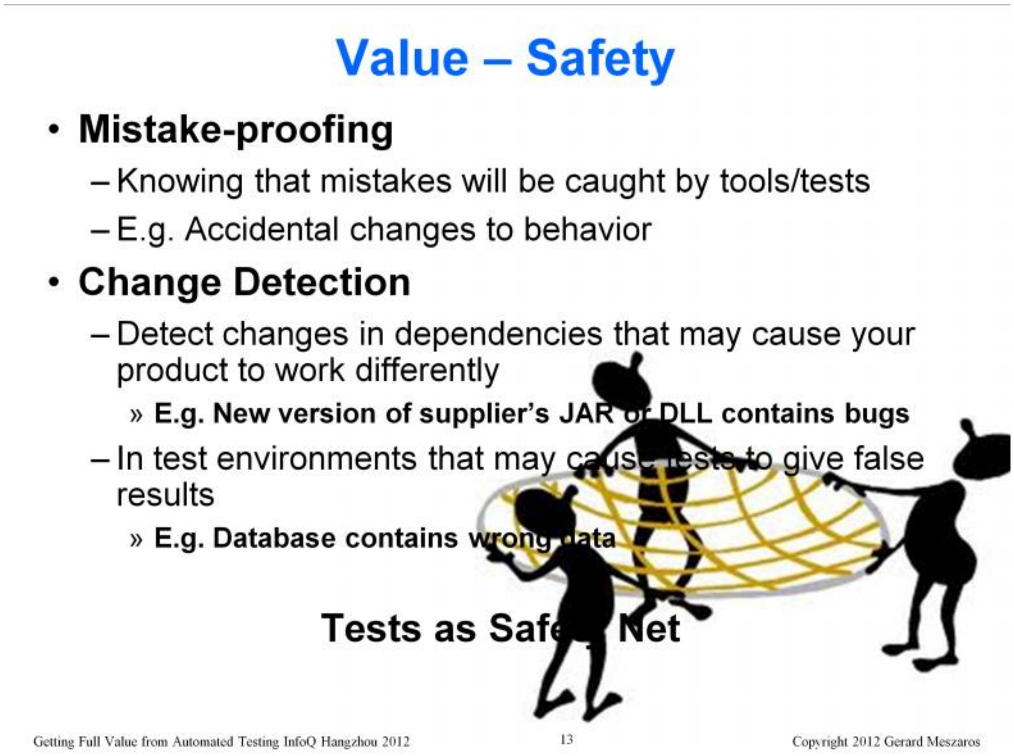


## Value - Focus

- Know what to do next
- ATDD – Next functional test to focus on (micro story)
- (C)TDD – Next design step
- (U)TDD – Next programming step

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 12 Copyright 2012 Gerard Meszaros

But cost savings aren’t the only form of value. When we test-drive our code, the tests give us focus. Acceptance test driving our code helps us focus on the next scenario we need to implement. Component and unit test driving helps us understand what piece of logic we need to program next. If we only write code to pass a failing test, we avoid writing any unnecessary code.



## Value – Safety

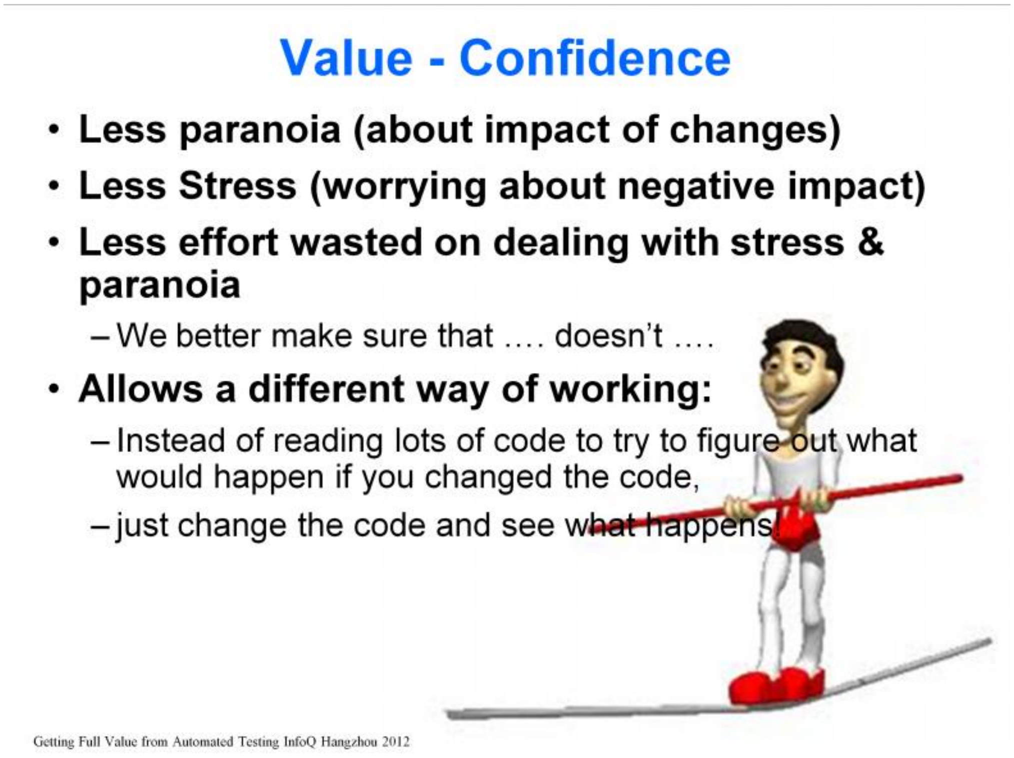
- **Mistake-proofing**
  - Knowing that mistakes will be caught by tools/tests
  - E.g. Accidental changes to behavior
- **Change Detection**
  - Detect changes in dependencies that may cause your product to work differently
    - » **E.g. New version of supplier's JAR or DLL contains bugs**
  - In test environments that may cause tests to give false results
    - » **E.g. Database contains wrong data**

**Tests as Safety Net**

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 13 Copyright 2012 Gerard Meszaros

Of course, a big part of the value proposition is being able to run the suite of tests to avoid regression bugs. With these tests in place we can work fearlessly knowing that any mistakes we make will be caught by the tools. We won't have to spend a lot of effort double-checking everything we do.

And if we have tests that express our expectations of any suppliers' code, we'll be notified immediately if there was any changes in behavior in a new version of their software. We can even use tests to verify that our environment is set up the way our tests expect thus avoiding tests failing when they should pass. The tests truly do act as a safety net.

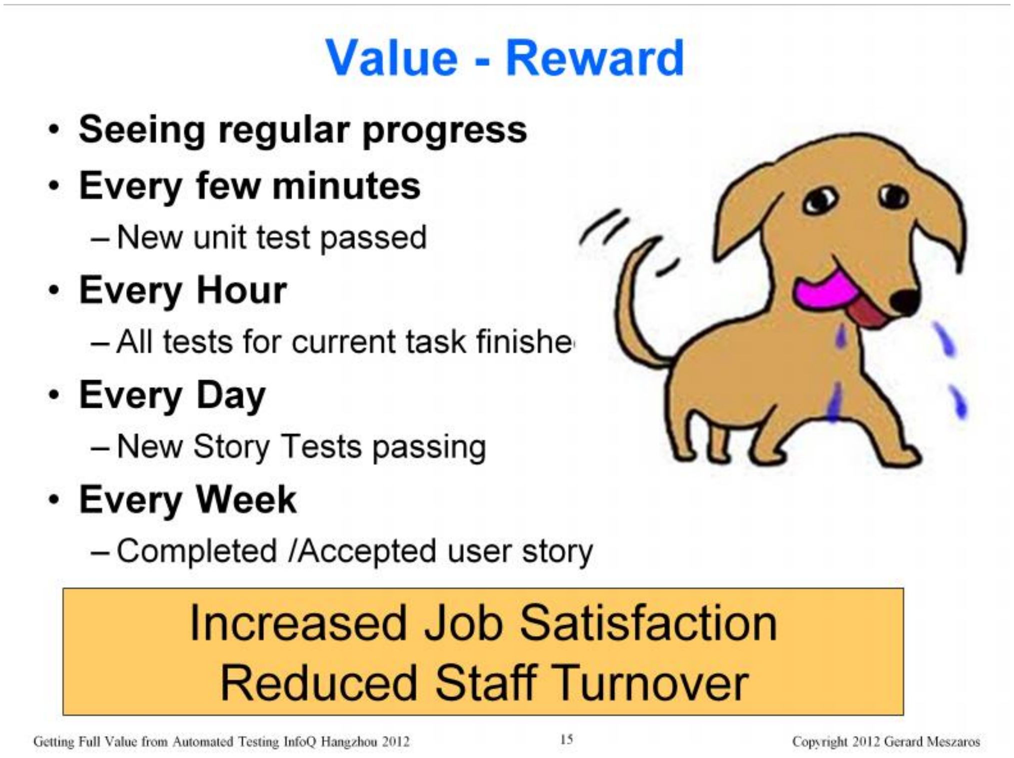


## Value - Confidence

- **Less paranoia (about impact of changes)**
- **Less Stress (worrying about negative impact)**
- **Less effort wasted on dealing with stress & paranoia**
  - We better make sure that …. doesn't ….
- **Allows a different way of working:**
  - Instead of reading lots of code to try to figure out what would happen if you changed the code,
  - just change the code and see what happens!

Getting Full Value from Automated Testing InfoQ Hangzhou 2012

The safety provided by the tests allow us to be less paranoid and work with less stress. This allows us to reduce the amount of effort we waste on "we better make sure that …" typedealing with paranoid concerns and stress. And it allows us to work in a more experimental fashion. Rather than analysing code to determine the impact if we changed it, we can simply make the change, run the tests and find out what changed. Because that's what our tests are: a huge change detector.



## Value - Reward

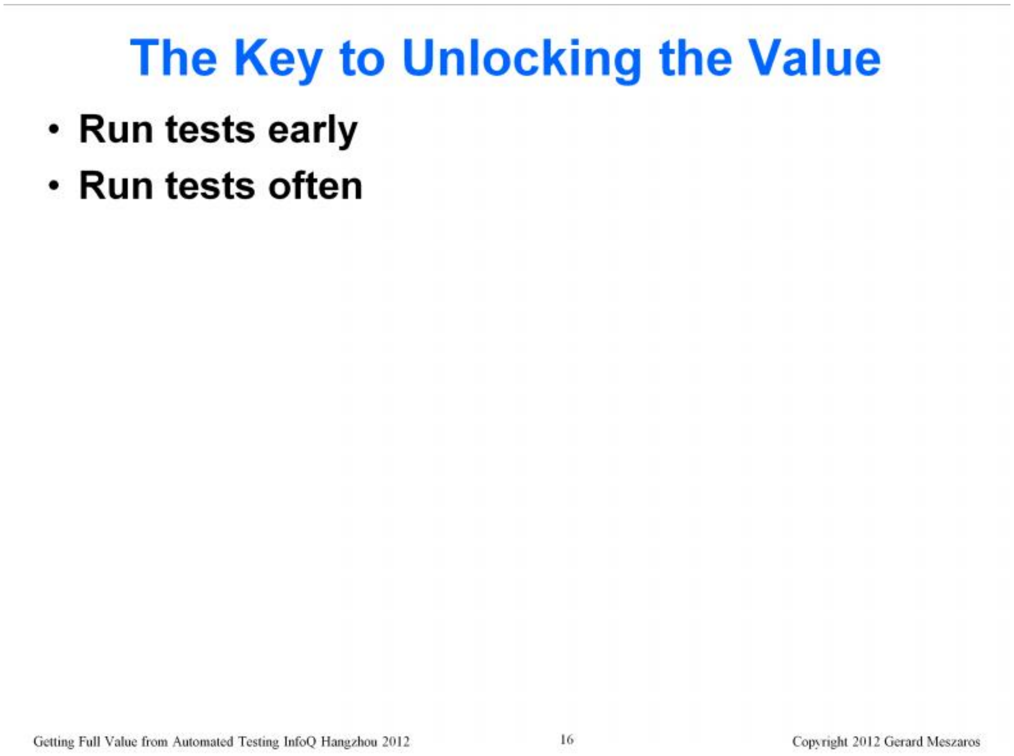
- **Seeing regular progress**
- **Every few minutes**
  - New unit test passed
- **Every Hour**
  - All tests for current task finishe
- **Every Day**
  - New Story Tests passing
- **Every Week**
  - Completed /Accepted user story

Increased Job Satisfaction
Reduced Staff Turnover

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 15 Copyright 2012 Gerard Meszaros

And that brings us to the least tangible but still very important benefit of having a suite of automated tests:

The tests make our work more rewarding by making our progress highly visible both to ourselves and our stakeholders. Every few minutes we get to see another unit test pass. Every hour, we can mark a task as completed. Every day we'll see new story tests passing and every week we can mark several more stories as done. It's hard to calculate how much value this kind of regular positive feedback gives us but it's sure to show up in <ANIMATE> job satisfaction ratings and reduced staff turnover.



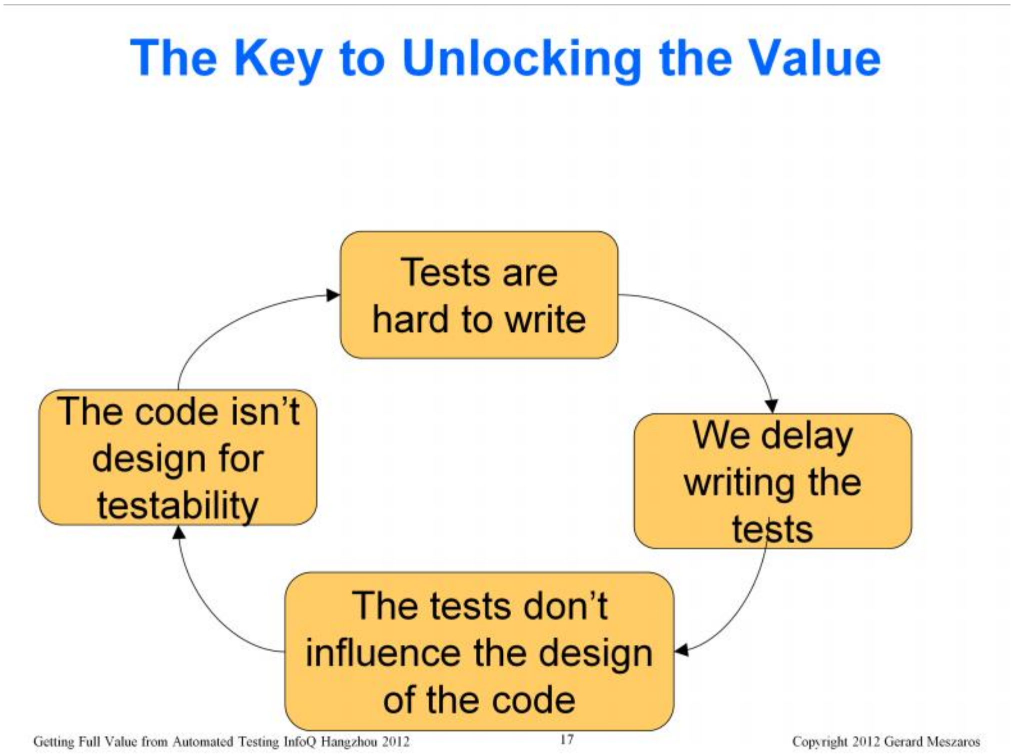
## The Key to Unlocking the Value

- Run tests early
- Run tests often

The key to unlocking all this value is to run your tests early and run them often.

Running them early means running them immediately after the code is written. And by immediately, I don’t mean a few days or weeks, I mean within seconds. The only practical way to achieve this is by writing the test first. Test-driving the code.

Running them often requires that the tests run quickly.



## The Key to Unlocking the Value

Tests are hard to write

We delay writing the tests

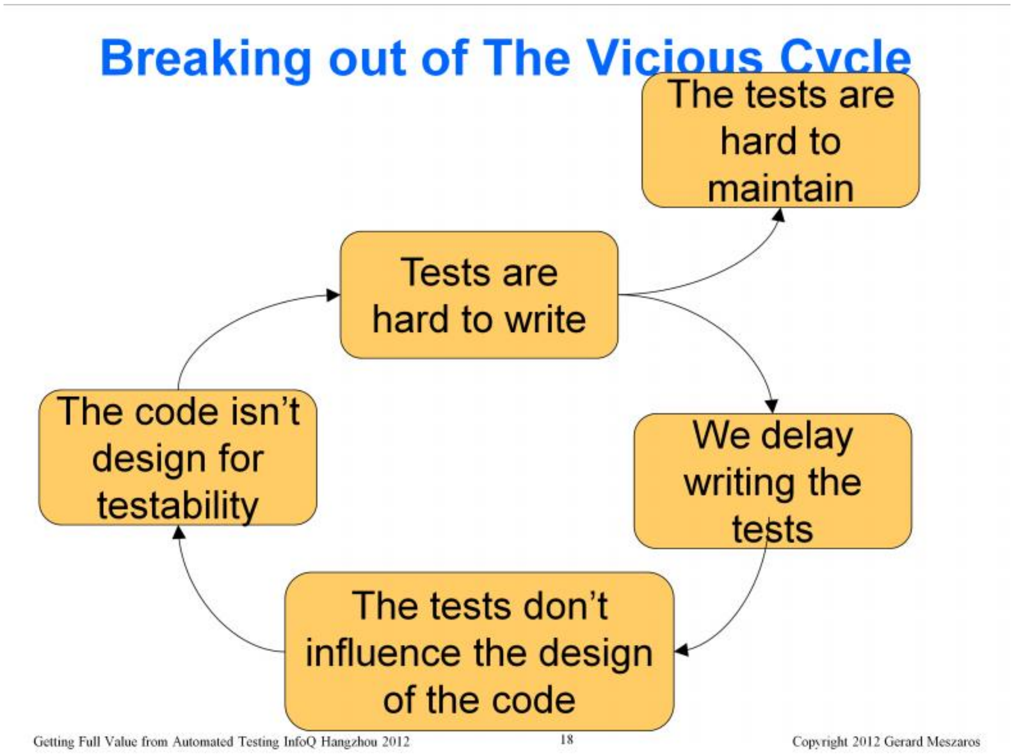
The tests don’t influence the design of the code

The code isn’t design for testability

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 17 Copyright 2012 Gerard Meszaros

Who here has this problem: The tests are hard to write? So we delay writing the tests (because it is hard and slows us down.) But this just ensures that the tests cannot influence the design of the code because they don’t exist before the code. And that ensures that the code jus tisn’t designed for testability which in turn makes it hard to write the tests.

Now, anyone who has studied “Systems Thinking” will recognize this as a self-amplifying phenonomumn. Or in plain language, a “Vicious Cycle”.



## Breaking out of The Vicious Cycle

The tests are hard to maintain

Tests are hard to write

We delay writing the tests

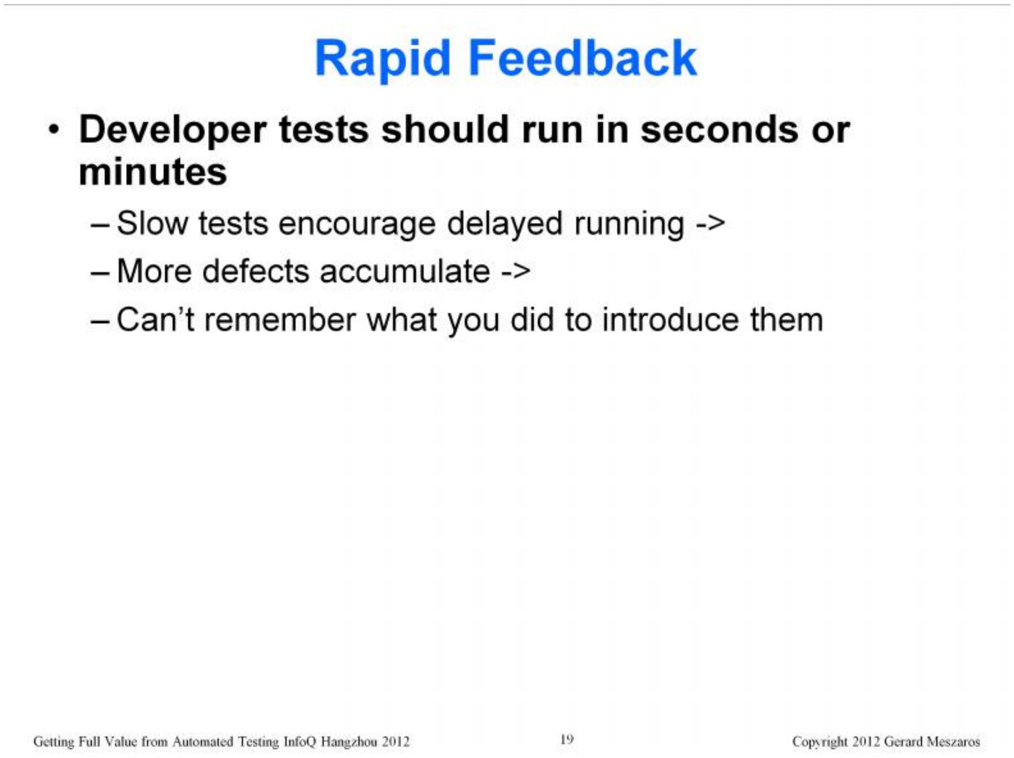
The tests don’t influence the design of the code

The code isn’t design for testability

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 18 Copyright 2012 Gerard Meszaros

Even worse, the are other things affected by tests being hard to write and that is that the tests are hard to maintain. So writing the tests late is a root cause for the cost of test automation being too high. The only way to break out of this vicious cycle is to attack the only one of these items we actually have direct control over: When we write the tests.

So we have to find a way to avoid delaying writing the tests so that each of things it causes are also reduced.

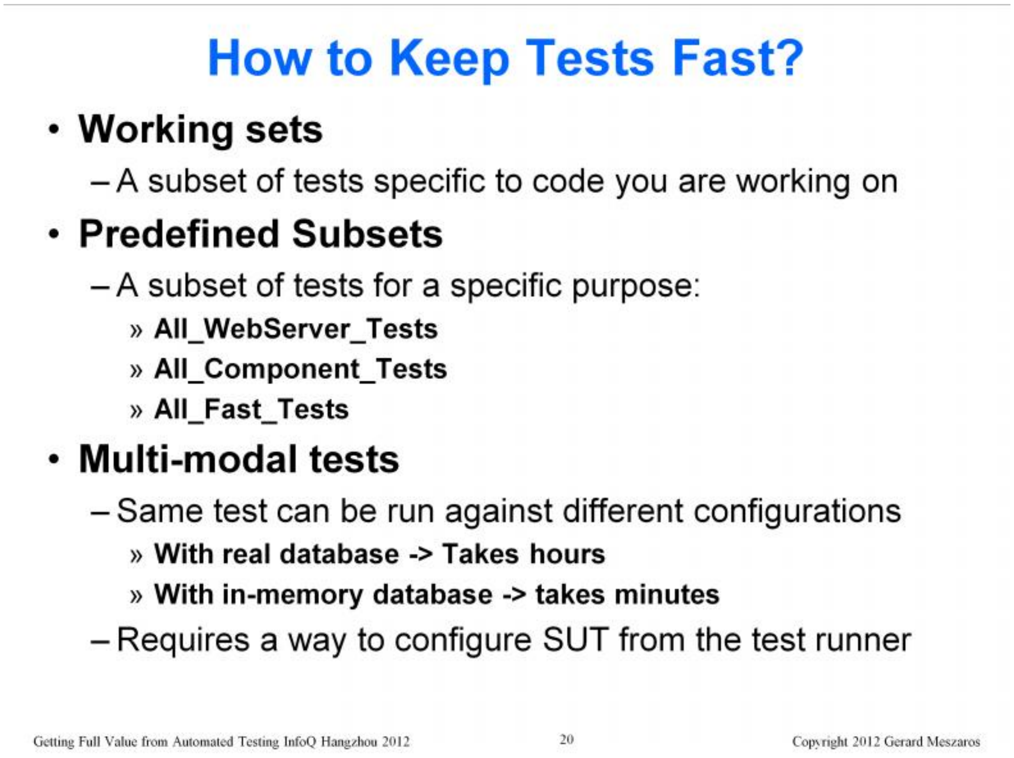


## Rapid Feedback

- **Developer tests should run in seconds or minutes**
  - Slow tests encourage delayed running ->
  - More defects accumulate ->
  - Can’t remember what you did to introduce them

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 19 Copyright 2012 Gerard Meszaros

A large part of the value with which the tests provide us is due to the rapid feedback they give us. It’s really important to keep that feedback loop as tight as possible. We have to be careful not to allow the tests to take too long to run. Slow test encourage people to work longer between test runs. That gives us more time to insert defects. And when the tests finally do get run, we won’t remember what they did to cause the defect because it was several hours ago. One team I worked with had a rule: if the acceptance tests took more than 15 minutes to run, they would put tasks into the backlog to get it back down under 10 minutes. They used all sorts of techniques to speed up the tests including optimizing the test fixture setup, eliminating duplicate test, buying faster hardware, etc.  Other people I know set up their IDE to automatically run the tests in the background after every save. They don’t even pay any attention to the tests running until a failed test  pops up a warning.



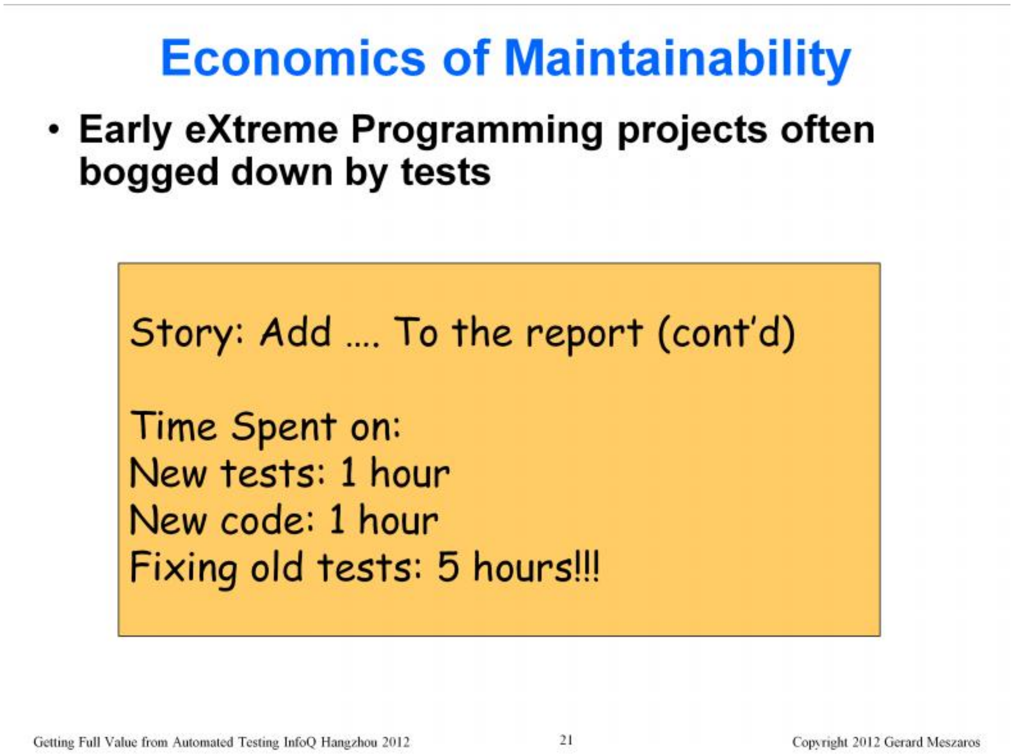
## How to Keep Tests Fast?

- **Working sets**
  - A subset of tests specific to code you are working on
- **Predefined Subsets**
  - A subset of tests for a specific purpose:
    - » **All_WebServer_Tests**
    - » **All_Component_Tests**
    - » **All_Fast_Tests**
- **Multi-modal tests**
  - Same test can be run against different configurations
    - » **With real database -> Takes hours**
    - » **With in-memory database -> takes minutes**
  - Requires a way to configure SUT from the test runner

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 20 Copyright 2012 Gerard Meszaros

Some other ways to keep the tests running fast including defining a temporary working set for the code your are working on now, predefined subsets for various parts of the system or various stages of the checking process, and multi-modal tests.

An example of a multi-modal test: On several projects, we had acceptance tests that did a lot of reading and writing from a database. These tests took quite a long time to run. We made it possible for the tests to replace the database with an in-memory fake database and this sped up our tests by 2 orders of magnitude. Test that took nearly an hour could be run in about a minute when the database was replaced by the fake.



## Economics of Maintainability

- Early eXtreme Programming projects often bogged down by tests

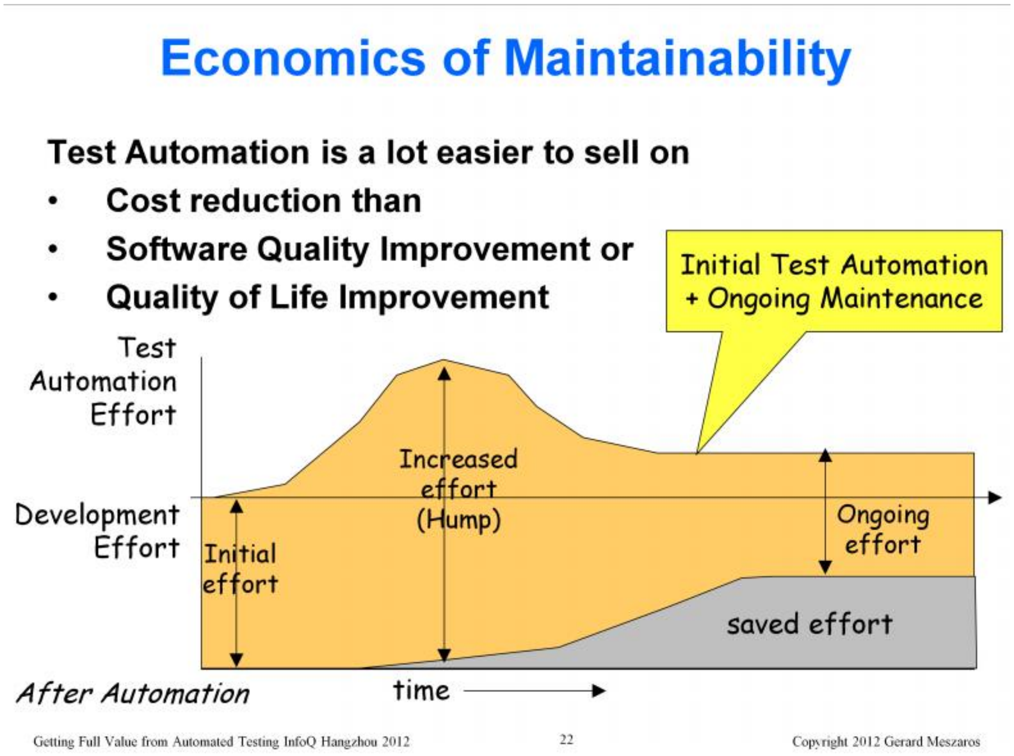
Story: Add …. To the report (cont'd)

Time Spent on:
New tests: 1 hour
New code: 1 hour
Fixing old tests: 5 hours!!!

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 21 Copyright 2012 Gerard Meszaros

On my first eXtreme Programming project, we were writing unit tests for all our code. It was taking longer and longer to implement each story. I asked everyone on the project to track how much time they spent on writing product code vs writing new tests vs. updating existing tests. The results were shocking! We were spending up to 90% of our time maintaining existing tests. So I set out to find out why and what we could do to reduce this. It turned out that the problem was in how we were coding our tests. They were not coded in a maintainable style.

So, why is maintainability so important? Let's take a quick look.



## Economics of Maintainability

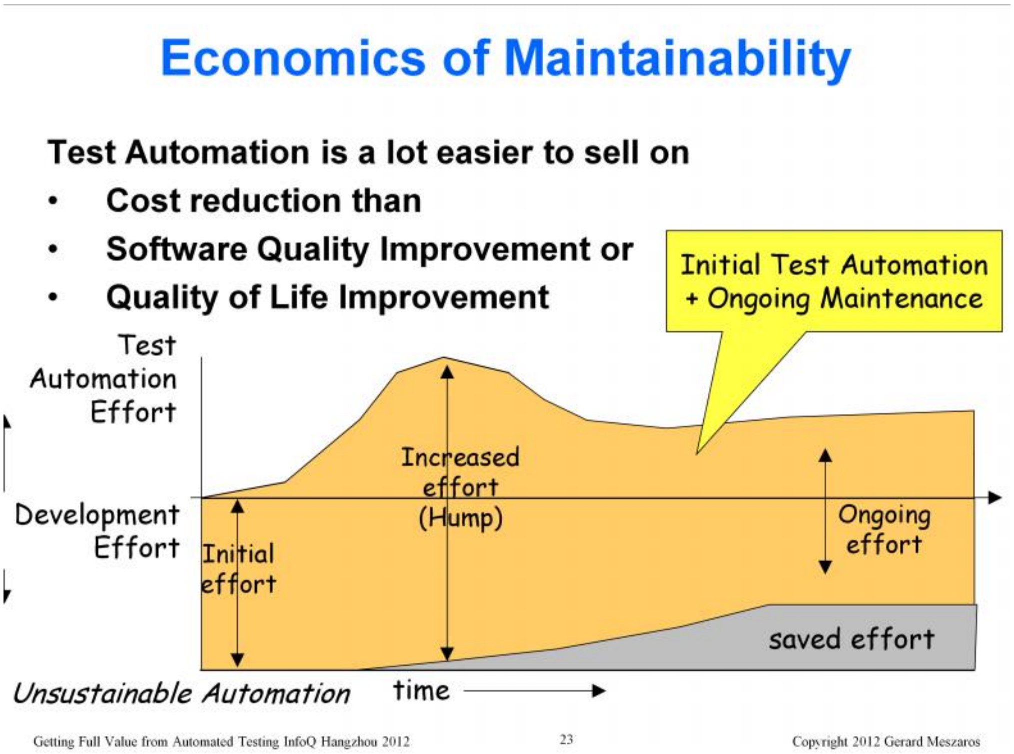
**Test Automation is a lot easier to sell on**

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

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 22 Copyright 2012 Gerard Meszaros

The area below the horizontal line is the amount of effort we spend on developing the potentially shippable production code.

The area above the line is the effort spent developing the automated test code.  Early in the automation the effort is significantly higher because we need to learn how to do the automation; we need to develop our reusable test infrastructure. As we get experience, the incremental cost of automating the tests is reduced and the savings increase. These savings come from a variety of sources but a large part is the effort saved by avoiding the debugging of code. (Ask your developers what percentage of time they spend debugging; it is usually between 50 and 80 %!)



## Economics of Maintainability

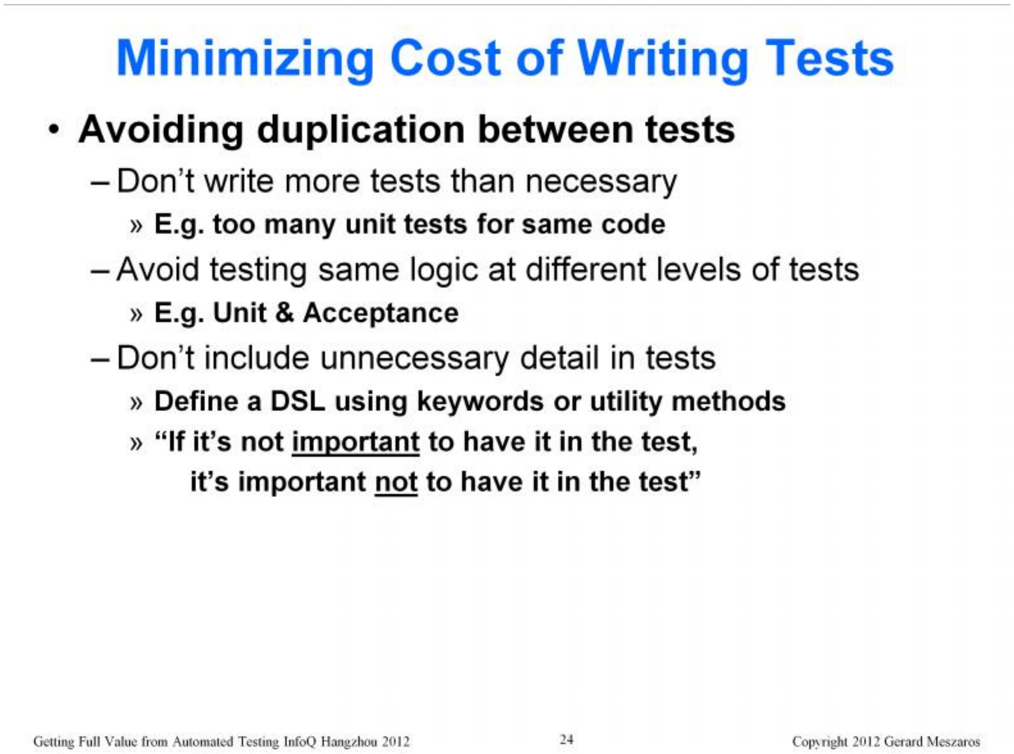
**Test Automation is a lot easier to sell on**

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

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 23 Copyright 2012 Gerard Meszaros

If we don’t pay enough attention to maintainability of the test code, the cost to write new tests or maintain existing tests increases significantly.

If we don’t automate our tests until after we’ve debugged the code then our cost savings drops . Either of these can make our overall cost higher than without test automation. At this point we would have to justify the extra cost based on improved quality; a much harder sell than a net increase in productivity!



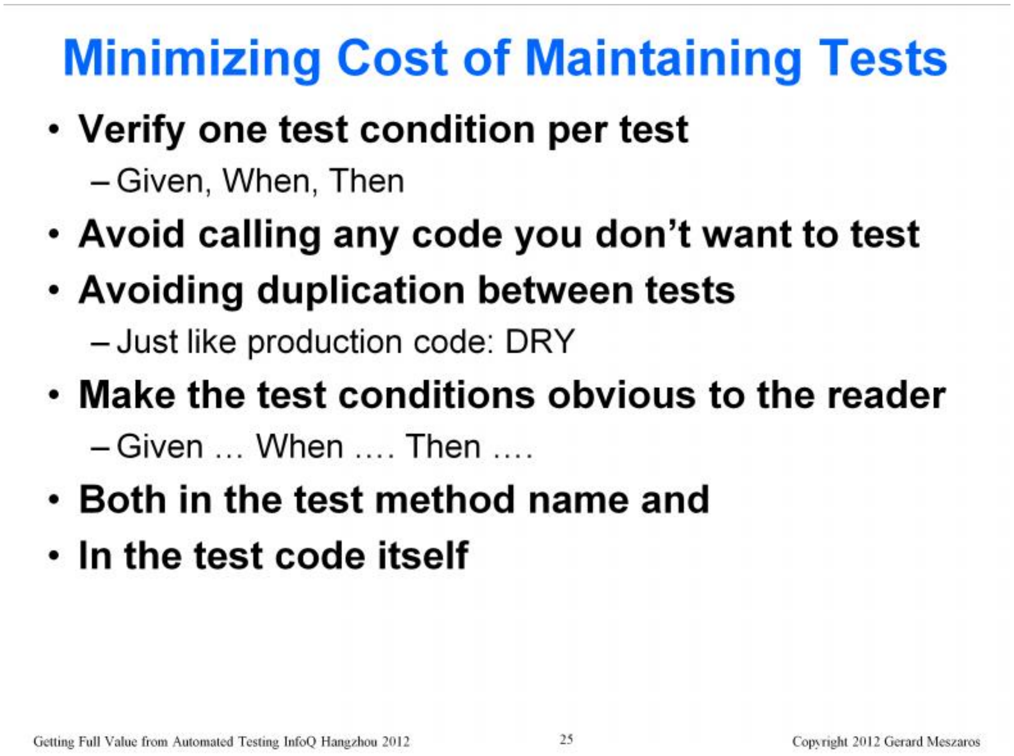
## Minimizing Cost of Writing Tests

- **Avoiding duplication between tests**
  - Don’t write more tests than necessary
    - » **E.g. too many unit tests for same code**
  - Avoid testing same logic at different levels of tests
    - » **E.g. Unit & Acceptance**
  - Don’t include unnecessary detail in tests
    - » **Define a DSL using keywords or utility methods**
    - » **“If it’s not important to have it in the test, it’s important not to have it in the test”**

Earlier, I said that we want to minimize the cost of writing tests. Some ways to do this include:

Avoiding writing any more tests than needed, avoiding overlap between tests at one level vs another level (e.g. unit tests vs acceptance tests)

Avoiding unnecessary detail in tests

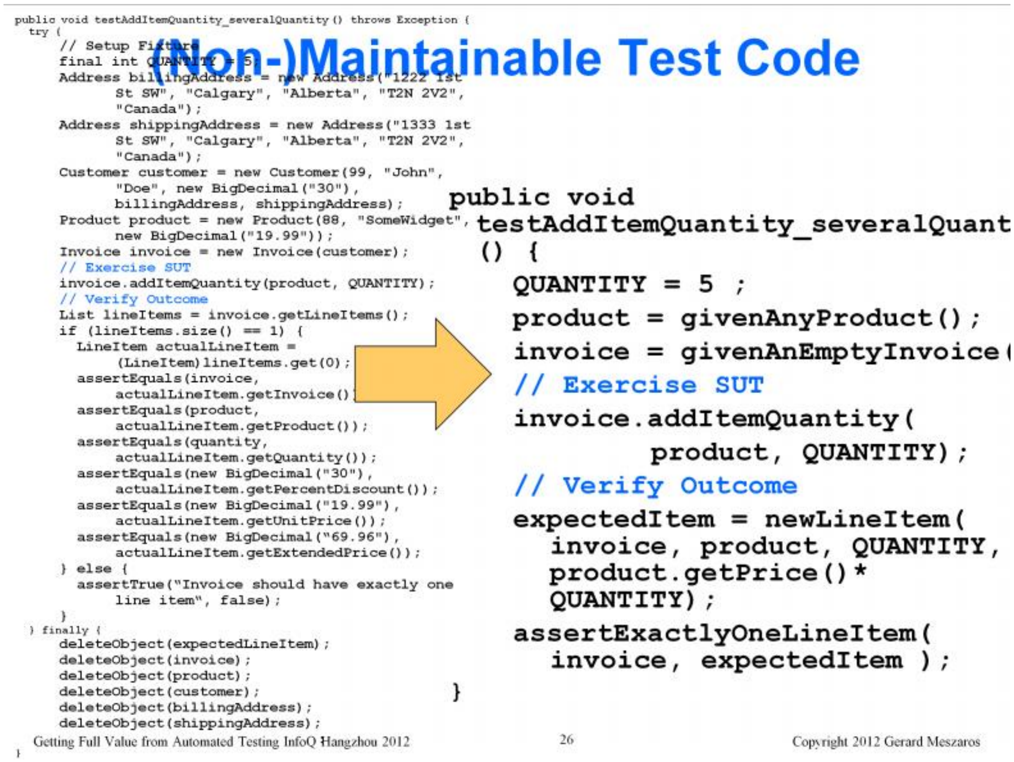


## Minimizing Cost of Maintaining Tests

- **Verify one test condition per test**
  - Given, When, Then
- **Avoid calling any code you don't want to test**
- **Avoiding duplication between tests**
  - Just like production code: DRY
- **Make the test conditions obvious to the reader**
  - Given … When …. Then ….
- **Both in the test method name and**
- **In the test code itself**

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 25 Copyright 2012 Gerard Meszaros

When we are reading tests to understand the code or because we need to update the tests because the code has to be changed, they are a lot easier to understand if we only verify a single test condition in each test. The Given, when, Then format helps make this clear.  We want to avoid using any code that we don't want to test because it can cause our tests to fail for unrelated reasons. We don't want to have to change the same code in several different tests so we factor out duplicate code into utility methods. We structure our code and name our test methods and classes to make the test conditions obvious.



## (Non-)Maintainable Test Code

```
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);
}
```

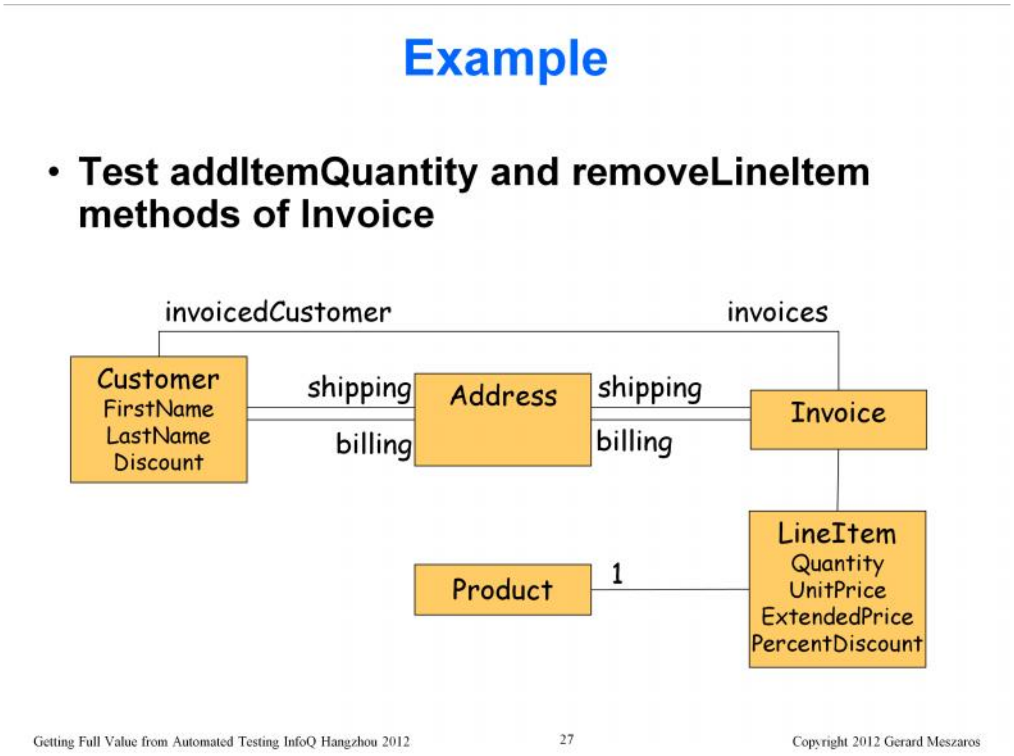
```
public void
 testAddItemQuantity_severalQuant
 () {
   QUANTITY = 5 ;
   product = givenAnyProduct();
   invoice = givenAnEmptyInvoice(
   // Exercise SUT
   invoice.addItemQuantity(
            product, QUANTITY);
   // Verify Outcome
   expectedItem = newLineItem(
     invoice, product, QUANTITY,
     product.getPrice()*
     QUANTITY);
   assertExactlyOneLineItem(
     invoice, expectedItem );
}
```

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 26 Copyright 2012 Gerard Meszaros

Here's an example of what a test might look like if the writer isn't focused on maintainability. The test is long, verbose, and hard to understand. This test is also hard to maintain; if we need to change it because we are changing the code it verifies, changing it will take longer. Complex tests such as this make our software less "soft"!

The test on the right focuses on the essence of the requirement. Imagine how much less time it would take to write a test in this format than the format on the left!! So how can you learn to write tests this simple? Let's work throw an example of how we can simplify complex tests we have already written.

(BTW, I travel around the world to train developers how to write tests such as the one on the right.)



## Example

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

invoicedCustomer

invoices

Customer
FirstName
LastName
Discount

shipping

billing

Address

shipping

billing

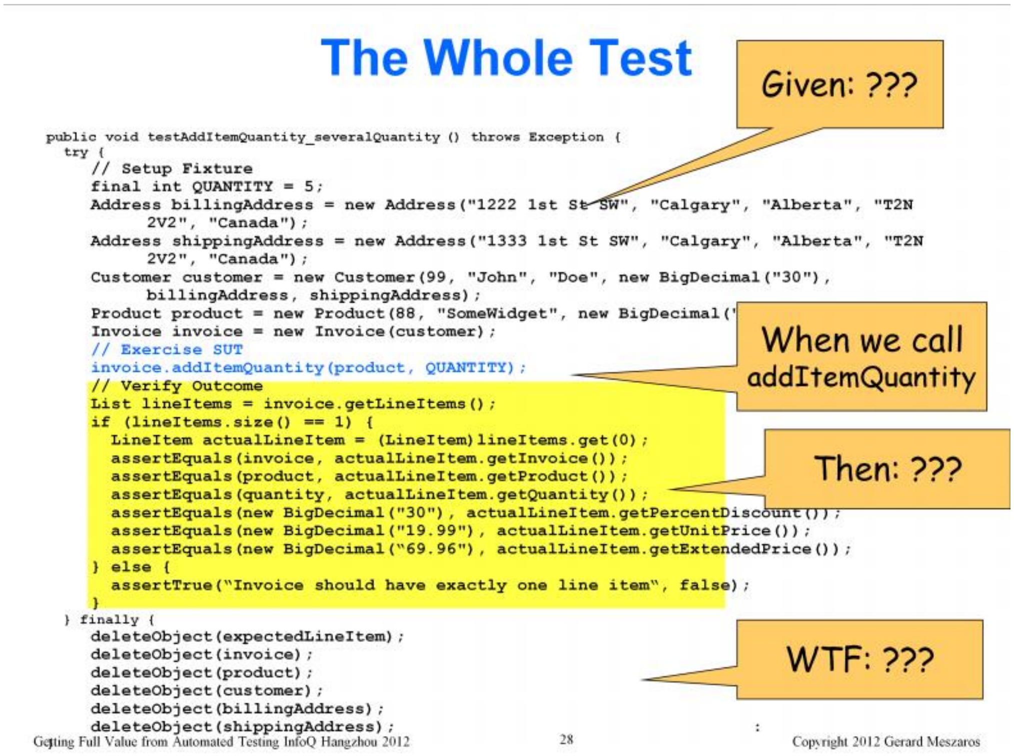
Invoice

LineItem
Quantity
UnitPrice
ExtendedPrice
PercentDiscount

Product

1

Here’s our requirement: We are testing the a method on the Invoice class but we cannot create an invoice without a customer and a shipping and billing address. We will be adding LineItems to the invoice and each LineItem has exactly one product.



## 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("
    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);
```

Given: ???

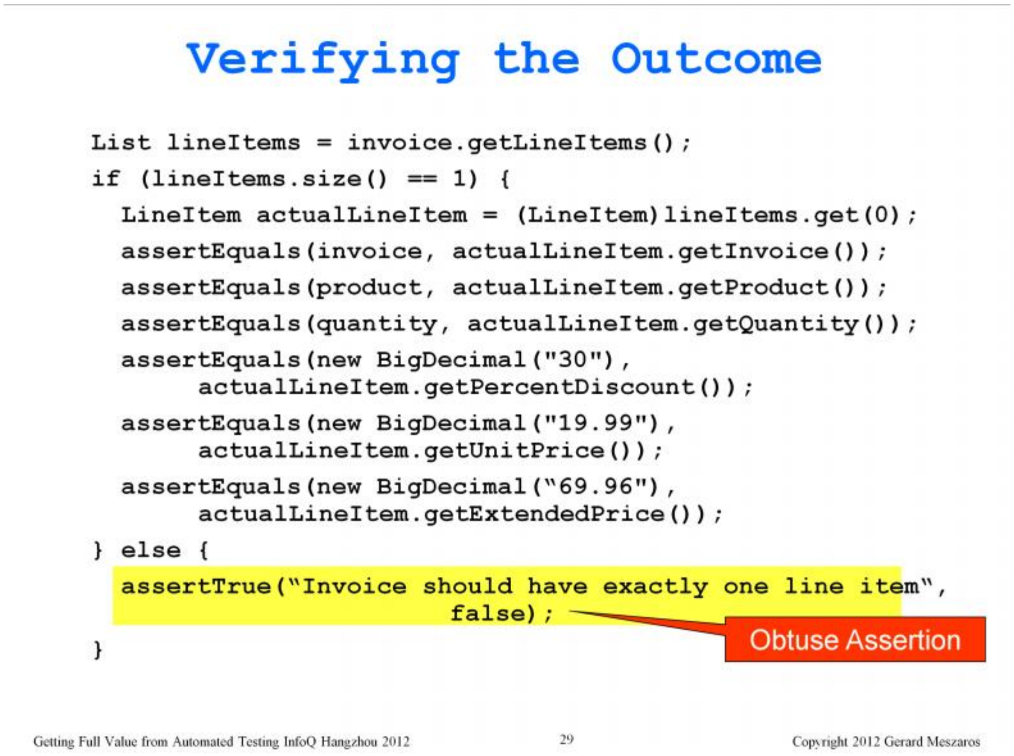
When we call addItemQuantity

Then: ???

WTF: ???

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 28 Copyright 2012 Gerard Meszaros

Here’s one test case for the method addItemQuantity() on the Invoice class. I apologize for the small font size but this test is pretty typical of the tests I see many developers writing. Can you even recognize the Given, When and Then parts of the test? Can you summarize them for me?



## Verifying the Outcome

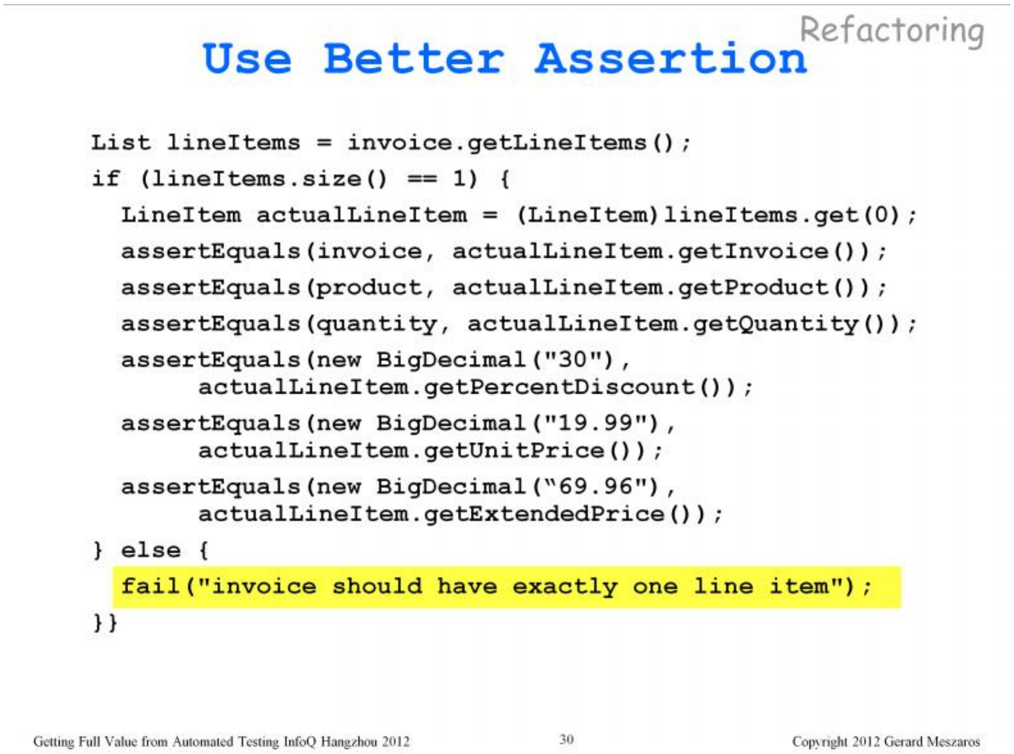
```
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
  LineItem actualLineItem = (LineItem)lineItems.get(0);
  assertEquals(invoice, actualLineItem.getInvoice());
  assertEquals(product, actualLineItem.getProduct());
  assertEquals(quantity, actualLineItem.getQuantity());
  assertEquals(new 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

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 29 Copyright 2012 Gerard Meszaros

Let’s focus in on the part of the test that verifies the outcome was correct. The part that specifies the “Then”. This is non-trivial! How do we start?

One piece of low-hanging fruit is the last assertion. What does assertTrue(… False) mean? This is an example of an Obtuse Assertion. Let’s clean this up by …



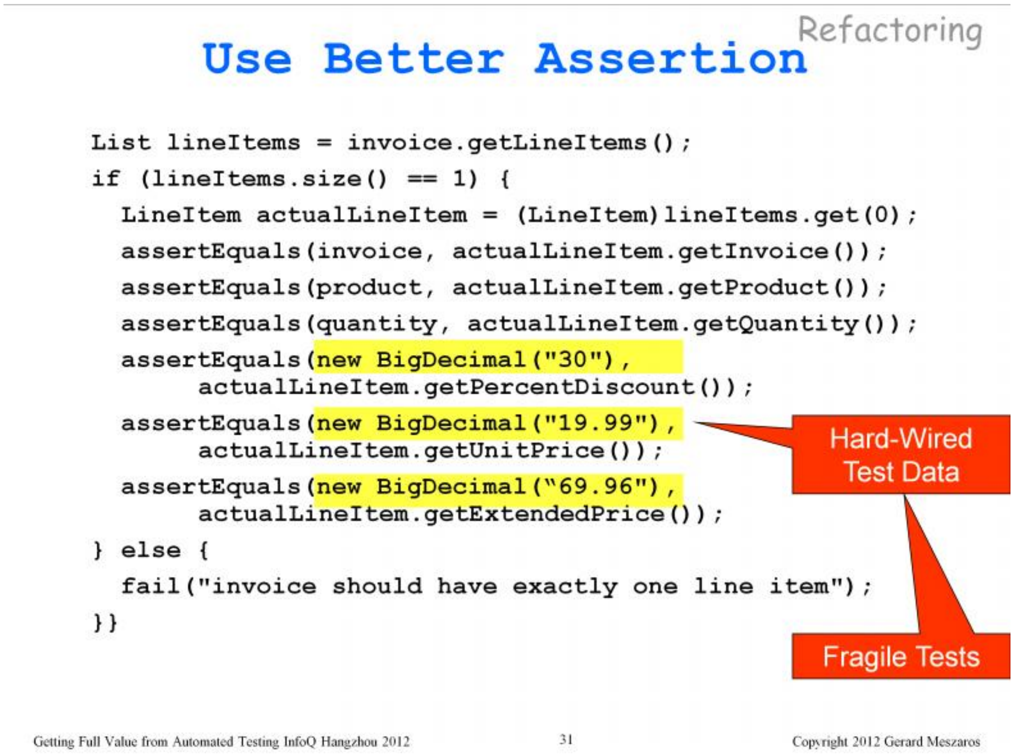
Refactoring

# 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");
}}
```

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 30 Copyright 2012 Gerard Meszaros

Replacing it with a better assertion. AssertTrue ( False) always fails, so let’s just call fail( ) instead.!



Refactoring

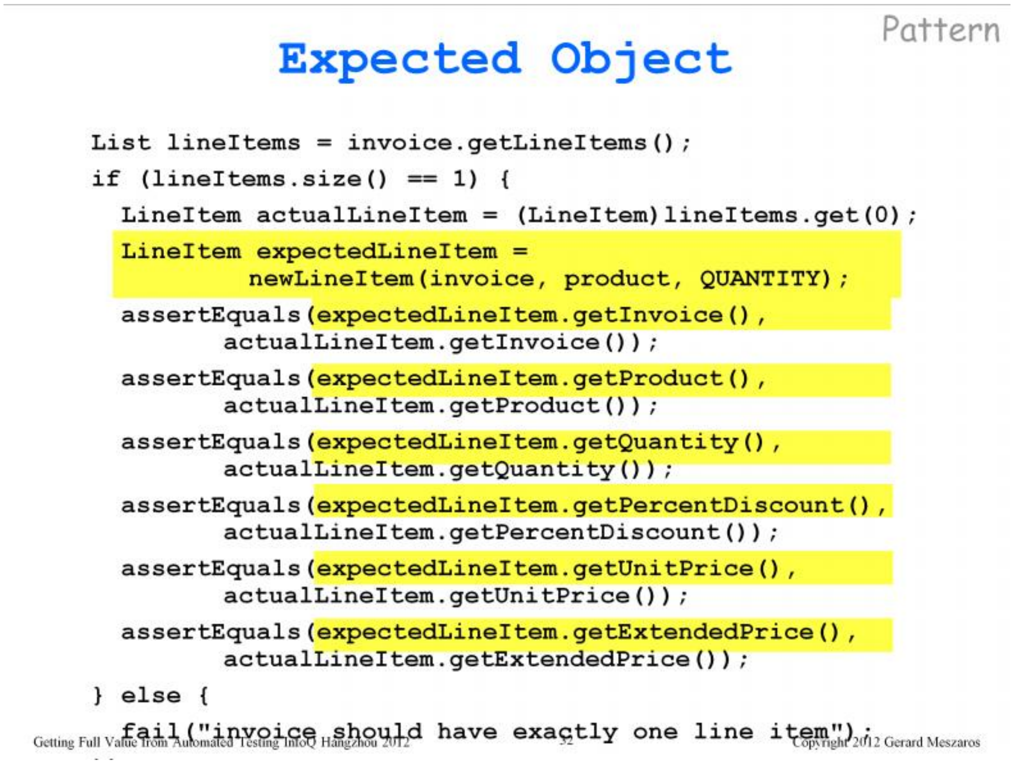
## Use Better Assertion

```
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

What is the significance of all this hard-code values. They make our test hard to understand and can also lead to fragile tests.



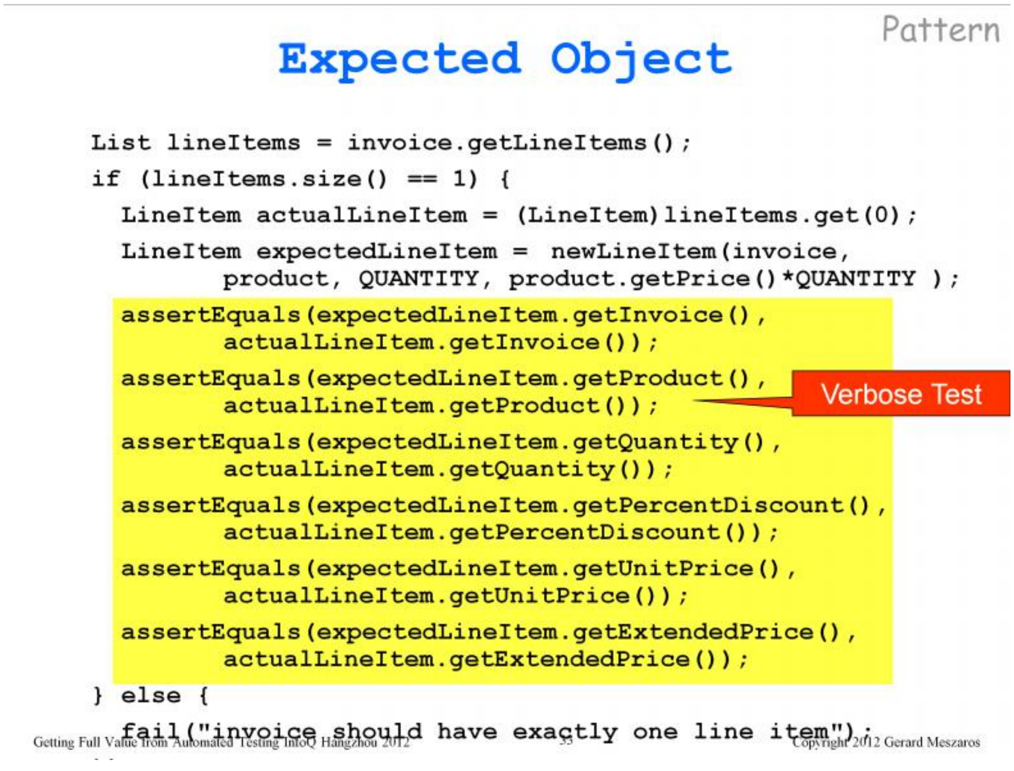
Pattern

## 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");
```

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 32 Copyright 2012 Gerard Meszaros

A better approach is to compare the actual results with an Expected Object that contains the expected results like this. But this is a lot of code to say “assert these two LineItems are equivalent”.



Pattern

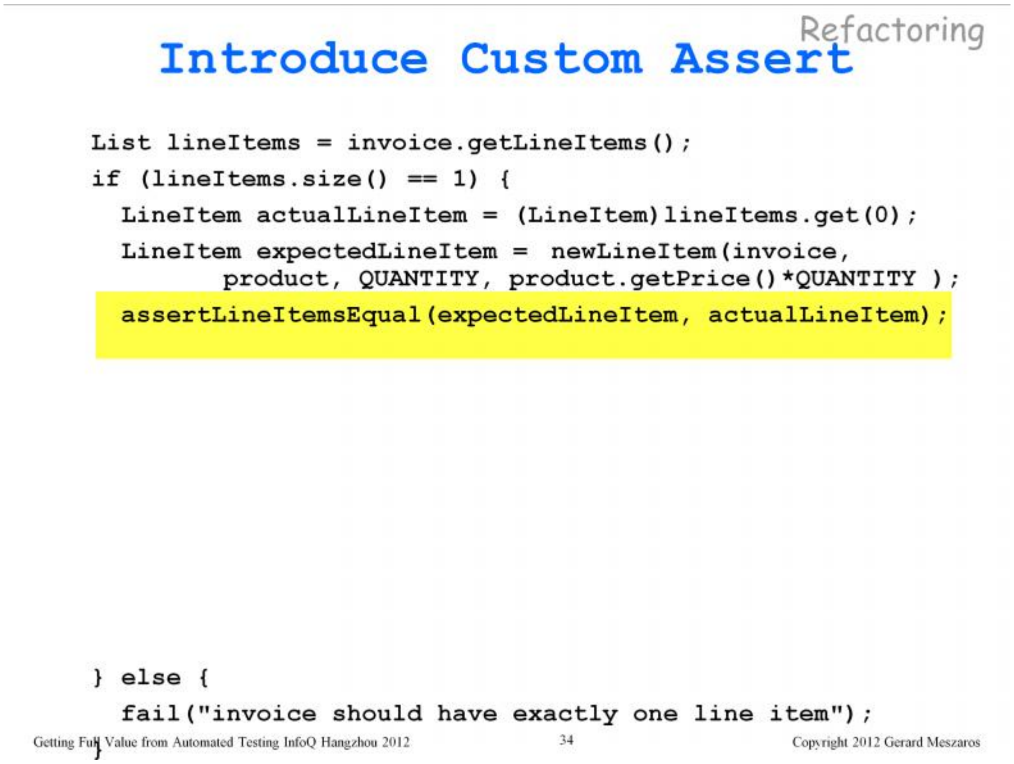
## Expected Object

```
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

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 33 Copyright 2012 Gerard Meszaros

We can reduce the verbosity of this test by ..

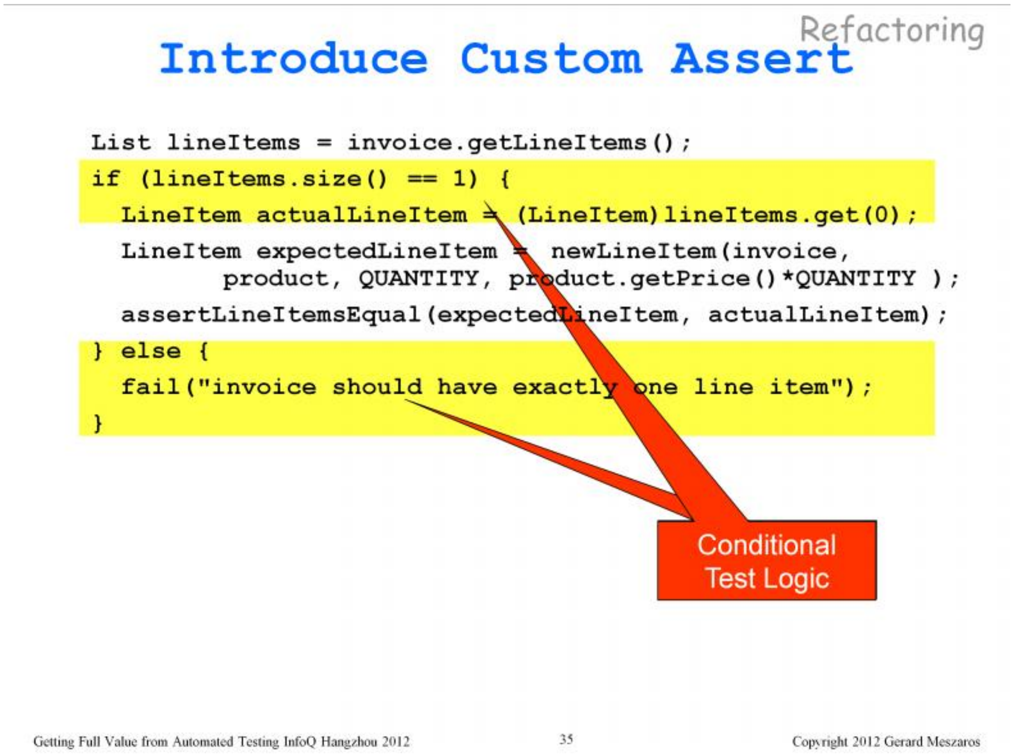


Refactoring

## 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");
}
```

Doing an Extract Method refactoring to create a Custom Assertion called assertLineItemsEqual.



Refactoring

## Introduce Custom Assert

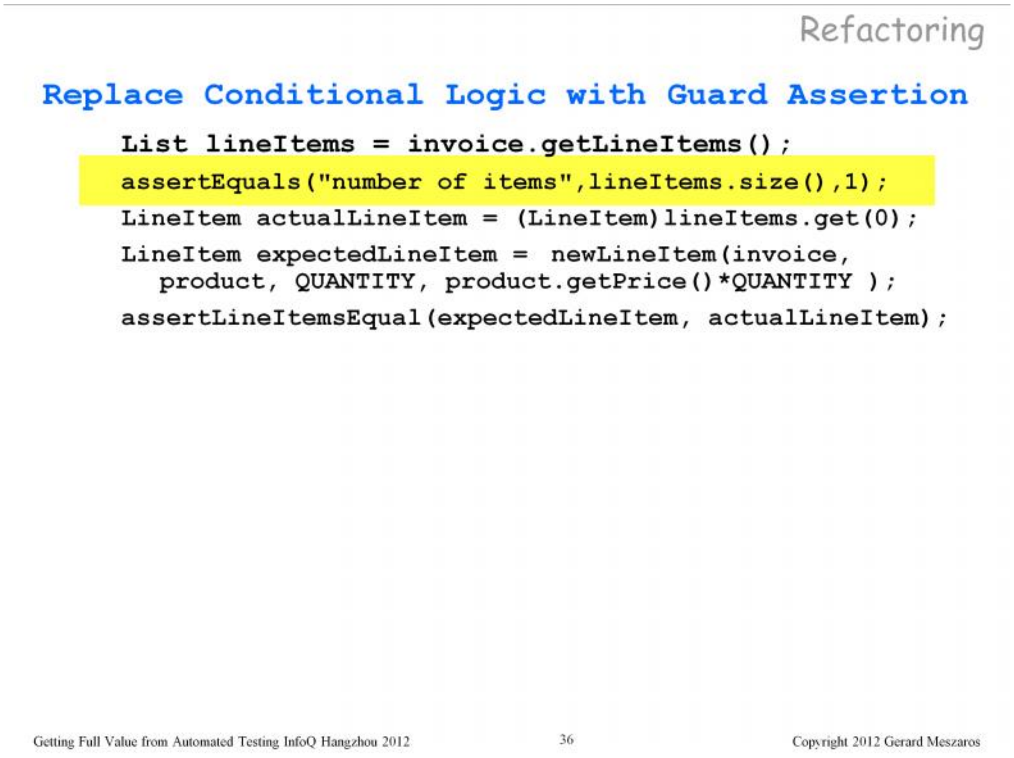
```
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
  LineItem actualLineItem = (LineItem)lineItems.get(0);
  LineItem expectedLineItem = newLineItem(invoice,
         product, QUANTITY, product.getPrice()*QUANTITY );
  assertLineItemsEqual(expectedLineItem, actualLineItem);
} else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

Conditional Test Logic

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 35 Copyright 2012 Gerard Meszaros

That reduce the amount of code in the test significantly. Now what else is wrong with this code?

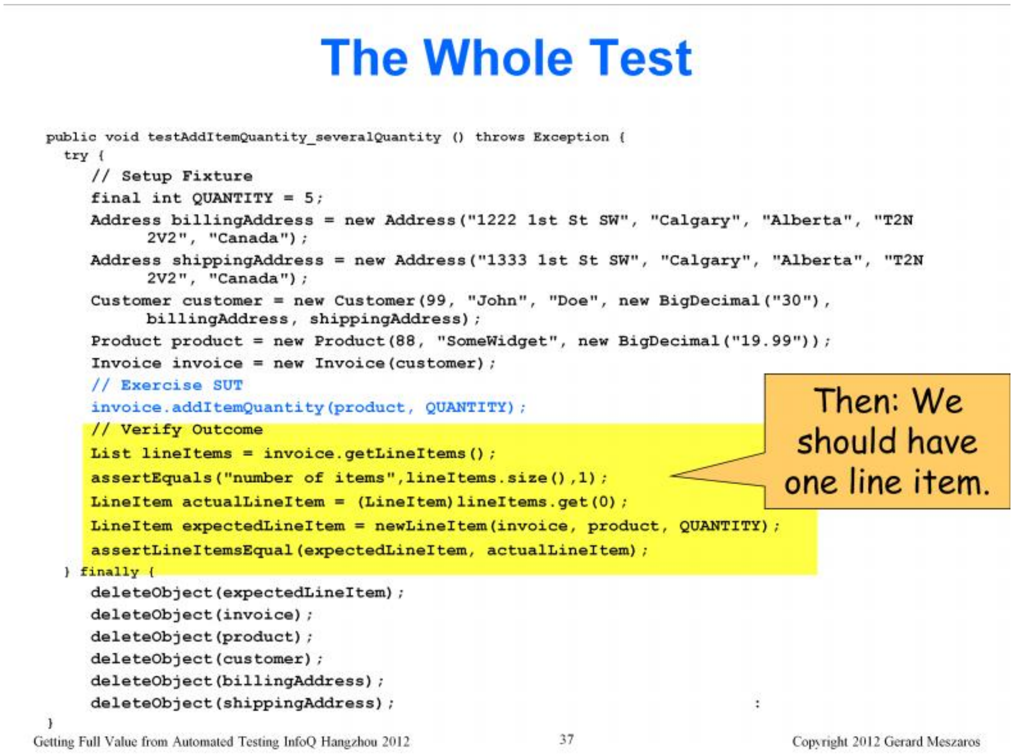
Well, it contains conditional logic in the form of an IF statement. This is bad because we can’t be sure which path is being executed; tests are easier to understand if they are purely sequential. Luckily, we can express the same thing more clearly by …



## 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);
```

Replacing the conditional logic with a Guard Assertion. Because a failed assertion transfers control back to the test runner, we won’t execute the next statement if the number of lineItems is wrong.



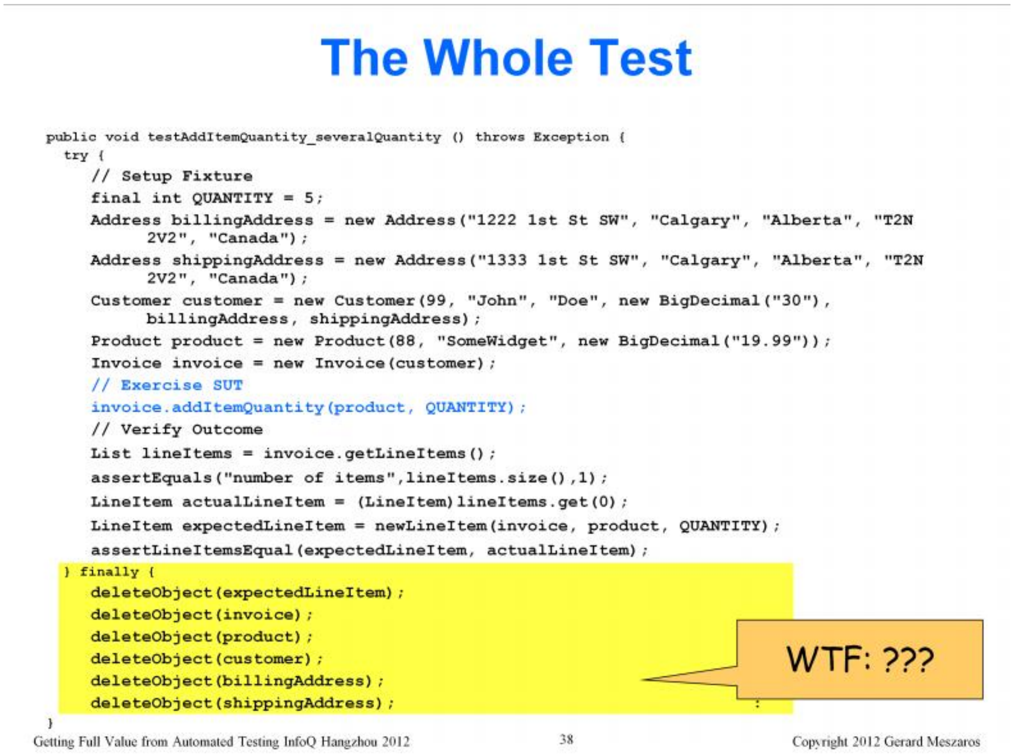
## The Whole Test

```
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);
}
```

Then: We should have one line item.

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 37 Copyright 2012 Gerard Meszaros

So back to the whole test which is a bit more readable now.



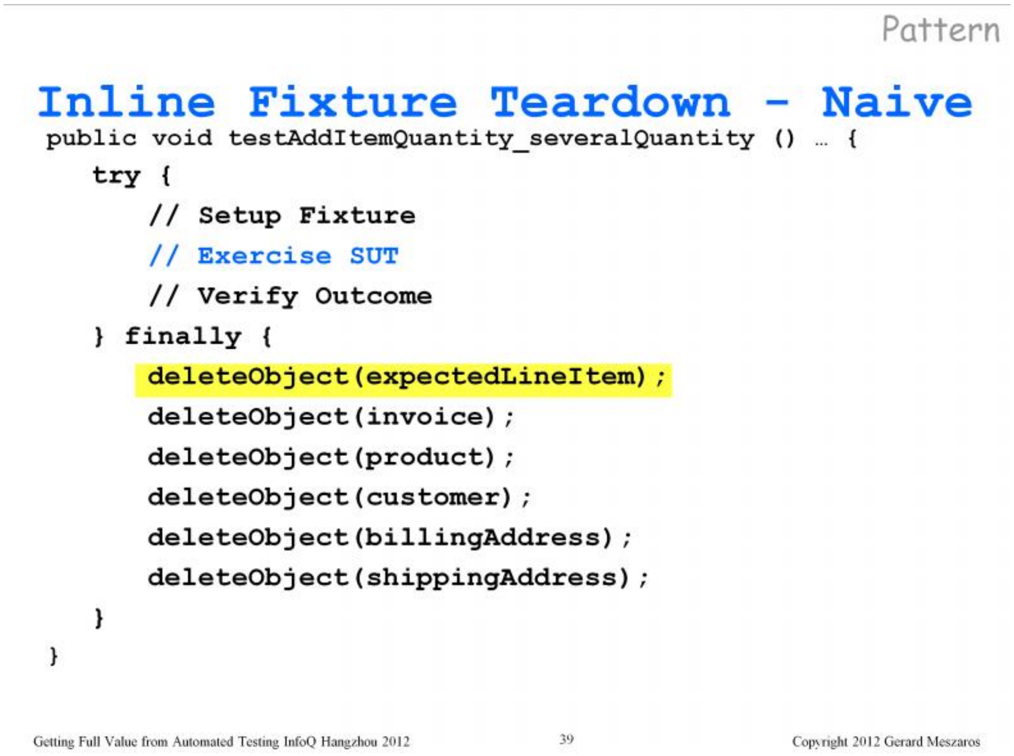
## The Whole Test

```
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);
}
```

WTF: ???

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 38 Copyright 2012 Gerard Meszaros

Let’s look at this code at the end; what does it do?



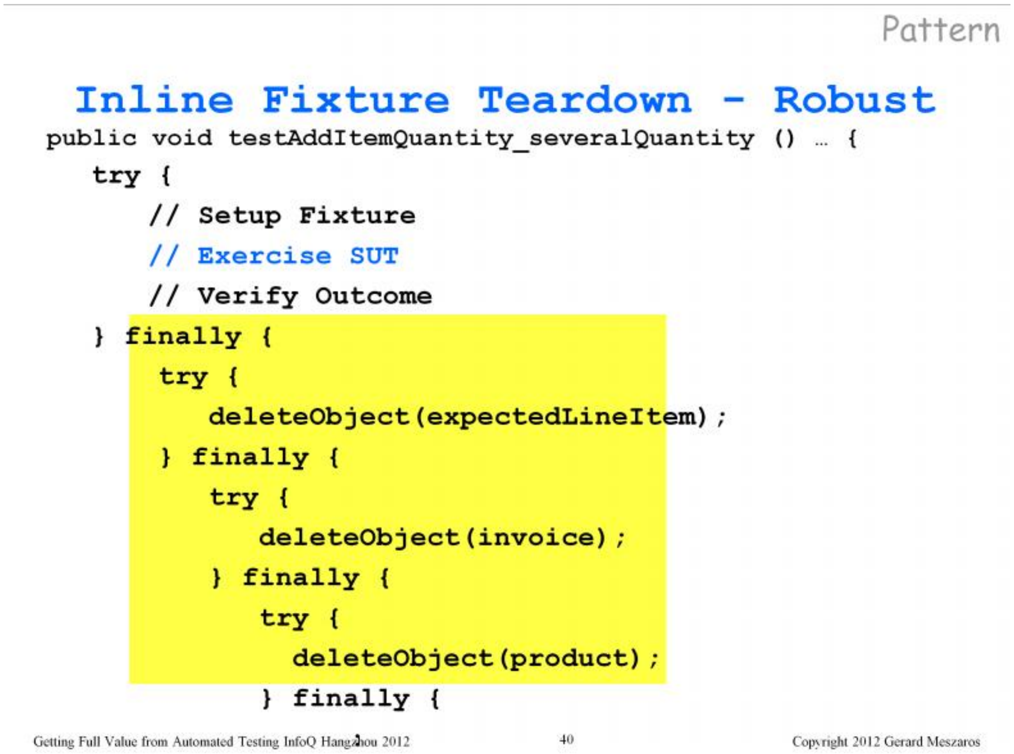
Pattern

## Inline Fixture Teardown - Naive

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

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 39

Oh, it’s doing teardown of the test. The hint is the Finally statement which ensures this code gets run whether the rest of the code runs clean or throws an exception. But this teardown code could fail. Then what happens? That’s right, the rest of the teardown code won’t run.

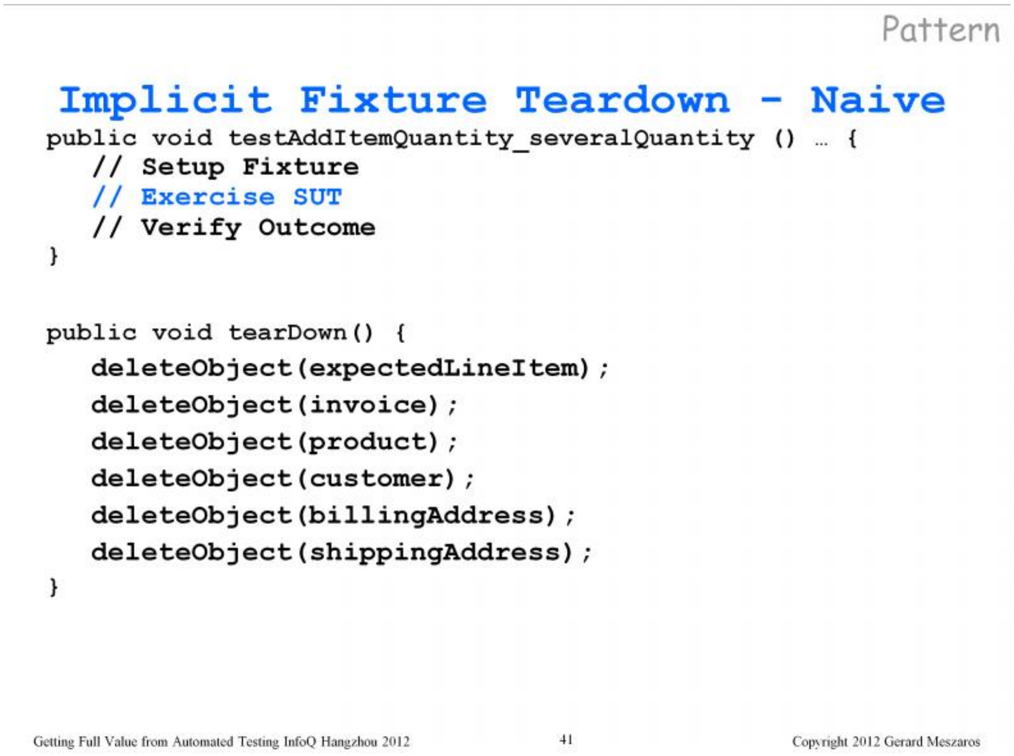


Pattern

## Inline Fixture Teardown - Robust

```
public void testAddItemQuantity_severalQuantity () … {
   try {
      // Setup Fixture
      // Exercise SUT
      // Verify Outcome
   } finally {
       try {
          deleteObject(expectedLineItem);
       } finally {
          try {
             deleteObject(invoice);
          } finally {
             try {
               deleteObject(product);
             } finally {
```

To ensure that it does, we’d have to code it like this.

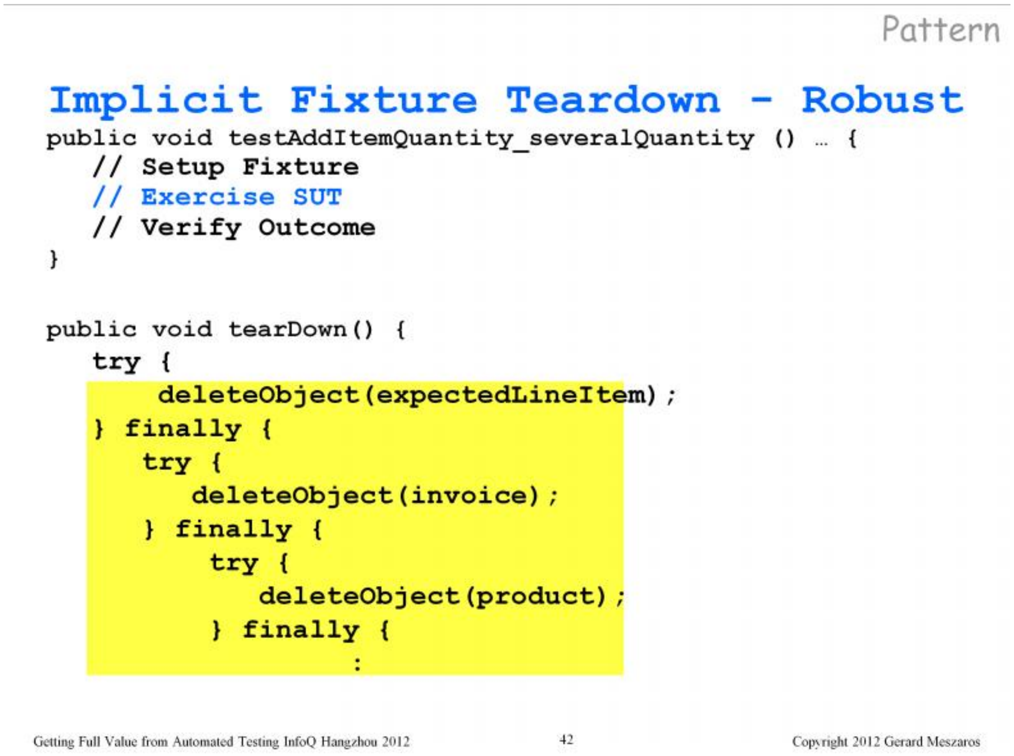


Pattern

## Implicit Fixture Teardown - Naive

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

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

Another option is to move it into the tearDown method but we have the same problem here.



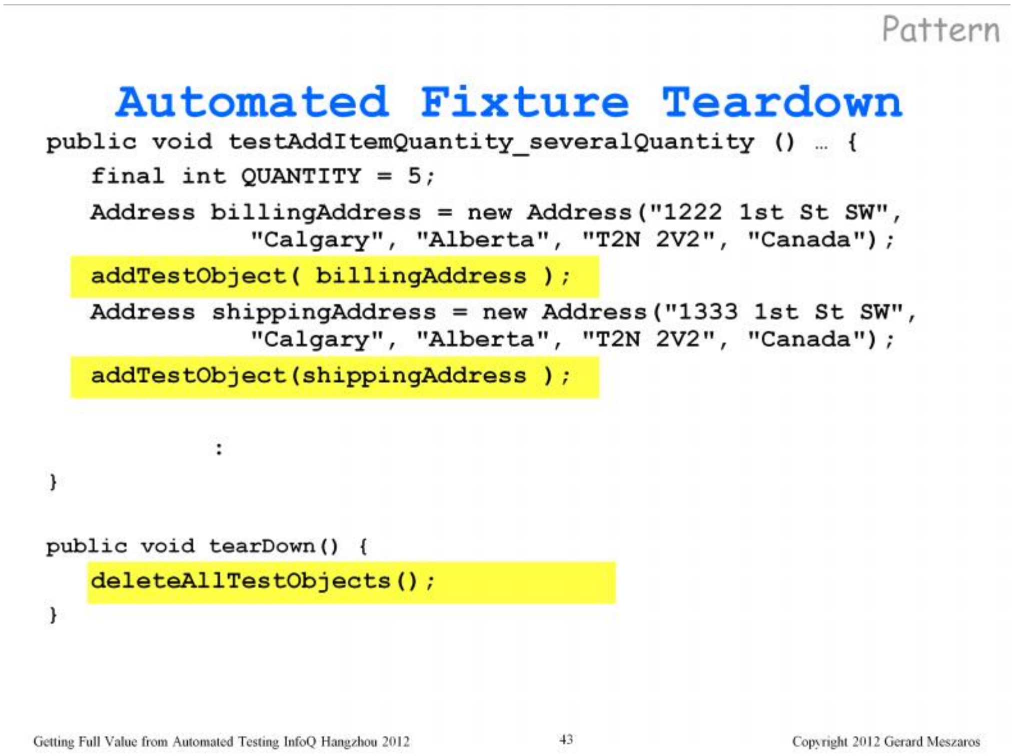
Pattern

## Implicit Fixture Teardown - Robust

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

public void tearDown() {
   try {
       deleteObject(expectedLineItem);
   } finally {
      try {
         deleteObject(invoice);
      } finally {
         try {
            deleteObject(product);
         } finally {
                  :
```

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 42 Copyright 2012 Gerard Meszaros

So we need to use nested Try/Finally’s here too.



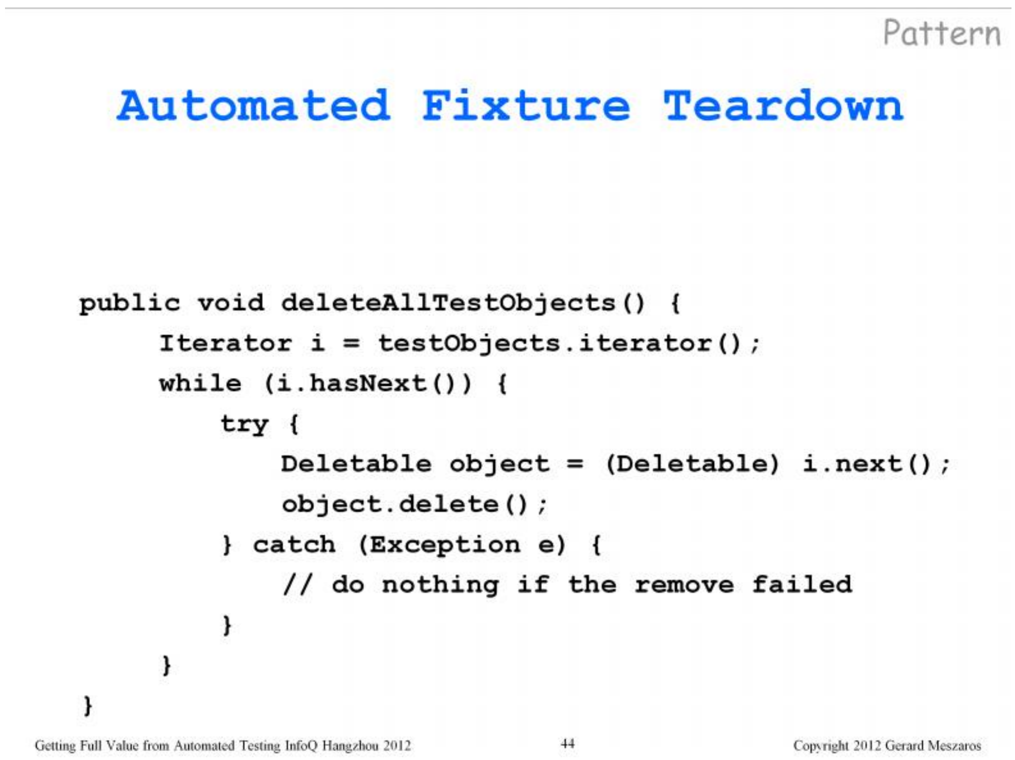
Pattern

# 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();
}
```

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 43 Copyright 2012 Gerard Meszaros

A better alternative is to register each object we create and then use a well-tested utility method to do the teardown for us.

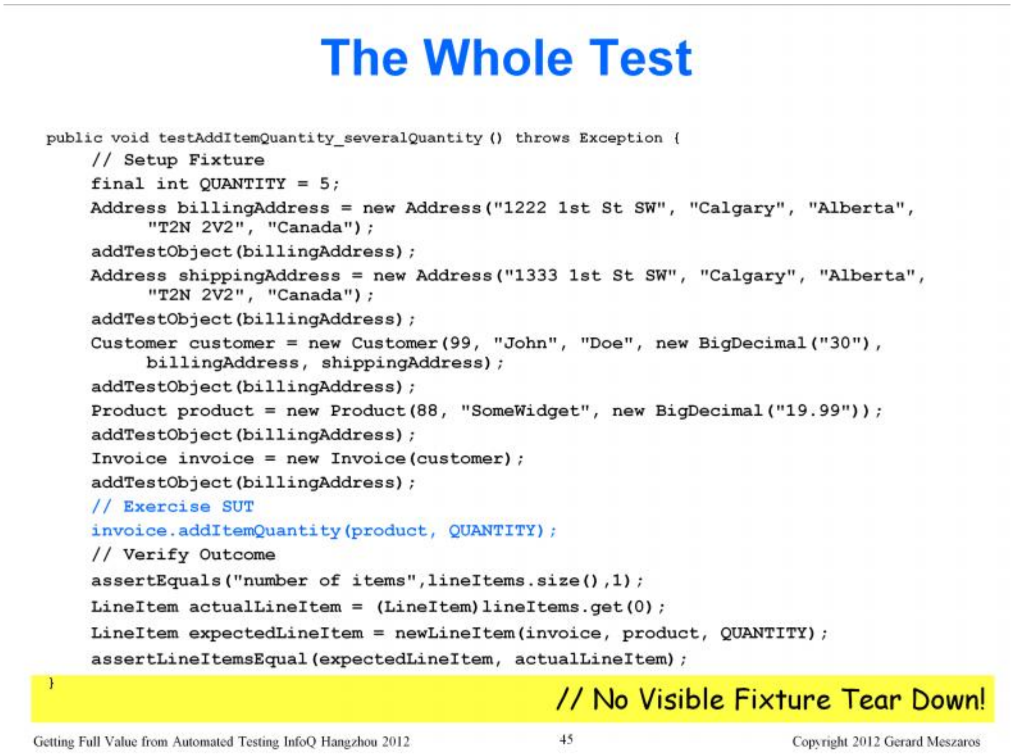


Pattern

# 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
         }
     }
}
```

This method simply iterates through the list of objects and tries deleting each one. If delete() throws an exception, it catches it and continues with the next object. This guarantees that all the objects will have delete() called on them.



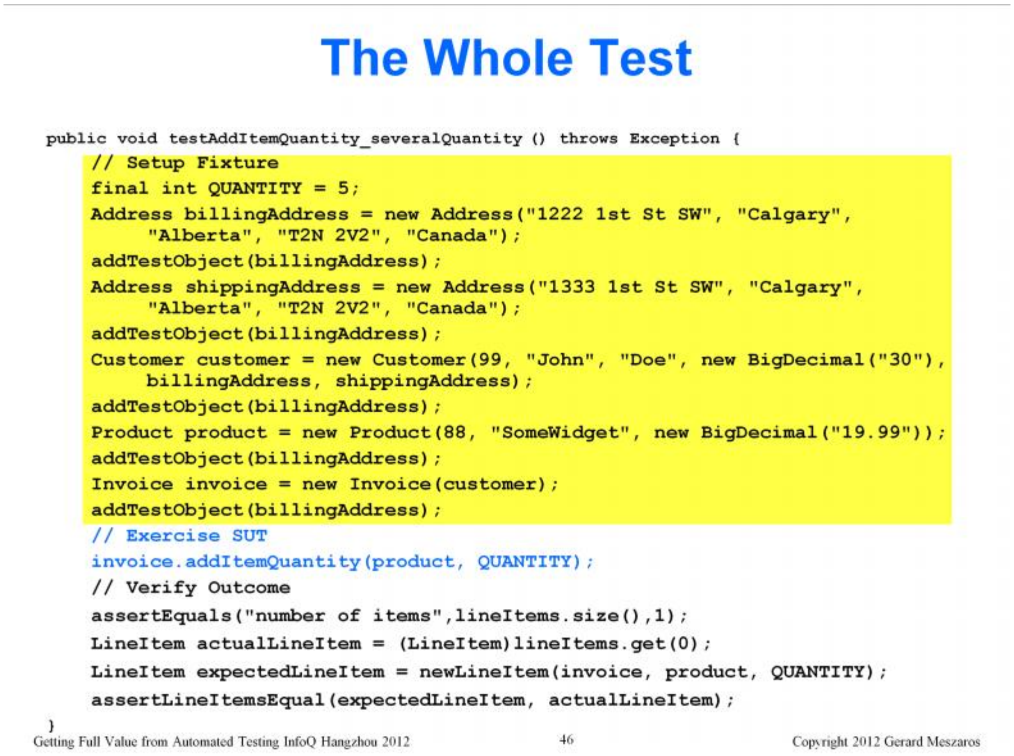
## The Whole Test

```
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!

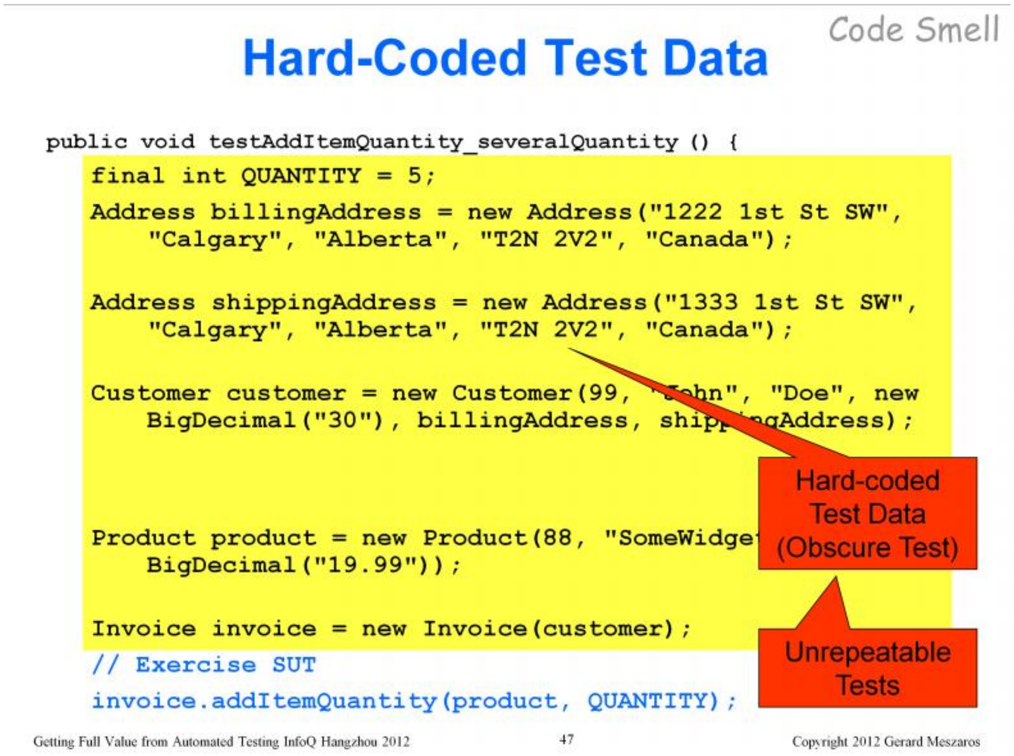
Getting Full Value from Automated Testing InfoQ Hangzhou 2012    45    Copyright 2012 Gerard Meszaros

So now that we’ve eliminated the need for custom teardown code, what else can we do to clean up this test?



# 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);
}
```



Code Smell

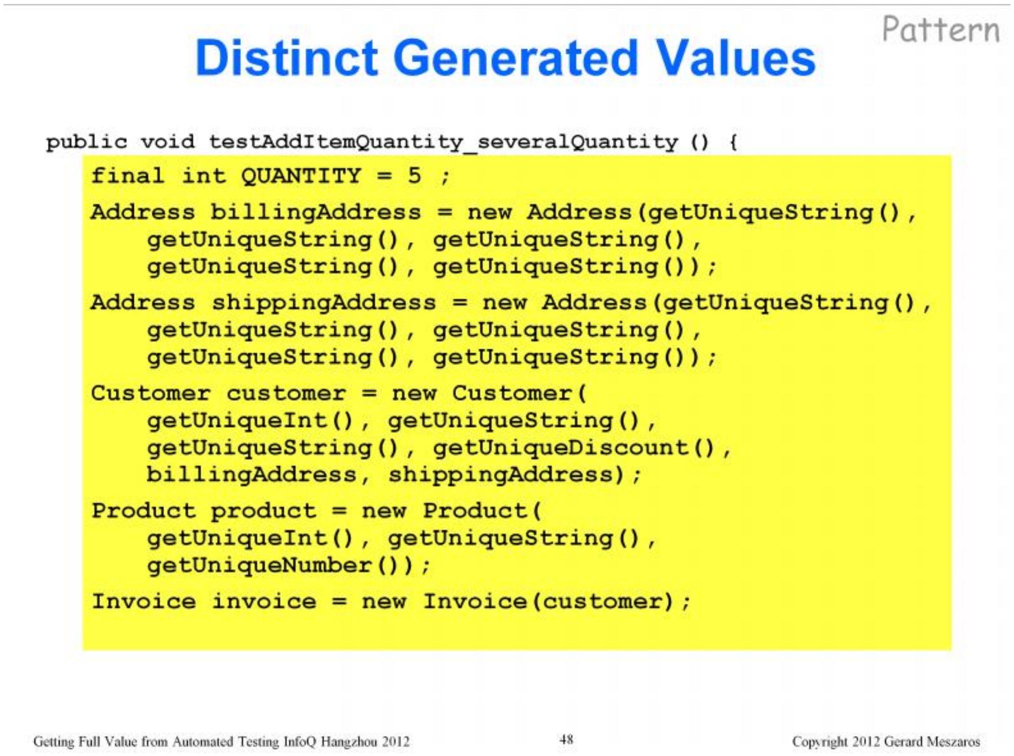
## Hard-Coded Test Data

```
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, "SomeWidge
        BigDecimal("19.99"));

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

Hard-coded Test Data (Obscure Test)

Unrepeatable Tests

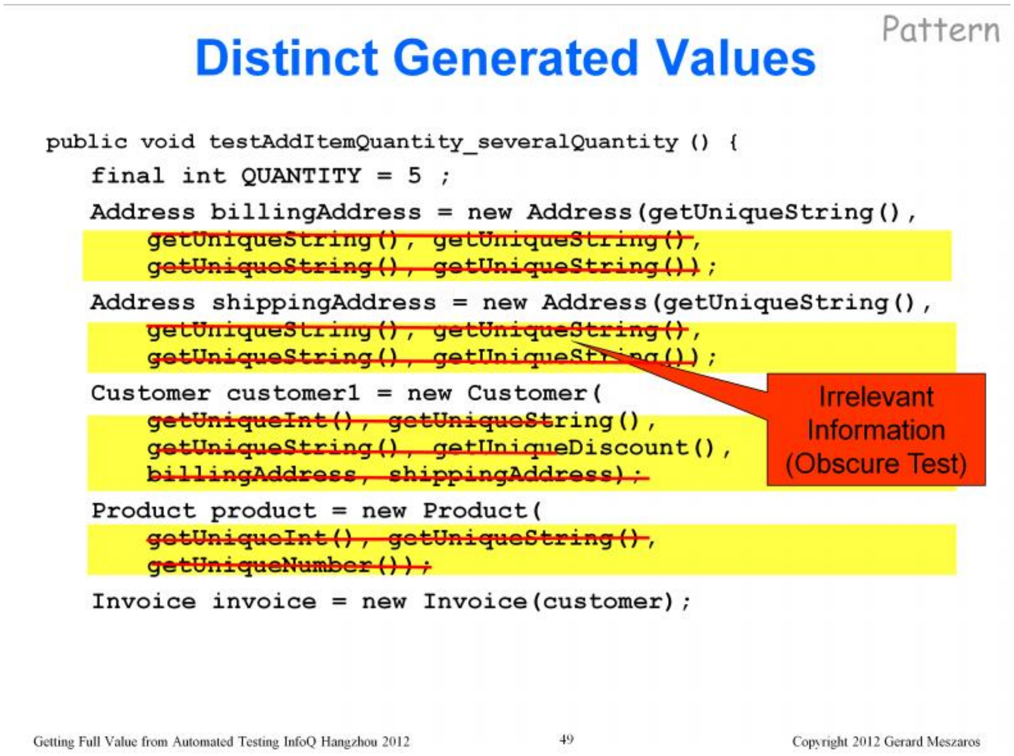
Getting Full Value from Automated Testing InfoQ Hangzhou 2012 47 Copyright 2012 Gerard Meszaros



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);
```



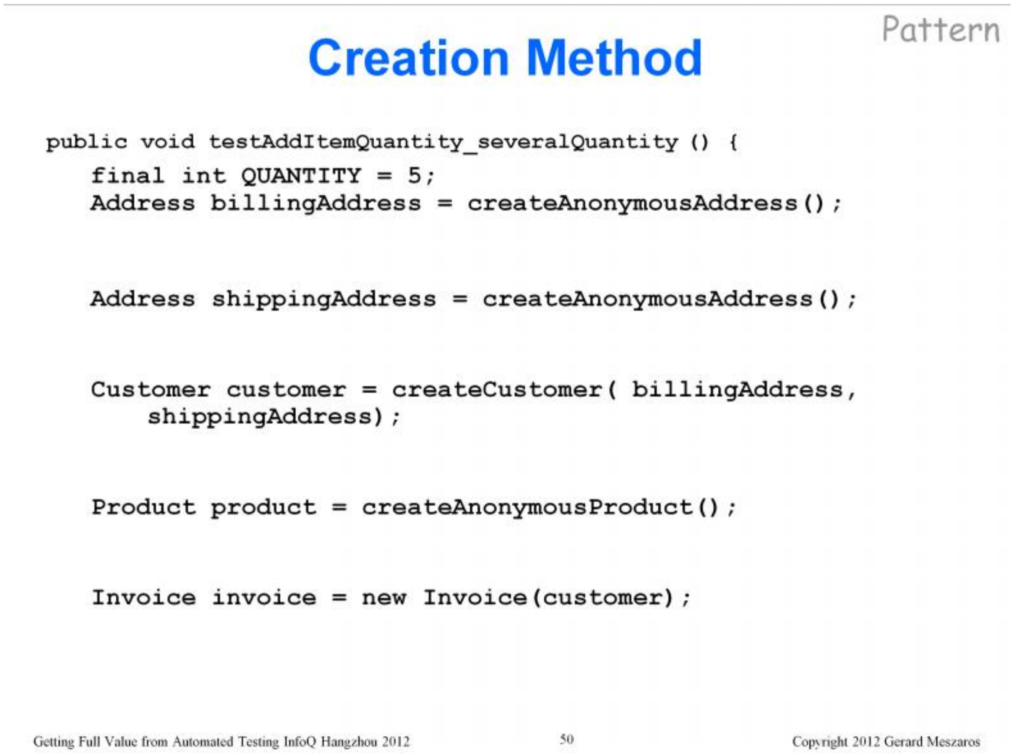
Pattern

# 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(), getUniqueSt~~ring(),
       ~~getUniqueString(), getUniqu~~eDiscount(),
       ~~billingAddress, shippingAddress);~~
   Product product = new Product(
       ~~getUniqueInt(), getUniqueString()~~,
       ~~getUniqueNumber());~~
   Invoice invoice = new Invoice(customer);
```

Irrelevant Information (Obscure Test)

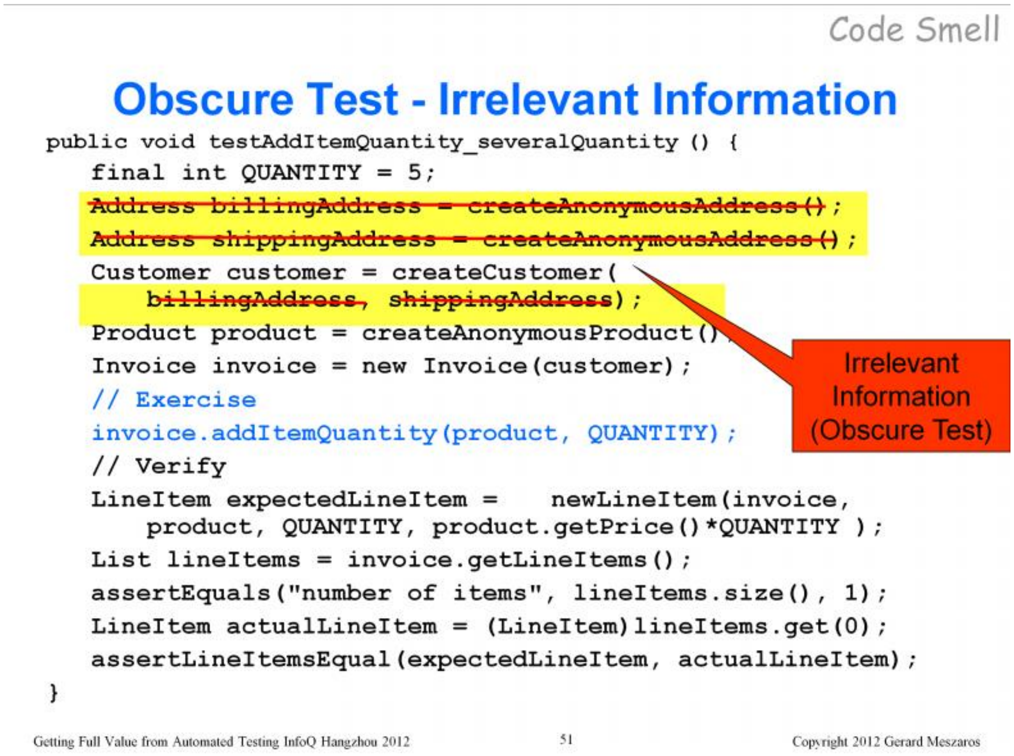
Getting Full Value from Automated Testing InfoQ Hangzhou 2012 49



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);
```

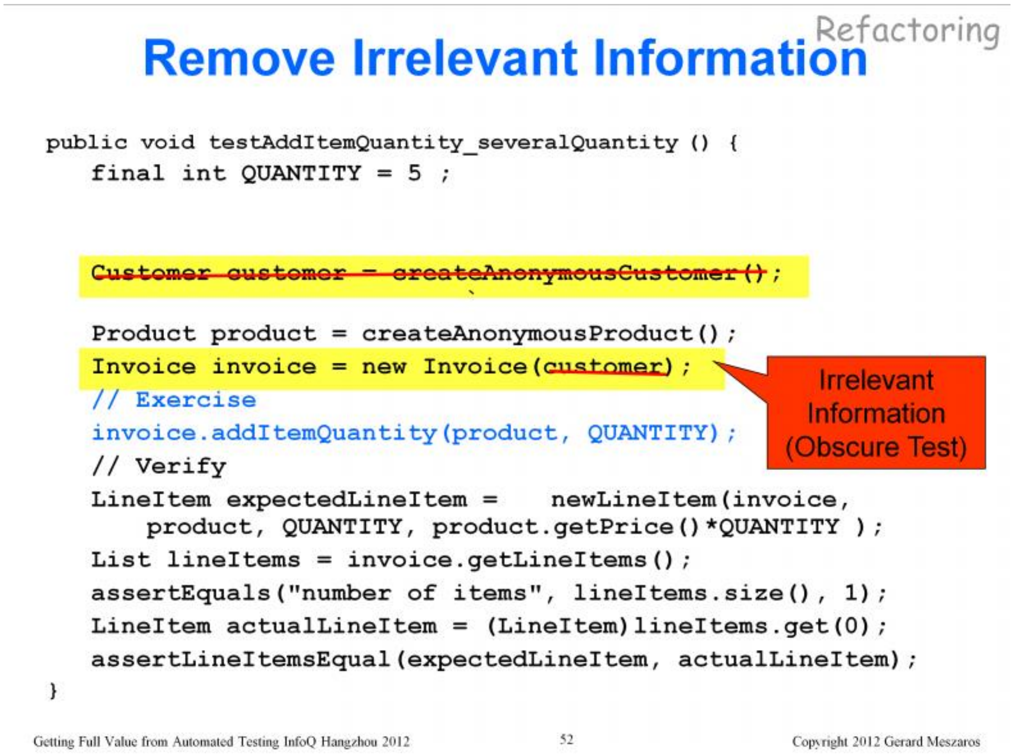


Code Smell

# 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)

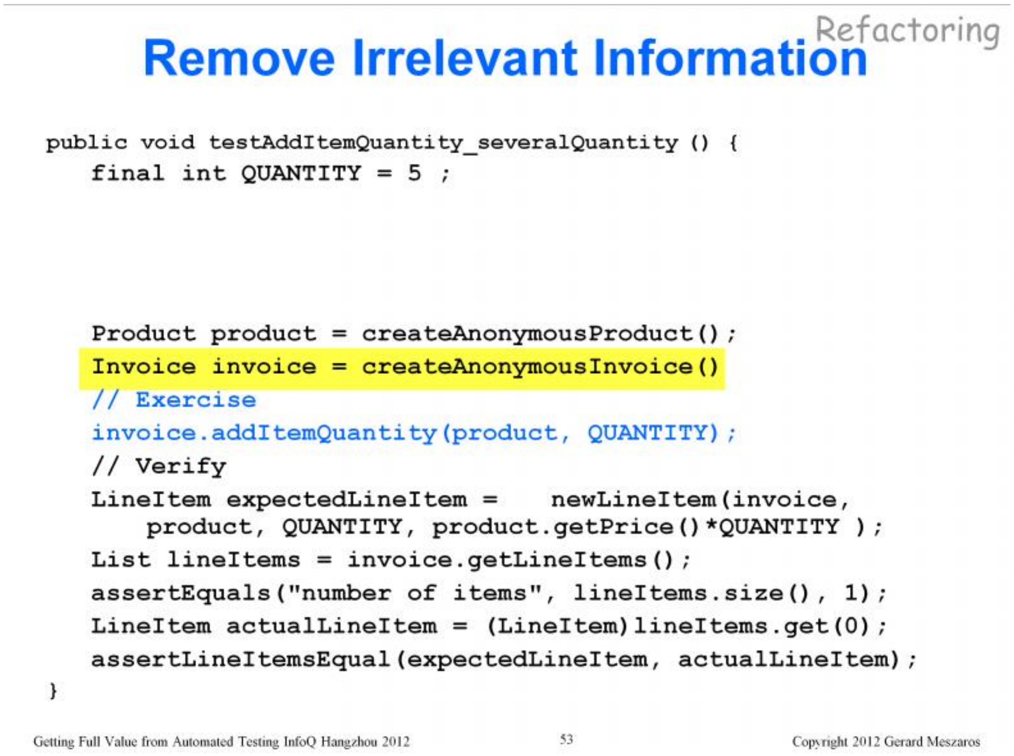


Refactoring

# 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);
}
```

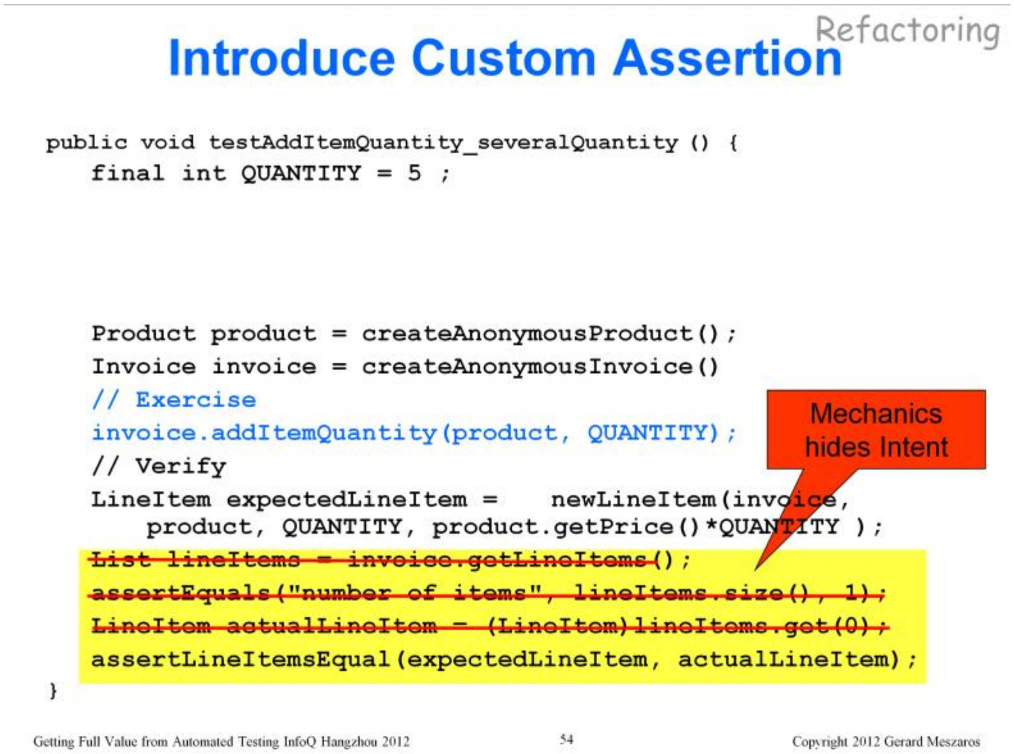
Irrelevant Information (Obscure Test)



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);
}
```



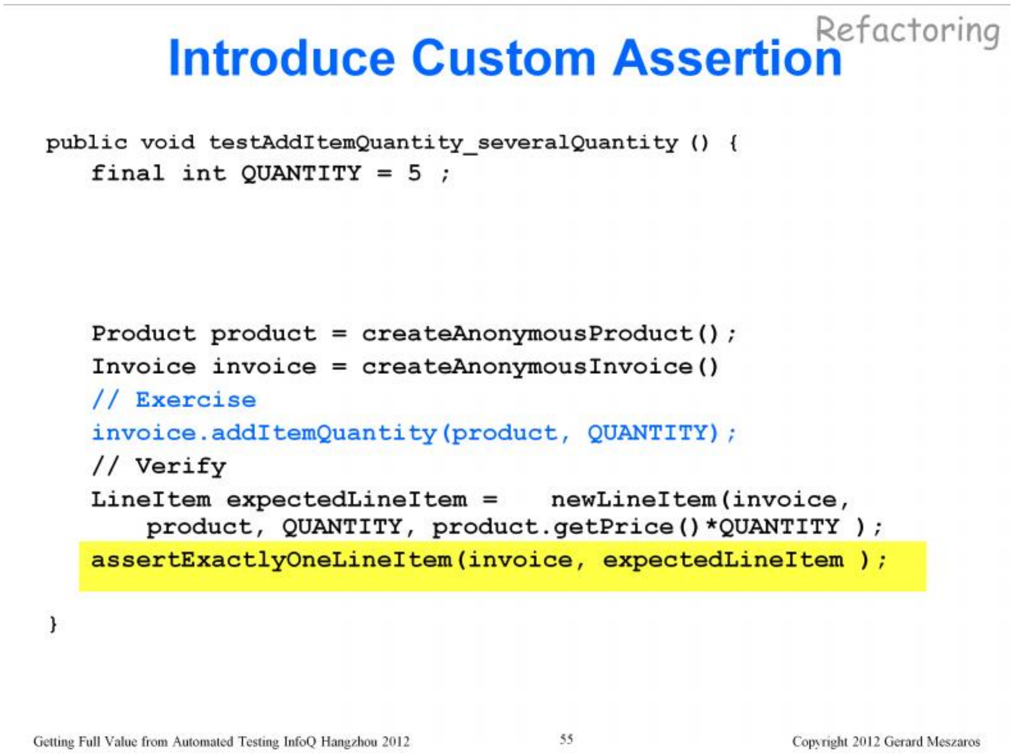
Refactoring

# 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

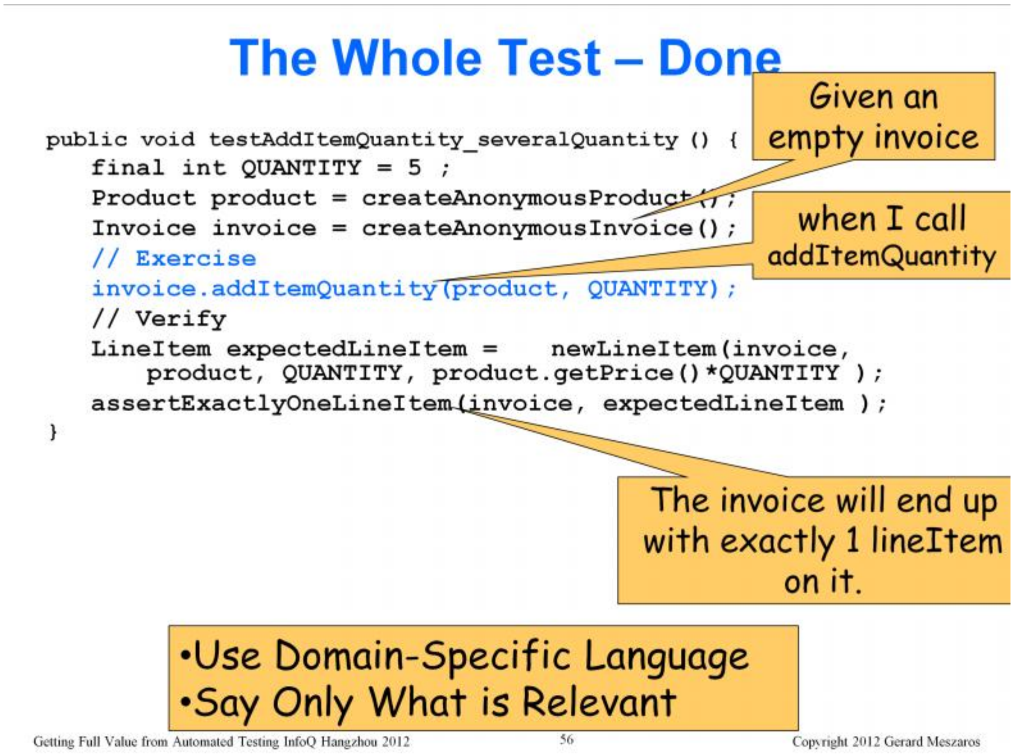
Getting Full Value from Automated Testing InfoQ Hangzhou 2012    54    Copyright 2012 Gerard Meszaros



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 );
}
```

Given an empty invoice

when I call addItemQuantity

The invoice will end up with exactly 1 lineItem on it.

- •Use Domain-Specific Language
- •Say Only What is Relevant

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 56 Copyright 2012 Gerard Meszaros

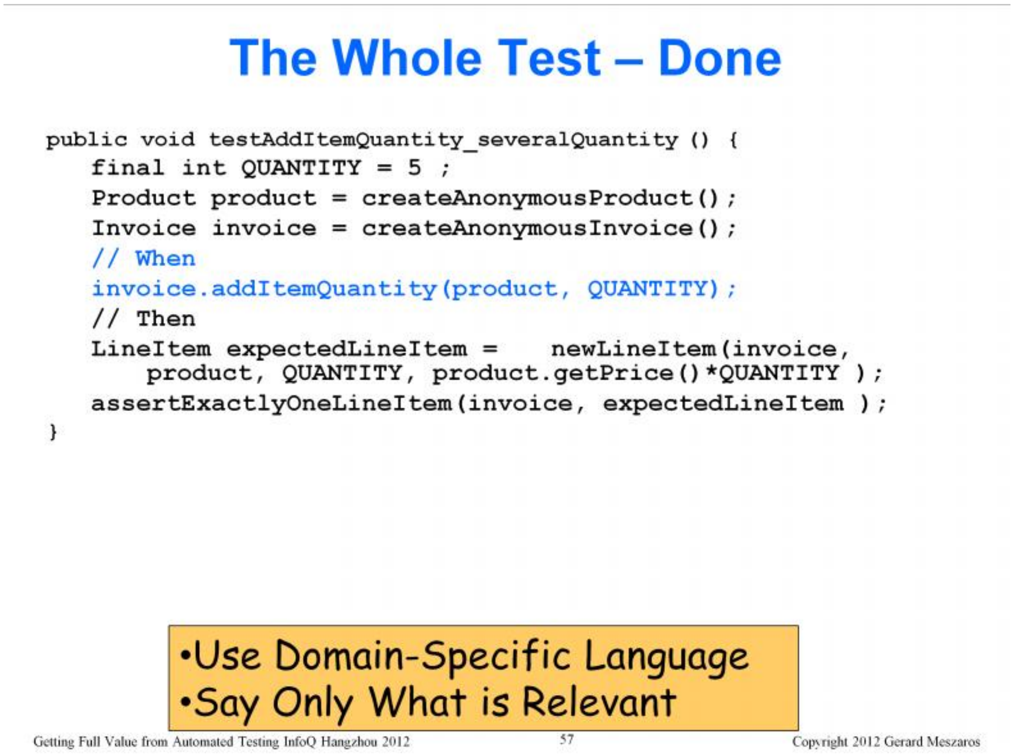
So now we have our test down to just 6 lines of code. We can read out the test condition very easily.

Given an empty invoice

When I call addItemQuantity

Then, the invoice will end up with exactly one line item on it with the item value equal to the product price multipled by the quantity.

This is pretty clear but weshould always be looking for ways to improve the readability of our tests.



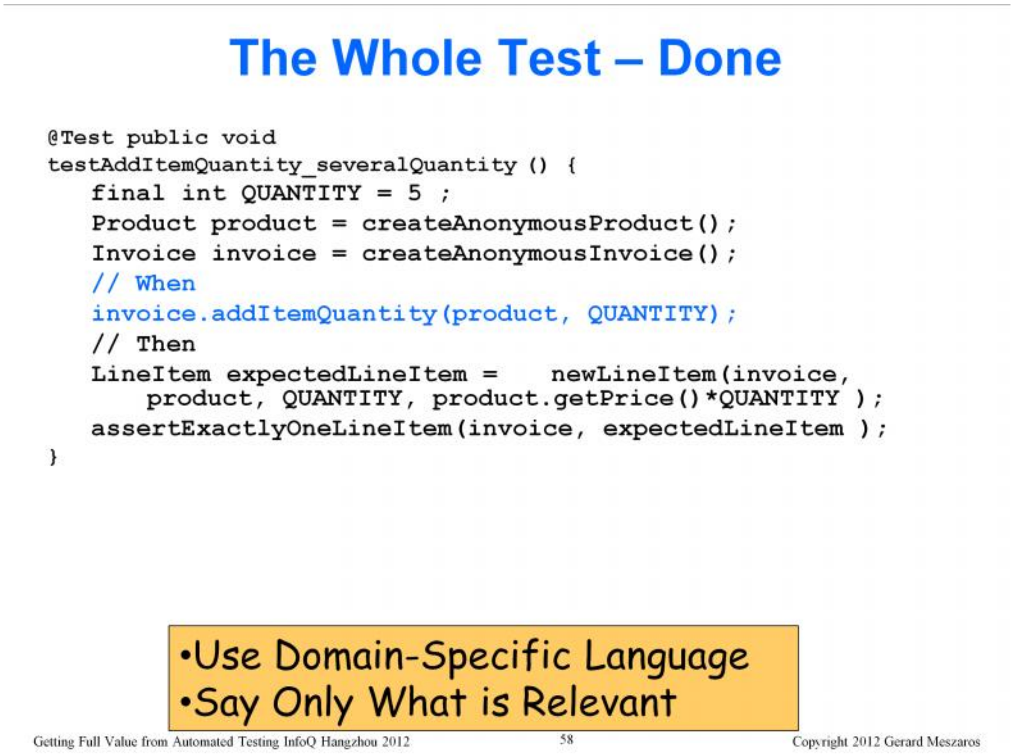
## The Whole Test – Done

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

- Use Domain-Specific Language
- Say Only What is Relevant

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 57 Copyright 2012 Gerard Meszaros

We can replace the procedural comments Exercise and Verify with When and Then.



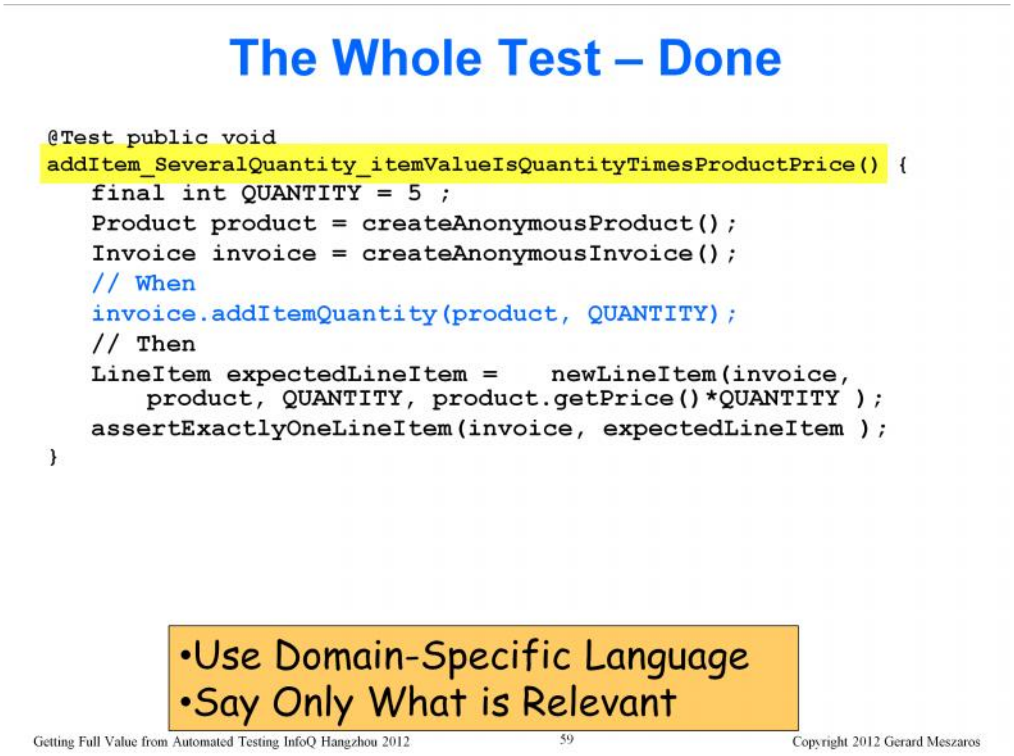
## The Whole Test – Done

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

- Use Domain-Specific Language
- Say Only What is Relevant

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 58 Copyright 2012 Gerard Meszaros

Here I just split the method name from the returns type to give myself a little room. Here's what I want to do:

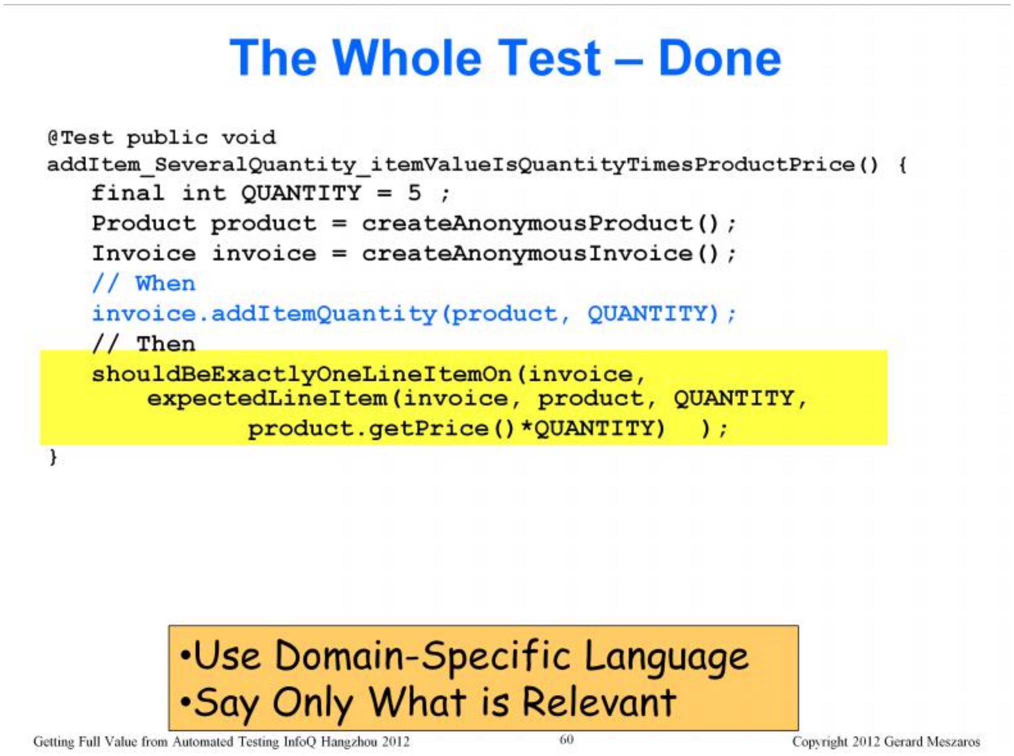


## The Whole Test – Done

```
@Test public void
addItem_SeveralQuantity_itemValueIsQuantityTimesProductPrice() {
   final int QUANTITY = 5 ;
   Product product = createAnonymousProduct();
   Invoice invoice = createAnonymousInvoice();
   // When
   invoice.addItemQuantity(product, QUANTITY);
   // Then
   LineItem expectedLineItem =    newLineItem(invoice,
       product, QUANTITY, product.getPrice()*QUANTITY );
   assertExactlyOneLineItem(invoice, expectedLineItem );
}
```

- Use Domain-Specific Language
- Say Only What is Relevant

Rename the method to reflect the when and the expected outcome. When I call AddItemQuanity with several quantity, the item’s value is expected to be the quantity time the product’s price.



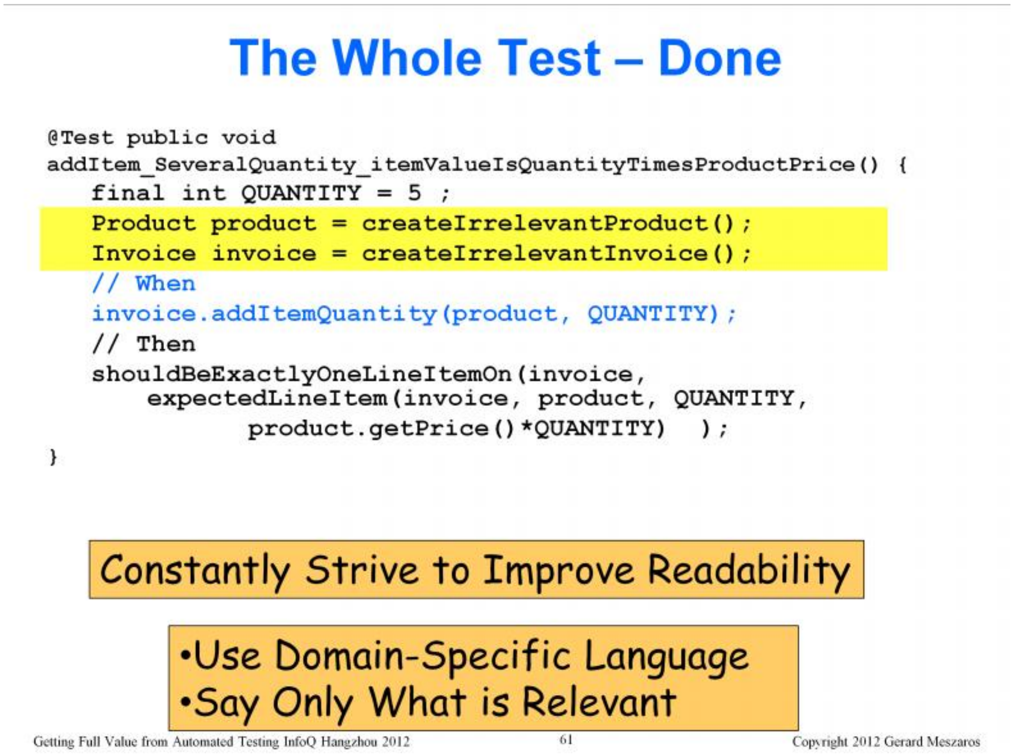
## The Whole Test – Done

```
@Test public void
addItem_SeveralQuantity_itemValueIsQuantityTimesProductPrice() {
   final int QUANTITY = 5 ;
   Product product = createAnonymousProduct();
   Invoice invoice = createAnonymousInvoice();
   // When
   invoice.addItemQuantity(product, QUANTITY);
   // Then
   shouldBeExactlyOneLineItemOn(invoice,
       expectedLineItem(invoice, product, QUANTITY,
              product.getPrice()*QUANTITY)   );
}
```

- •Use Domain-Specific Language
- •Say Only What is Relevant

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 60 Copyright 2012 Gerard Meszaros

We can also make that clearer Then part of the test by renaming the assertion to shouldBeExactlyOneLineItemOn the invoice and renaming the utility method to make it clear we are constructing an expected object.



## The Whole Test – Done

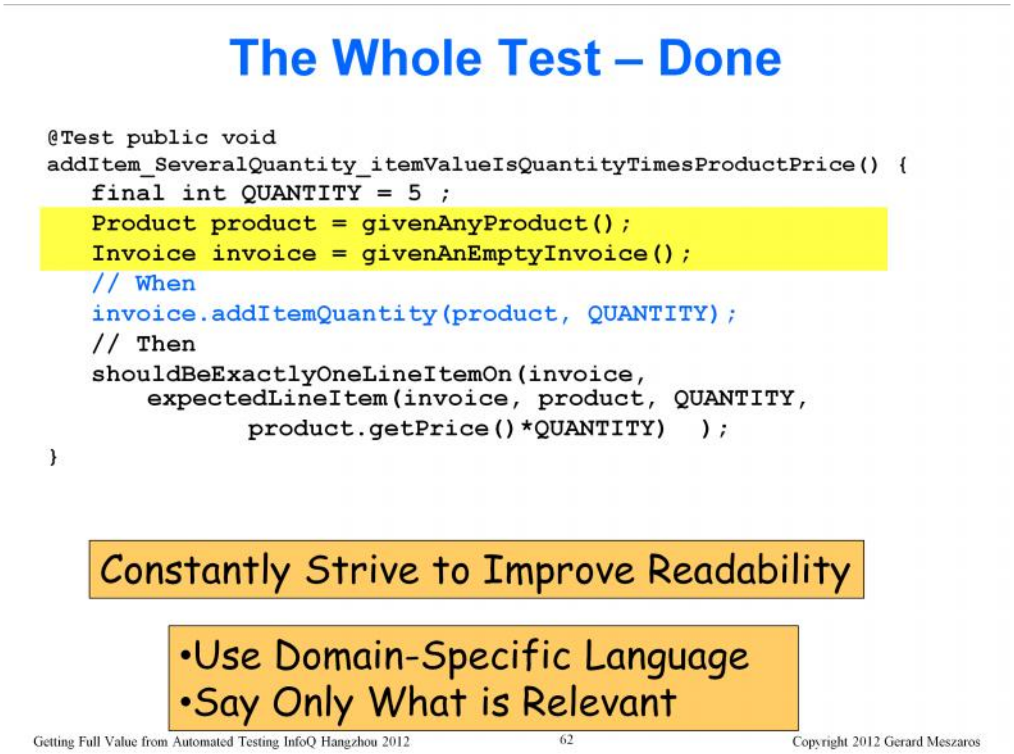
```
@Test public void
addItem_SeveralQuantity_itemValueIsQuantityTimesProductPrice() {
    final int QUANTITY = 5 ;
    Product product = createIrrelevantProduct();
    Invoice invoice = createIrrelevantInvoice();
    // When
    invoice.addItemQuantity(product, QUANTITY);
    // Then
    shouldBeExactlyOneLineItemOn(invoice,
        expectedLineItem(invoice, product, QUANTITY,
                product.getPrice()*QUANTITY)  );
}
```

Constantly Strive to Improve Readability

- Use Domain-Specific Language
- Say Only What is Relevant

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 61 Copyright 2012 Gerard Meszaros

We can replace the procedural comments Exercise and Verify with When and Then.
It would be more accurate to call them “createIrrelevantSomething”.



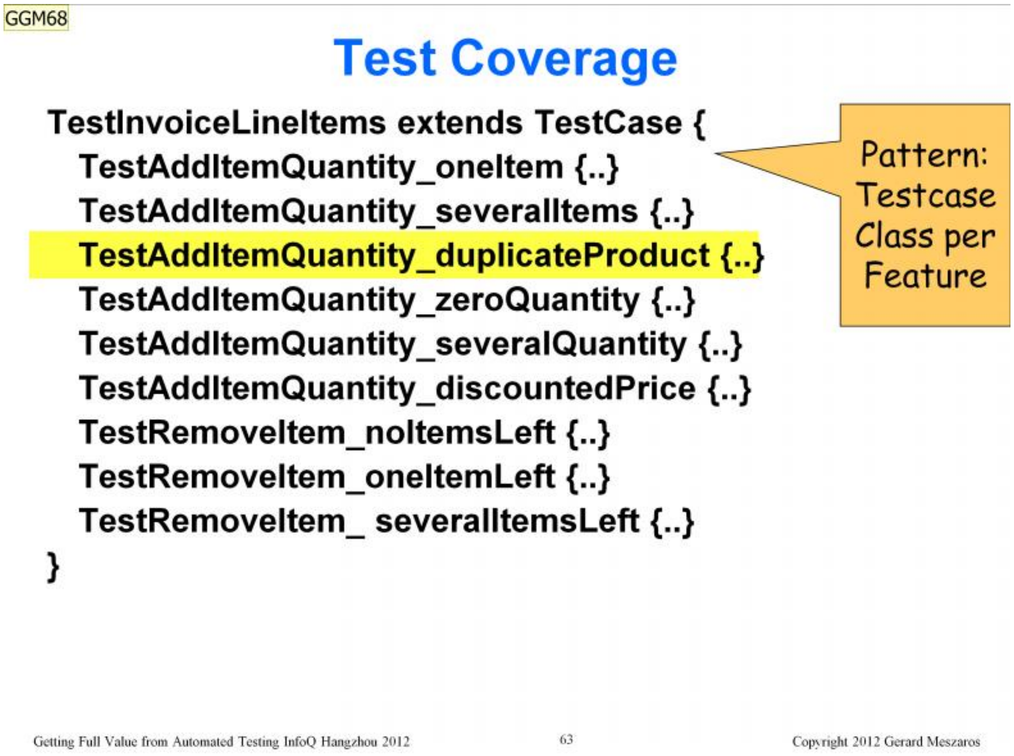
## The Whole Test – Done

```
@Test public void
addItem_SeveralQuantity_itemValueIsQuantityTimesProductPrice() {
    final int QUANTITY = 5 ;
    Product product = givenAnyProduct();
    Invoice invoice = givenAnEmptyInvoice();
    // When
    invoice.addItemQuantity(product, QUANTITY);
    // Then
    shouldBeExactlyOneLineItemOn(invoice,
        expectedLineItem(invoice, product, QUANTITY,
                product.getPrice()*QUANTITY)  );
}
```

Constantly Strive to Improve Readability

- Use Domain-Specific Language
- Say Only What is Relevant

But it is even clearer if we simply call them givenSomething().



GGM68

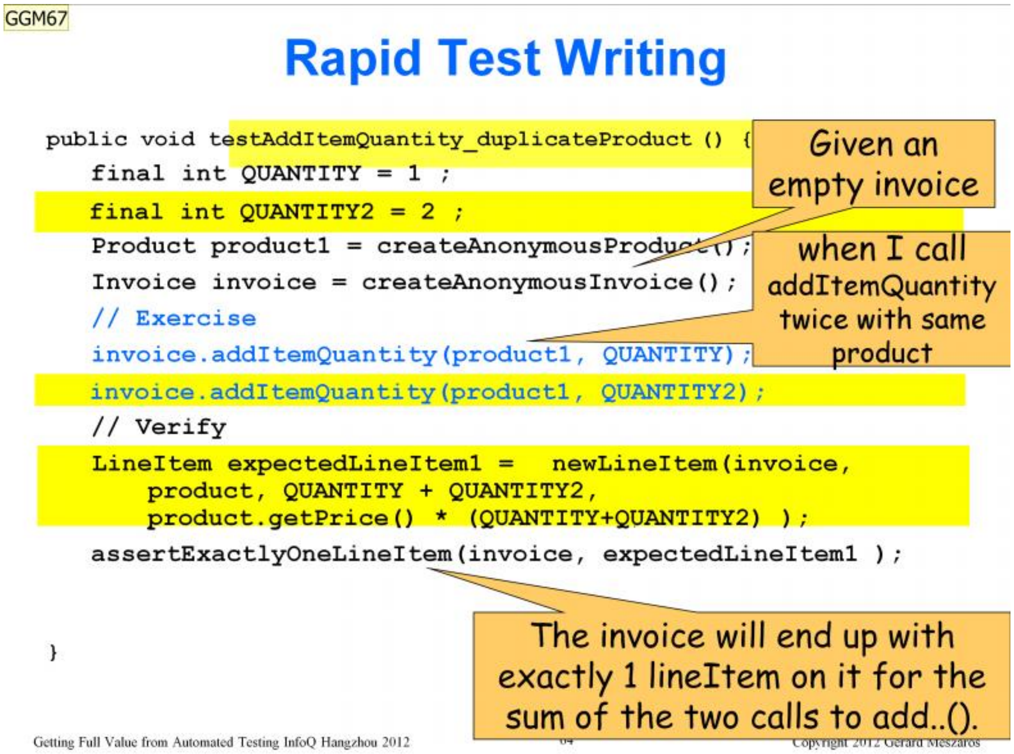
## Test Coverage

```
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

Slide 63

GGM68 Change to new naming conventions
Gerard Meszaros, 10/19/2012



GGM67

## Rapid Test Writing

```
public void testAddItemQuantity_duplicateProduct () {
    final int QUANTITY = 1 ;
    final int QUANTITY2 = 2 ;
    Product product1 = createAnonymousProduct();
    Invoice invoice = createAnonymousInvoice();
    // Exercise
    invoice.addItemQuantity(product1, QUANTITY);
    invoice.addItemQuantity(product1, QUANTITY2);
    // Verify
    LineItem expectedLineItem1 =  newLineItem(invoice,
        product, QUANTITY + QUANTITY2,
        product.getPrice() * (QUANTITY+QUANTITY2) );
    assertExactlyOneLineItem(invoice, expectedLineItem1 );


}
```

Given an empty invoice

when I call addItemQuantity twice with same product

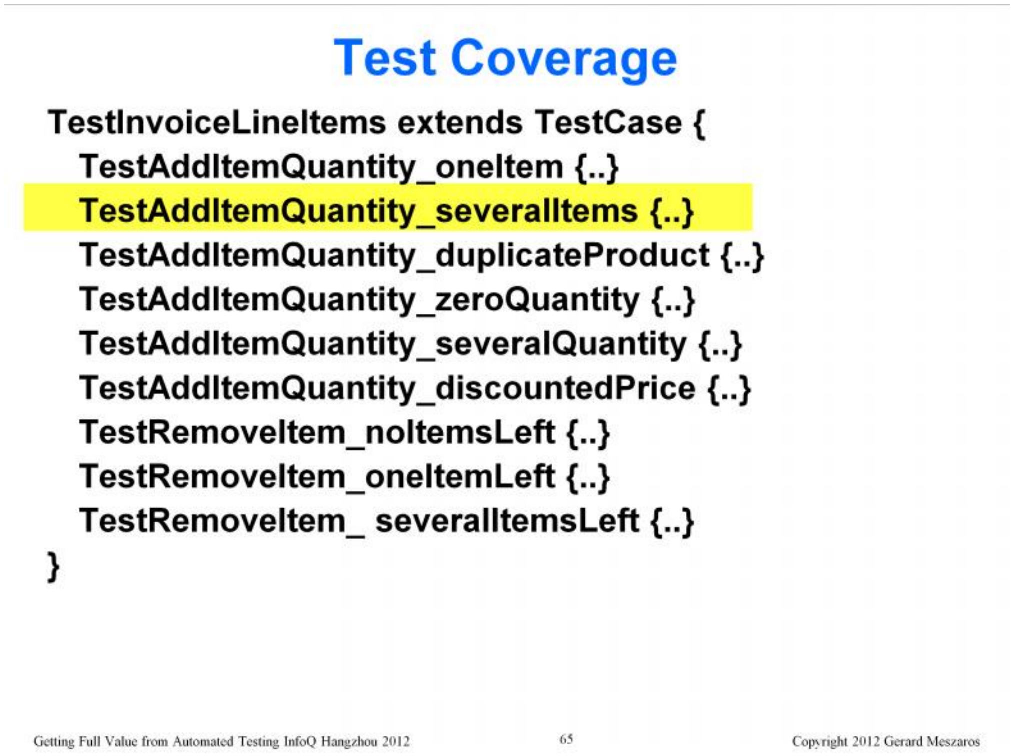
The invoice will end up with exactly 1 lineItem on it for the sum of the two calls to add..().

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 64 Copyright 2012 Gerard Meszaros

## Slide 64

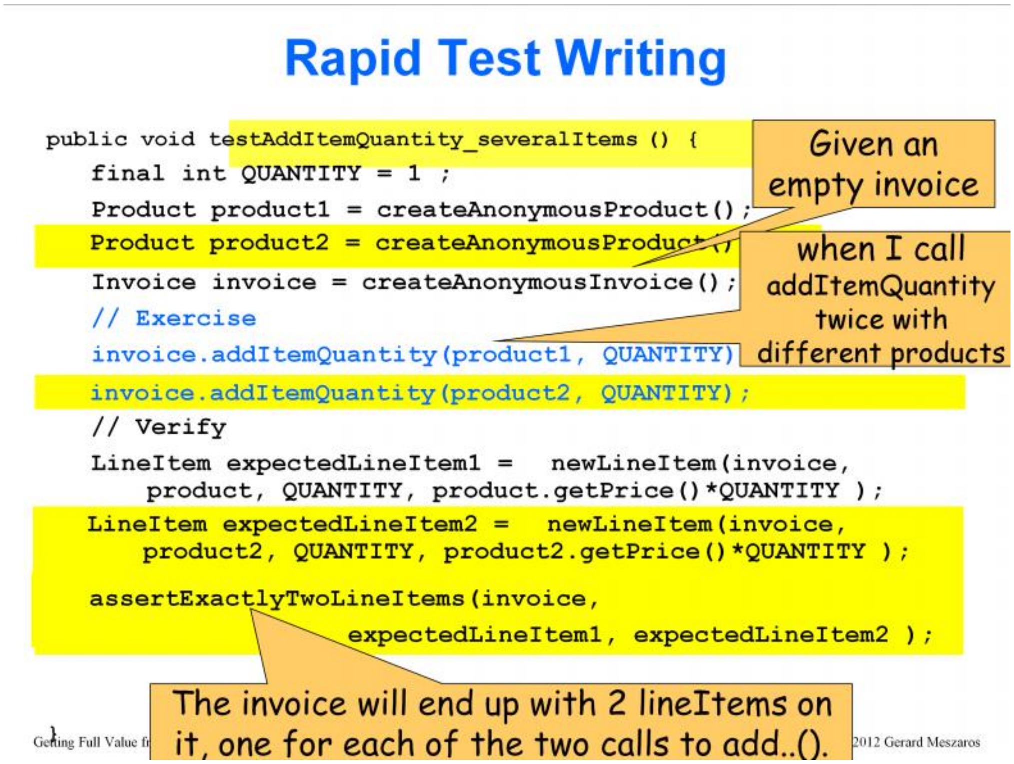
GGM67 Redo using new naming conventions

Gerard Meszaros, 10/19/2012



## Test Coverage

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



## Rapid Test Writing

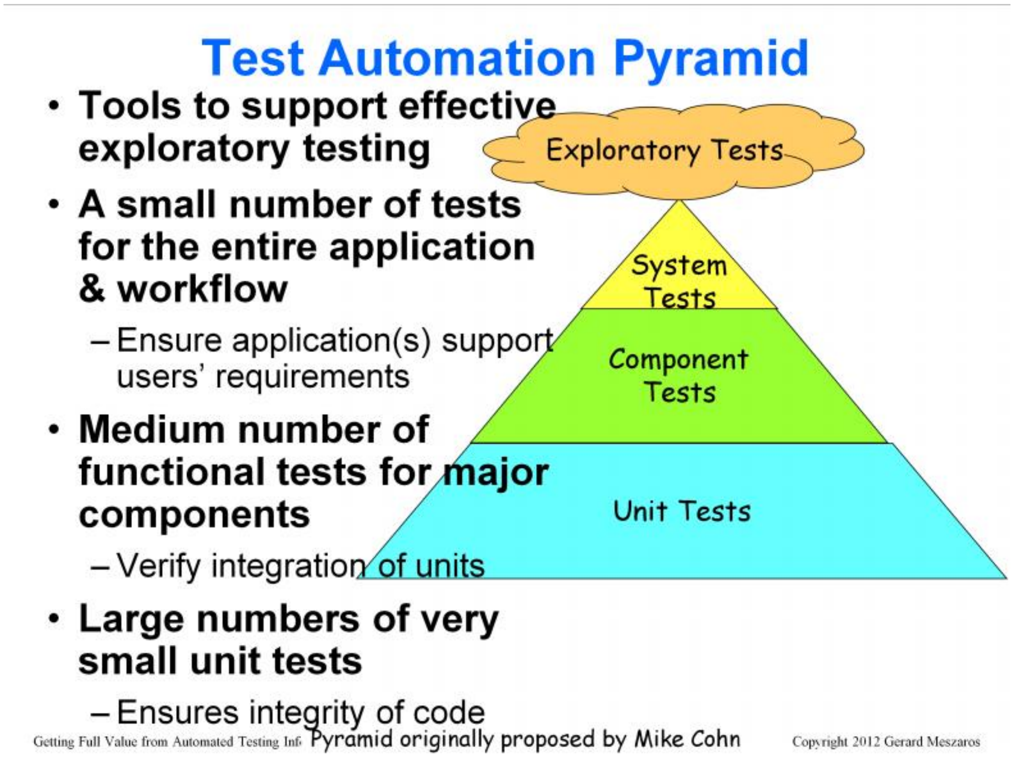
```
public void testAddItemQuantity_severalItems () {
   final int QUANTITY = 1 ;
   Product product1 = createAnonymousProduct();
   Product product2 = createAnonymousProduct();
   Invoice invoice = createAnonymousInvoice();
   // Exercise
   invoice.addItemQuantity(product1, QUANTITY)
   invoice.addItemQuantity(product2,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

Given an empty invoice

when I call addItemQuantity twice with different products

The invoice will end up with 2 lineItems on it, one for each of the two calls to add..().

Getting Full Value fr 2012 Gerard Meszaros

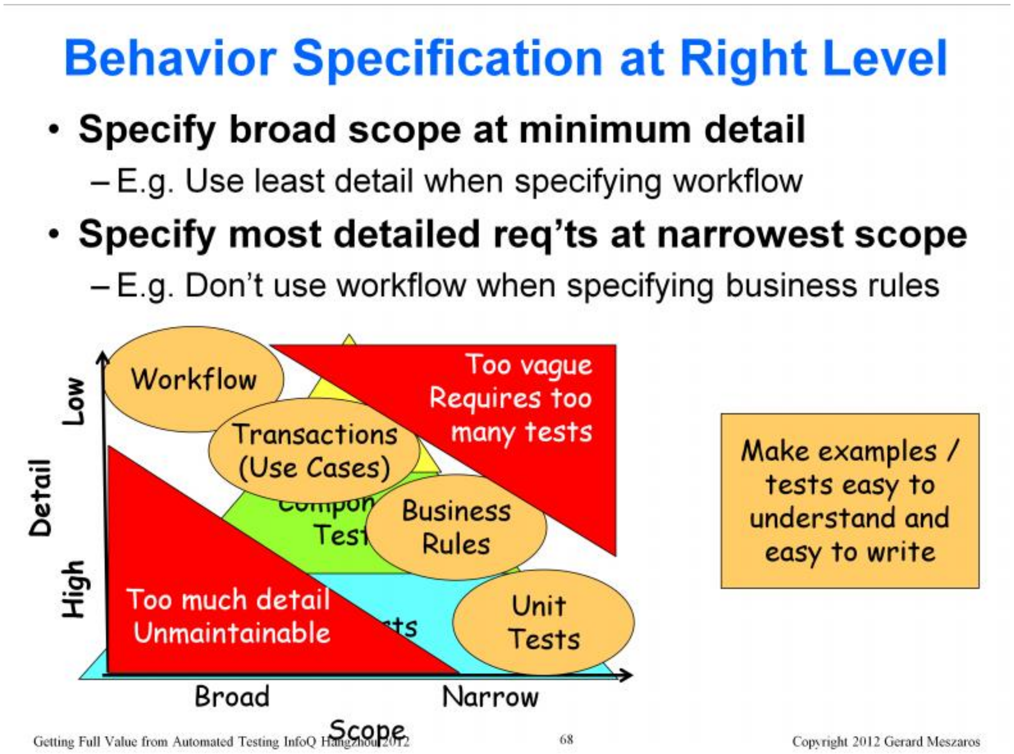


## Test Automation Pyramid

- **Tools to support effective exploratory testing**
- **A small number of tests for the entire application & workflow**
  - Ensure application(s) support users' requirements
- **Medium number of functional tests for major components**
  - Verify integration of units
- **Large numbers of very small unit tests**
  - Ensures integrity of code

Getting Full Value from Automated Testing Inf

Pyramid originally proposed by Mike Cohn



## Behavior Specification at Right Level

- **Specify broad scope at minimum detail**
  - E.g. Use least detail when specifying workflow
- **Specify most detailed req'ts at narrowest scope**
  - E.g. Don't use workflow when specifying business rules

Make examples / tests easy to understand and easy to write

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 68 Copyright 2012 Gerard Meszaros

It is important to specify each story at the right level.

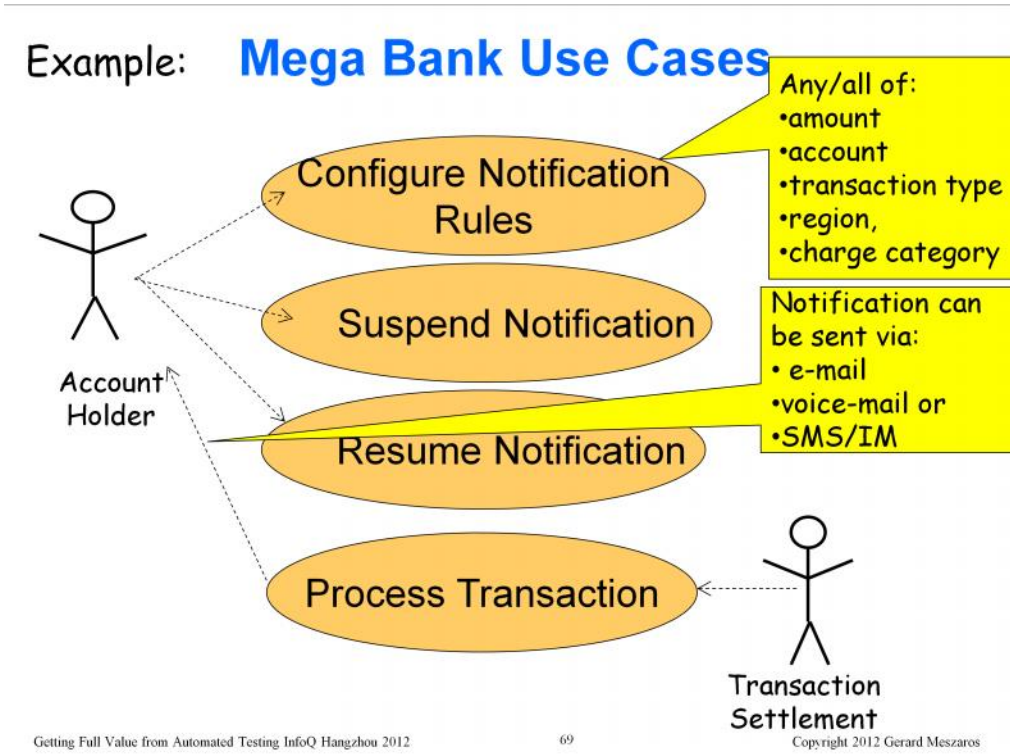
Tests with very broad scope (such as the end-to-end process) should be specified with a minimum of detail.

Tests that specify a great deal of detail should be kept very narrow in scope.

Specifying broad scope in high detail results in a lot of duplicated and unmaintainable detail.

Specifying with too little detail for narrow scope results in too many specs that say very little.

The goal is to make our examples and tests each to understand and easy to write. This requires using the right language in each spec.



Example: **Mega Bank Use Cases**

Configure Notification Rules

Suspend Notification

Resume Notification

Process Transaction

Account Holder

Transaction Settlement

Any/all of:
•amount
•account
•transaction type
•region,
•charge category

Notification can be sent via:
• e-mail
•voice-mail or
•SMS/IM

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 69 Copyright 2012 Gerard Meszaros

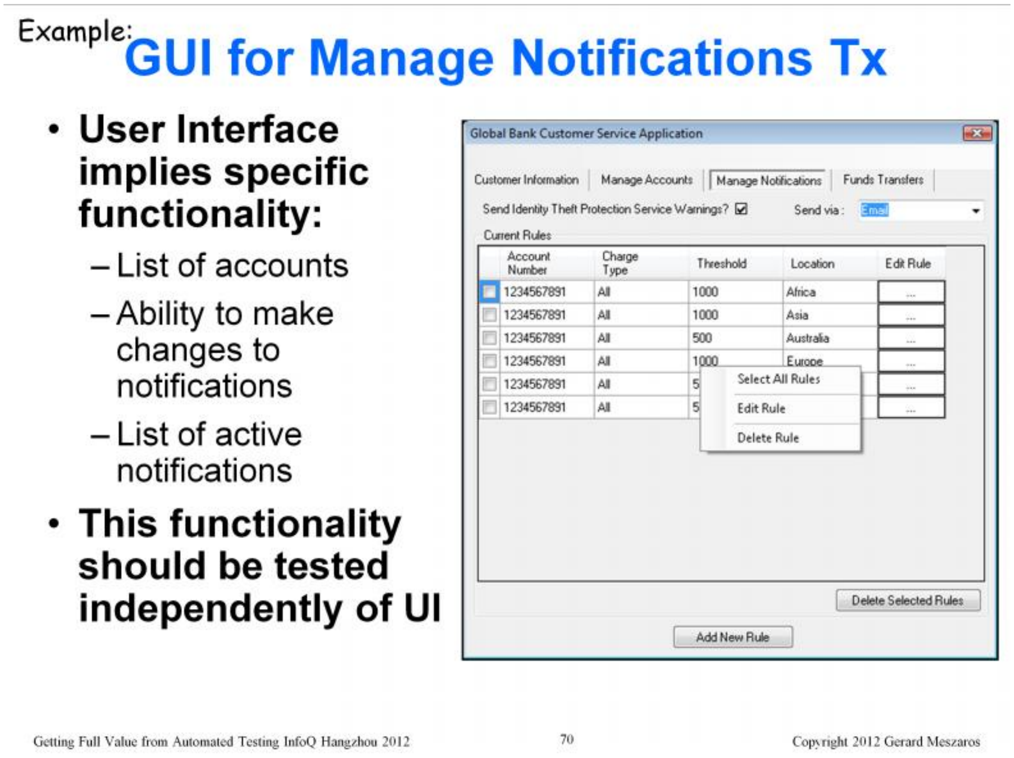
Let’s look at how we can structure the tests for a banking application that notifies the user of transactions against their accounts.

User can configure threshold amount for notification based on any/all of account, transaction type or region, charge category

Notification can be sent via e-mail, voice-mail or SMS/IM

User can suspend notifications indefinitely or for a defined period of time.

When the bank processes a charge or credit transaction, it uses these rules to decide whether and how to send the notification.



Example:

## GUI for Manage Notifications Tx

- **User Interface implies specific functionality:**
  - List of accounts
  - Ability to make changes to notifications
  - List of active notifications
- **This functionality should be tested independently of UI**

Global Bank Customer Service Application

Send Identity Theft Protection Service Warnings? ☑ Send via : Email

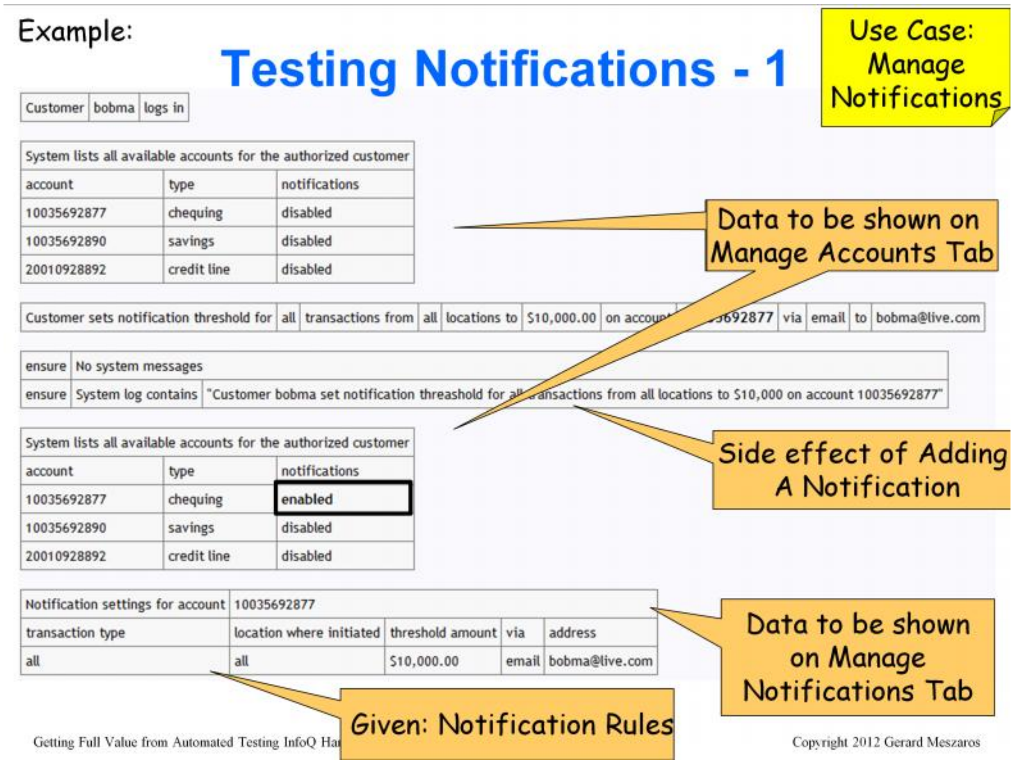
Current Rules

| | Account Number | Charge Type | Threshold | Location | Edit Rule |
|---|---|---|---|---|---|
| □ | 1234567891 | All | 1000 | Africa | ... |
| □ | 1234567891 | All | 1000 | Asia | ... |
| □ | 1234567891 | All | 500 | Australia | ... |
| □ | 1234567891 | All | 1000 | Europe | ... |
| □ | 1234567891 | All | 5 | | ... |
| □ | 1234567891 | All | 5 | | ... |

Now that we understand how the various use cases (or transactions) relate to each other (the overall workflow) we can design the user interface for Managing the Notifications. We often want to design the UI to handle a whole range of user stories to ensure consistency of the user experience even though we will typically implement it story by story.

Once we’ve designed the UI required by the User Stories, we’ll want to do some usability testing to find out whether users find it useful or cumbersome. Much cheaper to find out now that after we’ve built and deployed the stories.

We can also implement automated tests for the UI although this is a contentious topic; it’s rather cumbersome to automate but relatively easy to test manually.



Example:

## Testing Notifications - 1

Use Case: Manage Notifications

| | | |
|---|---|---|
| Customer | bobma | logs in |

| System lists all available accounts for the authorized customer | | |
|---|---|---|
| account | type | notifications |
| 10035692877 | chequing | disabled |
| 10035692890 | savings | disabled |
| 20010928892 | credit line | disabled |

Data to be shown on Manage Accounts Tab

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Customer sets notification threshold for | all | transactions from | all | locations to | $10,000.00 | on accou | 5692877 | via | email | to | bobma@live.com |

| | | |
|---|---|---|
| ensure | No system messages | |
| ensure | System log contains | "Customer bobma set notification threashold for al ansactions from all locations to $10,000 on account 10035692877" |

| System lists all available accounts for the authorized customer | | |
|---|---|---|
| account | type | notifications |
| 10035692877 | chequing | **enabled** |
| 10035692890 | savings | disabled |
| 20010928892 | credit line | disabled |

Side effect of Adding A Notification

| Notification settings for account | 10035692877 | | | |
|---|---|---|---|---|
| transaction type | location where initiated | threshold amount | via | address |
| all | all | $10,000.00 | email | bobma@live.com |

Data to be shown on Manage Notifications Tab

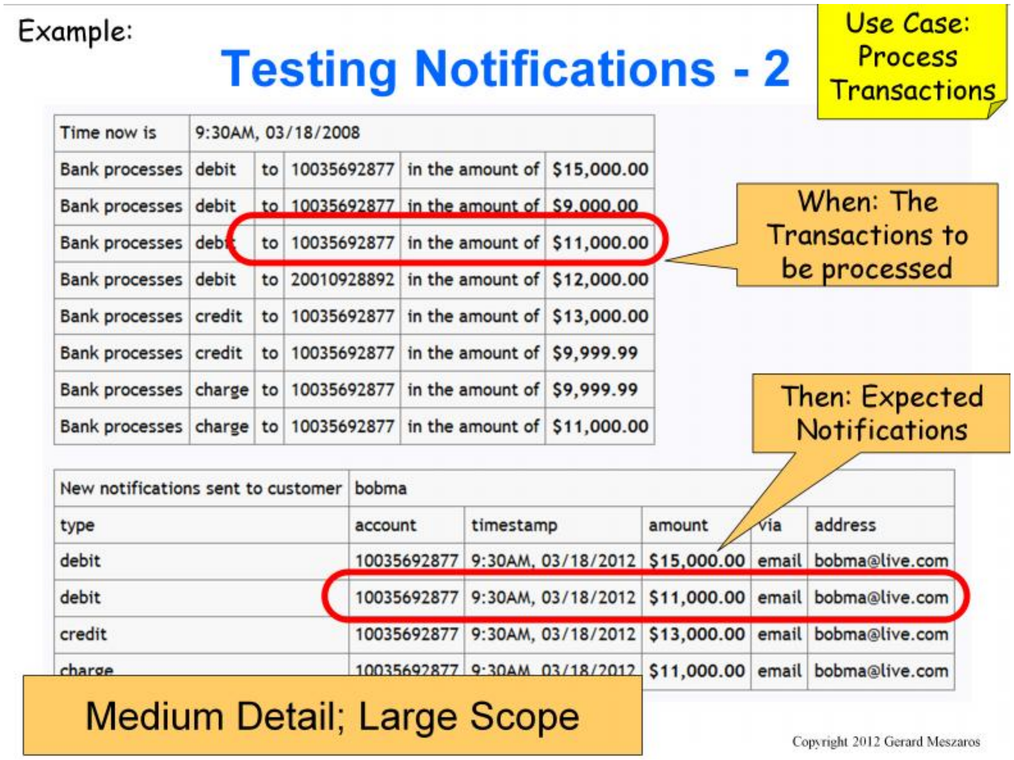
Given: Notification Rules

Getting Full Value from Automated Testing InfoQ Ha

Copyright 2012 Gerard Meszaros

Here’s a first crack at writing a test that specifes how our application should work. First, a user signs in and configures a notification rule on one accounts. All transactions of any type over $10,000 on their chequing account should result in an e-mail to them. They can verify that the rule was accepted by reviewing the list of accounts and the active notifications.  All of this effort just to set up the Given for the test conditions.

Next we need to create some transactions against these accounts.



Example:

# Testing Notifications - 2

Use Case: Process Transactions

| Time now is | 9:30AM, 03/18/2008 | | | | |
|---|---|---|---|---|---|
| Bank processes | debit | to | 10035692877 | in the amount of | $15,000.00 |
| Bank processes | debit | to | 10035692877 | in the amount of | $9,000.00 |
| Bank processes | debit | to | 10035692877 | in the amount of | $11,000.00 |
| Bank processes | debit | to | 20010928892 | in the amount of | $12,000.00 |
| Bank processes | credit | to | 10035692877 | in the amount of | $13,000.00 |
| Bank processes | credit | to | 10035692877 | in the amount of | $9,999.99 |
| Bank processes | charge | to | 10035692877 | in the amount of | $9,999.99 |
| Bank processes | charge | to | 10035692877 | in the amount of | $11,000.00 |

When: The Transactions to be processed

| New notifications sent to customer | bobma | | | | |
|---|---|---|---|---|---|
| type | account | timestamp | amount | via | address |
| debit | 10035692877 | 9:30AM, 03/18/2012 | $15,000.00 | email | bobma@live.com |
| debit | 10035692877 | 9:30AM, 03/18/2012 | $11,000.00 | email | bobma@live.com |
| credit | 10035692877 | 9:30AM, 03/18/2012 | $13,000.00 | email | bobma@live.com |
| charge | 10035692877 | 9:30AM, 03/18/2012 | $11,000.00 | email | bobma@live.com |

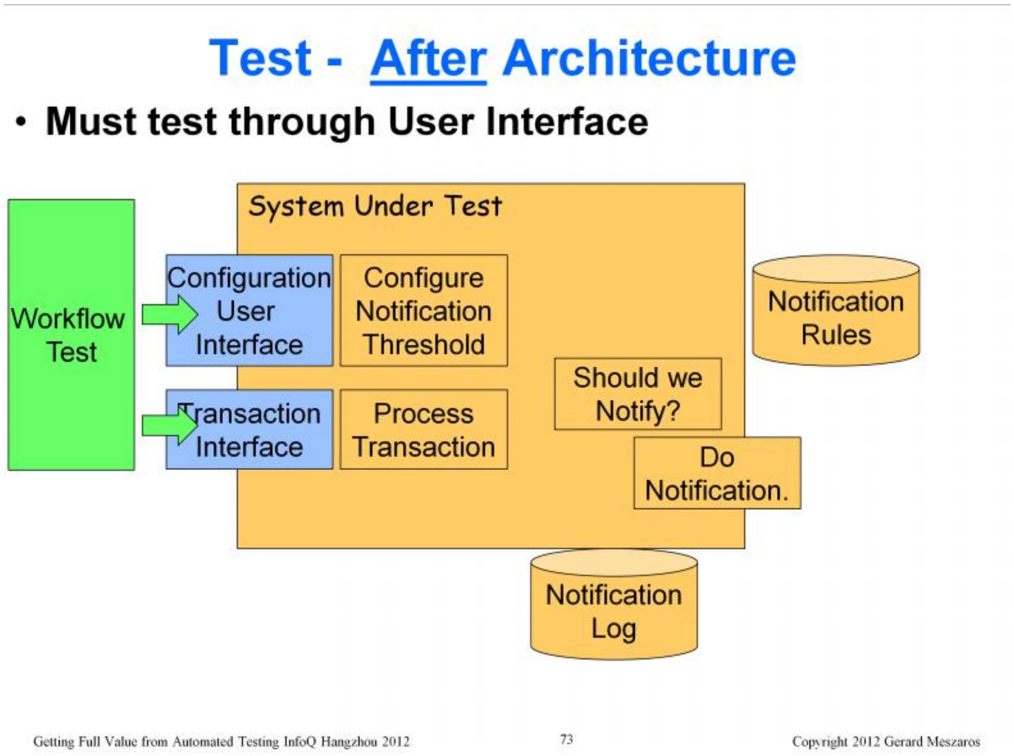
Then: Expected Notifications

Medium Detail; Large Scope

Copyright 2012 Gerard Meszaros

So we process a bunch of transactions in various amounts above and below the threshold for various accounts and transaction types. These constitute the When’s of the test conditions.  The final table lists the expected notifications; the Then’s of the test conditions.

It’s getting a bit difficult to follow along because the cause & effect are not close to each other. And this test only verifies one particular combination of notification rule. We’ll need to create other test cases for each of various combinations of rules we could have. That will result in a lot of repetition across the test cases. And these test cases will take a long time to run if we hook them up to the user interface of our application.



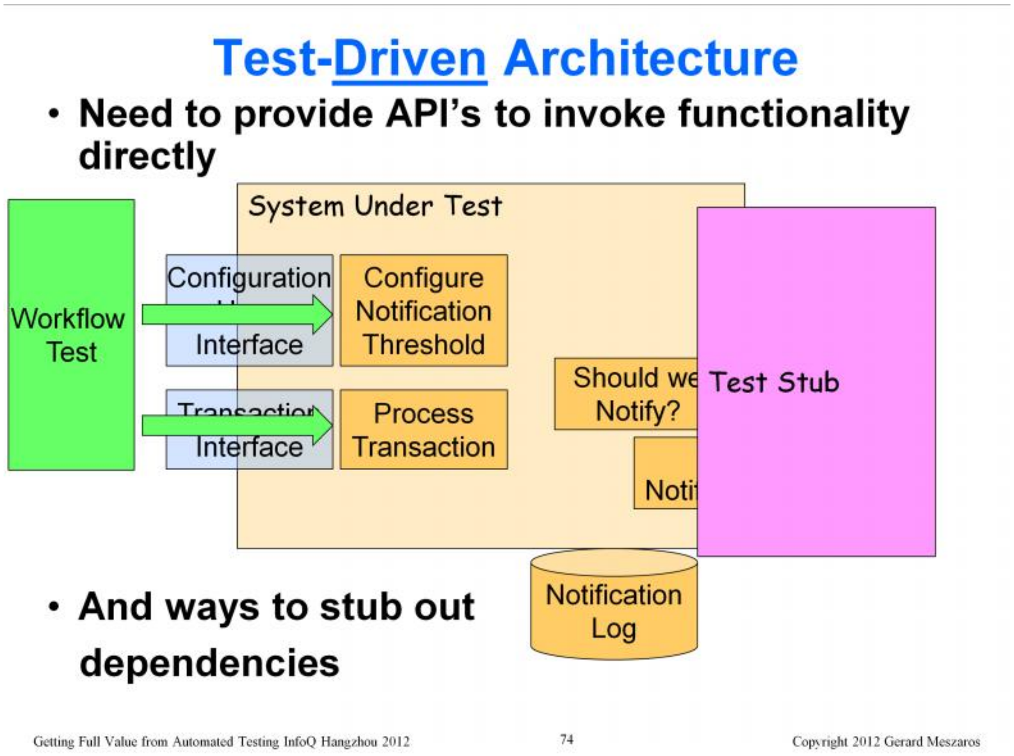
## Test - After Architecture

- **Must test through User Interface**

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 73 Copyright 2012 Gerard Meszaros

The problem with trying to automate the tests after we’ve finished building the system is that it is very difficult or even impossible to do effectively because the system wasn’t designed with testability in mind. We are typically forced to test too much code and via awkward interfaces such as the user interface. And it is very difficult to control the behavior of all those other things our application logic is expected use as input.

This is why test automation done by independent test groups is often an outright failure. And why the most successful test automation is seen on eXtreme Programming teams where the testing is part of the team’s job, not relegated to a separate group.



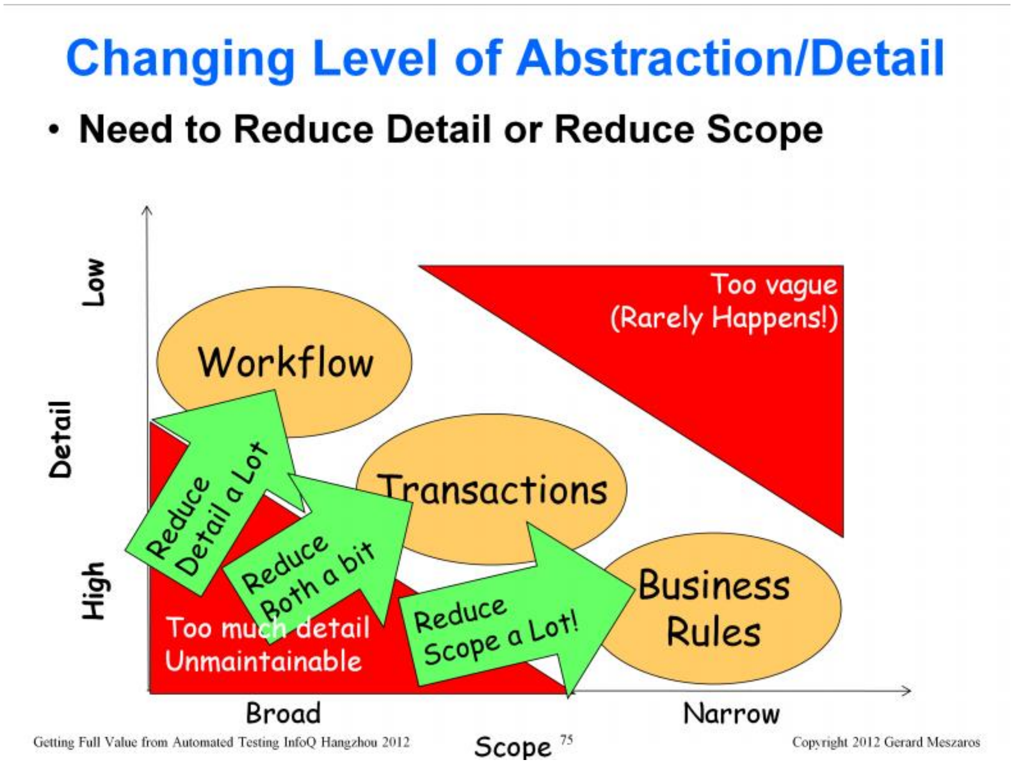
# Test-Driven Architecture

- Need to provide API's to invoke functionality directly

- And ways to stub out dependencies

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 74 Copyright 2012 Gerard Meszaros

When the team building the application is also responsible for testing it, they are highly motivated to make test automation easy. In fact, they will typically start the design process by deciding what kinds of tests they want to be able to automate and what affordances the application needs to provide the automated tests. When this kind of thinking is applied at the Component and System levels of the test automation pyramid, I call it test-driven architecture. That is, the architecture of the application is shaped by the test requirements as much as by the functional requirements. For example, to automate workflow tests of a business process, we want to be able to bypass the user interface so that the tests can be expressed in terms of business process actions, not UI actions. And we may need to be able to control the behaviour of certain sub-components to ensure that they provide the particular response our test case requires. We often fulfil this requirement by providing a means to stub out specific components dynamically in our test environment.



## Changing Level of Abstraction/Detail

- Need to Reduce Detail or Reduce Scope

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 75 Copyright 2012 Gerard Meszaros

It is important to specify each story at the right level.

Stories with very broad scope (such as the end-to-end process) should be specified with a minimum of detail.

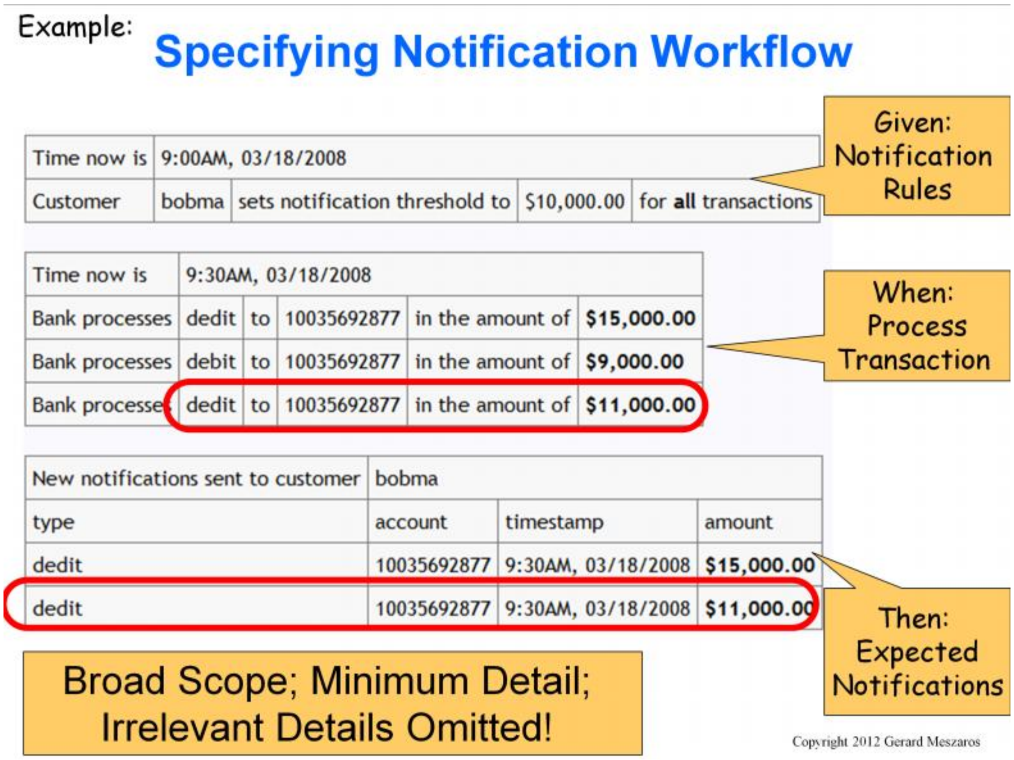
Stories that specify a great deal of detail  should be kept very narrow in scope.

Specifying broad scope in high detail results in a lot of duplicated and unmaintainable detail.

Specifying with too little detail for narrow scope results in too many specs that say very little.

The goal is to make our examples and tests each to understand and easy to write. This requires using the right language in each spec.

That test we just wrote falls into the bottom right quadrant; too much detail for the broad scope it encompasses. So we need to reduce the level of detail.



Example:

## Specifying Notification Workflow

| | | | | |
|---|---|---|---|---|
| Time now is | 9:00AM, 03/18/2008 | | | |
| Customer | bobma | sets notification threshold to | $10,000.00 | for **all** transactions |

Given: Notification Rules

| | | | | | |
|---|---|---|---|---|---|
| Time now is | 9:30AM, 03/18/2008 | | | | |
| Bank processes | dedit | to | 10035692877 | in the amount of | **$15,000.00** |
| Bank processes | debit | to | 10035692877 | in the amount of | **$9,000.00** |
| Bank processes | dedit | to | 10035692877 | in the amount of | **$11,000.00** |

When: Process Transaction

| | | | |
|---|---|---|---|
| New notifications sent to customer | bobma | | |
| type | account | timestamp | amount |
| dedit | 10035692877 | 9:30AM, 03/18/2008 | **$15,000.00** |
| dedit | 10035692877 | 9:30AM, 03/18/2008 | **$11,000.00** |

Then: Expected Notifications

Broad Scope; Minimum Detail; Irrelevant Details Omitted!

Copyright 2012 Gerard Meszaros

The overall workflow should be defined at a very high level. Only the essential details should be specified here because we'll need to touch on many of the transactions and providing too much detail would make these specs too hard to understand. The details can be provided in other specs.

Can be defined before user interface is designed.

Should be defined before software is built

Can (and should) be automated bypassing the user interface

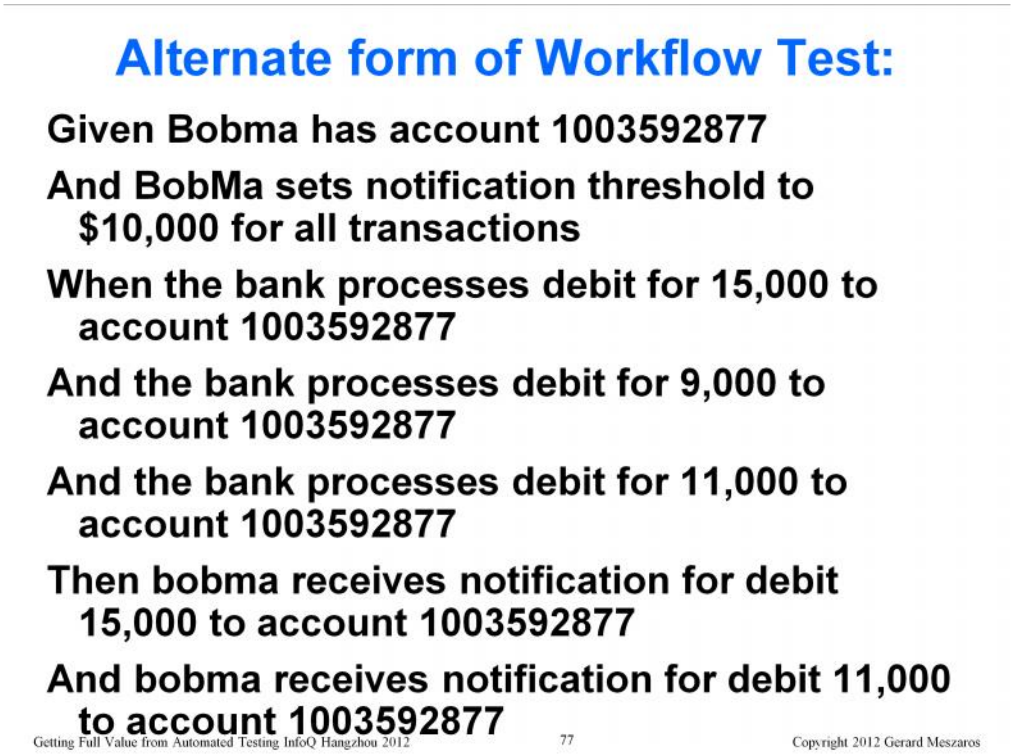
Should be run by developers as they build the software

Not the right way to test detailed functionality

    Too high level

Is it right way to test notification algorithms?

    Maybe too cumbersome



## Alternate form of Workflow Test:

Given Bobma has account 1003592877

And BobMa sets notification threshold to $10,000 for all transactions

When the bank processes debit for 15,000 to account 1003592877

And the bank processes debit for 9,000 to account 1003592877

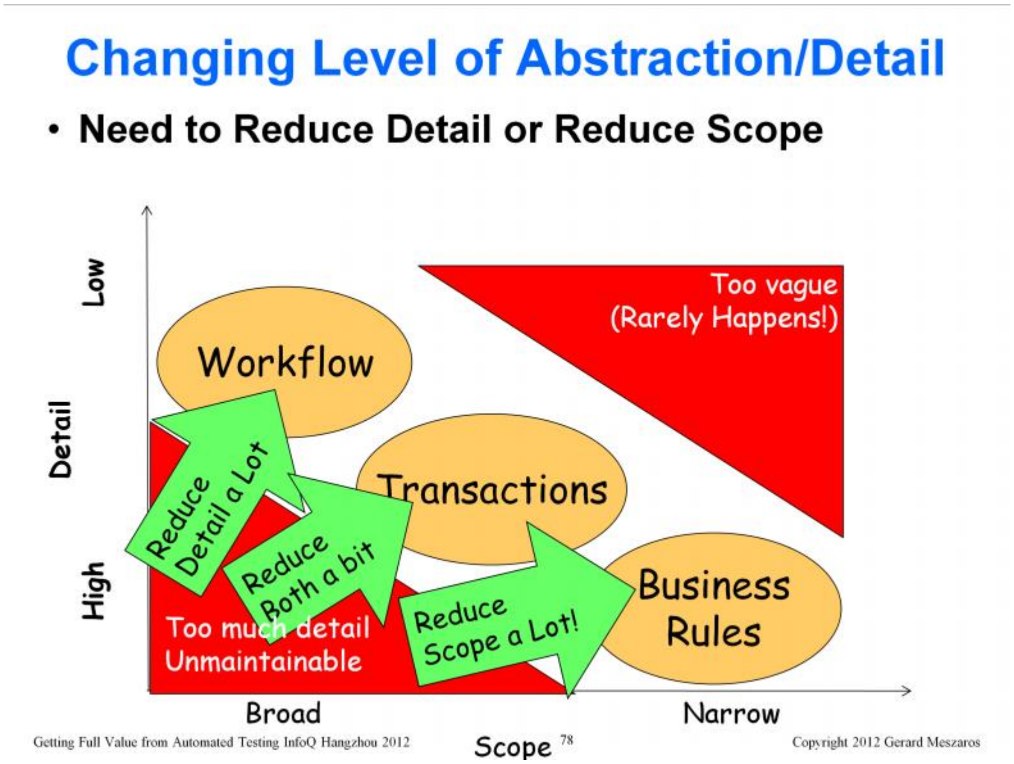
And the bank processes debit for 11,000 to account 1003592877

Then bobma receives notification for debit 15,000 to account 1003592877

And bobma receives notification for debit 11,000 to account 1003592877

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 77 Copyright 2012 Gerard Meszaros

Here’s another way to express the same test using the given-when-then terminology in a text-based domain-specific language.

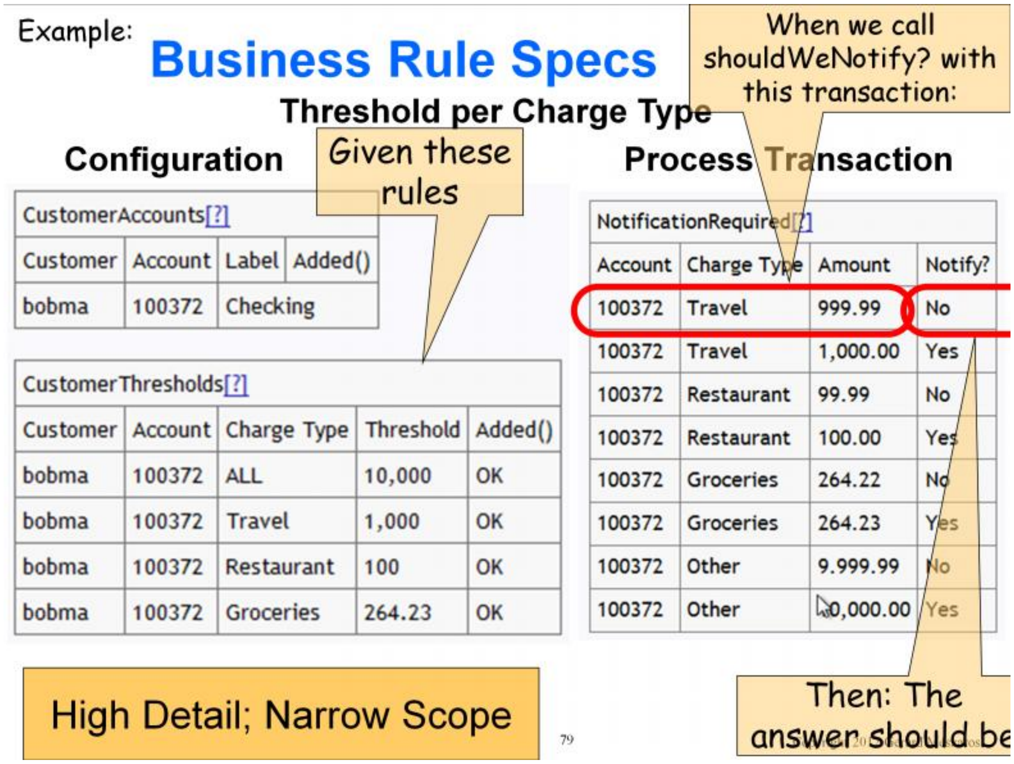


## Changing Level of Abstraction/Detail

- **Need to Reduce Detail or Reduce Scope**

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 78 Copyright 2012 Gerard Meszaros

How can we test the business rules around notification more effectively? Testing this via the overall workflow would be very tedious and slow. It takes too many steps and too much extraneous detail to do this effectively. The alternative is to reduce the scope drastically to just the part of the system that implements the decision whether to notify.



Example:

## Business Rule Specs

### Threshold per Charge Type

**Configuration**

| CustomerAccounts[?] | | | |
|---|---|---|---|
| Customer | Account | Label | Added() |
| bobma | 100372 | Checking | |

| CustomerThresholds[?] | | | | |
|---|---|---|---|---|
| Customer | Account | Charge Type | Threshold | Added() |
| bobma | 100372 | ALL | 10,000 | OK |
| bobma | 100372 | Travel | 1,000 | OK |
| bobma | 100372 | Restaurant | 100 | OK |
| bobma | 100372 | Groceries | 264.23 | OK |

**Process Transaction**

| NotificationRequired[?] | | | |
|---|---|---|---|
| Account | Charge Type | Amount | Notify? |
| 100372 | Travel | 999.99 | No |
| 100372 | Travel | 1,000.00 | Yes |
| 100372 | Restaurant | 99.99 | No |
| 100372 | Restaurant | 100.00 | Yes |
| 100372 | Groceries | 264.22 | No |
| 100372 | Groceries | 264.23 | Yes |
| 100372 | Other | 9.999.99 | No |
| 100372 | Other | 10,000.00 | Yes |

Given these rules

When we call shouldWeNotify? with this transaction:

Then: The answer should be

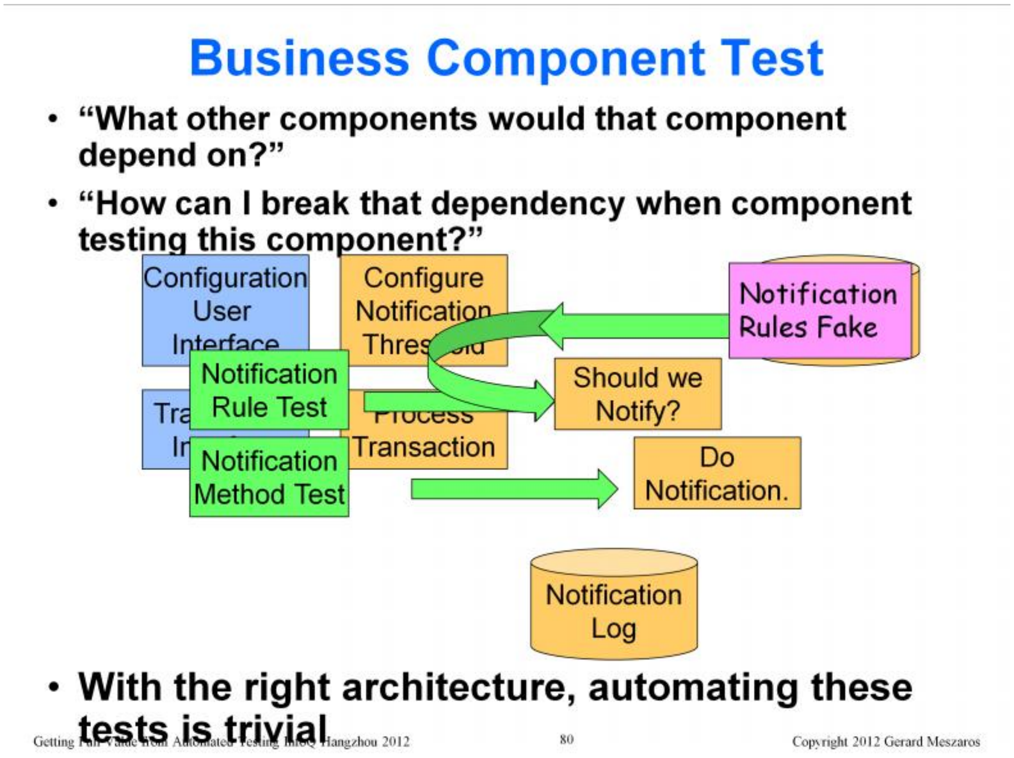
High Detail; Narrow Scope

79

Now that we have decide to isolate the Notification Logic, we can specify it’s behaviour using component tests. Here we are using Fit Column fixtures to configure the data (on the left side of the slide) and to invoke the Should We Notify component. Each row in the table on the right is one test. The first 3 columns are the inputs to the notification decision and the last column is the expected result. Once again, we can easily read the test conditions directly from these tests. Given these thresholds, when we call shouldWeNotify? With Account 100372 with Travel charge for 999.99, the answer should be No.

This approach allows us to test algorithms and business rules without overhead of use case or workflow tests.

Require access to component(s) that implement the business rules and that encourages a more modular software design.



## Business Component Test

- “What other components would that component depend on?”
- “How can I break that dependency when component testing this component?”

Configuration User Interface
Configure Notification Thres old
Notification Rules Fake
Notification Rule Test
Should we Notify?
Tra In
Process Transaction
Notification Method Test
Do Notification.
Notification Log

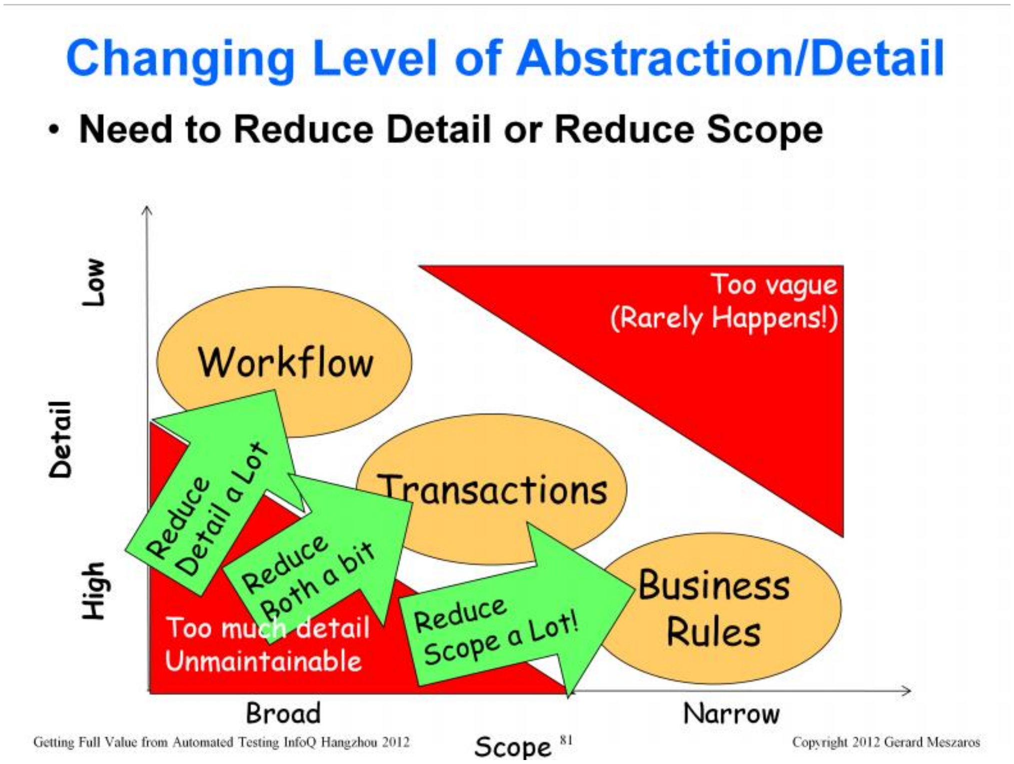
- With the right architecture, automating these tests is trivial

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 80 Copyright 2012 Gerard Meszaros

This requires us to structure the system so that the notification logic is easily accessed via an API thus allowing our tests to focus on what they want the Should We Notify component to do, not how to interact with it via the Process Transaction components.

We also want to make it easy to provide the rules to the component directly from the test so we can bypass the Configuration component.

We can achieve all this by asking the questions [READ FROM THE SLIDE]. This leads us to a testable architecture where the Should We Notify component is passed the Notification Rules by it’s caller, something I call Data Injection.



## Changing Level of Abstraction/Detail

- **Need to Reduce Detail or Reduce Scope**

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 81 Copyright 2012 Gerard Meszaros

It is important to specify each story at the right level.

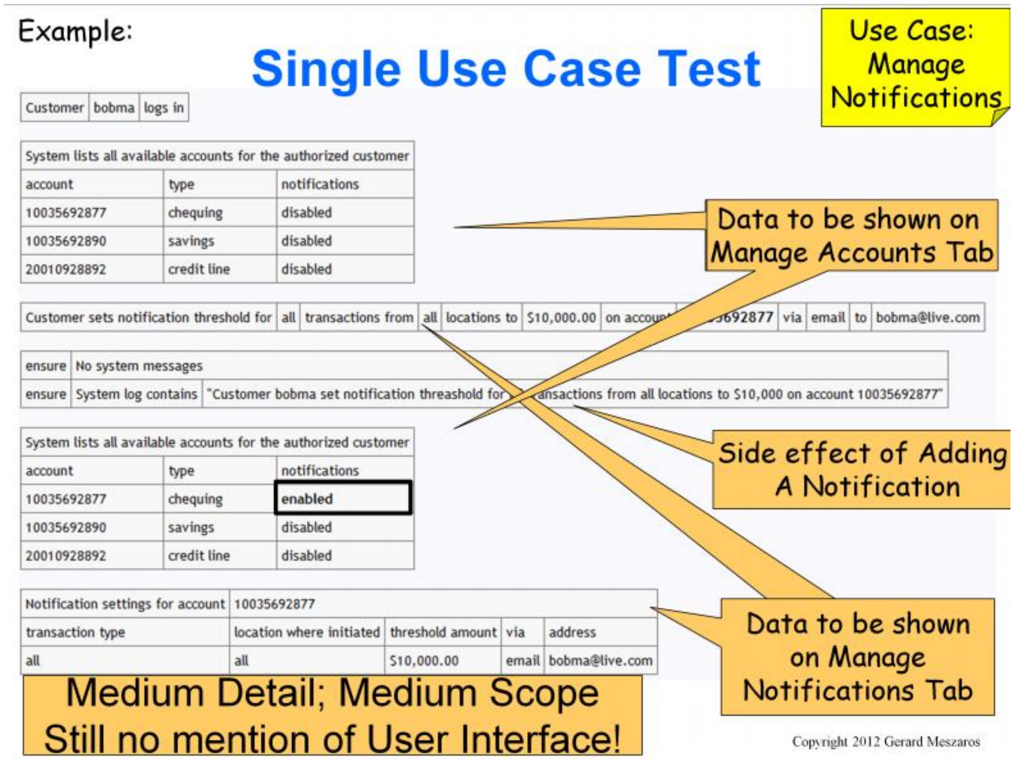
Stories with very broad scope (such as the end-to-end process) should be specified with a minimum of detail.

Stories that specify a great deal of detail  should be kept very narrow in scope.

Specifying broad scope in high detail results in a lot of duplicated and unmaintainable detail.

Specifying with too little detail for narrow scope results in too many specs that say very little.

The goal is to make our examples and tests each to understand and easy to write. This requires using the right language in each spec.



Example:

## Single Use Case Test

Use Case: Manage Notifications

| | | |
|---|---|---|
| Customer | bobma | logs in |

| System lists all available accounts for the authorized customer | | |
|---|---|---|
| account | type | notifications |
| 10035692877 | chequing | disabled |
| 10035692890 | savings | disabled |
| 20010928892 | credit line | disabled |

Data to be shown on Manage Accounts Tab

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Customer sets notification threshold for | all | transactions from | all | locations to | $10,000.00 | on account | 10035692877 | via | email | to | bobma@live.com |

| | | |
|---|---|---|
| ensure | No system messages | |
| ensure | System log contains | "Customer bobma set notification threashold for all transactions from all locations to $10,000 on account 10035692877" |

Side effect of Adding A Notification

| System lists all available accounts for the authorized customer | | |
|---|---|---|
| account | type | notifications |
| 10035692877 | chequing | **enabled** |
| 10035692890 | savings | disabled |
| 20010928892 | credit line | disabled |

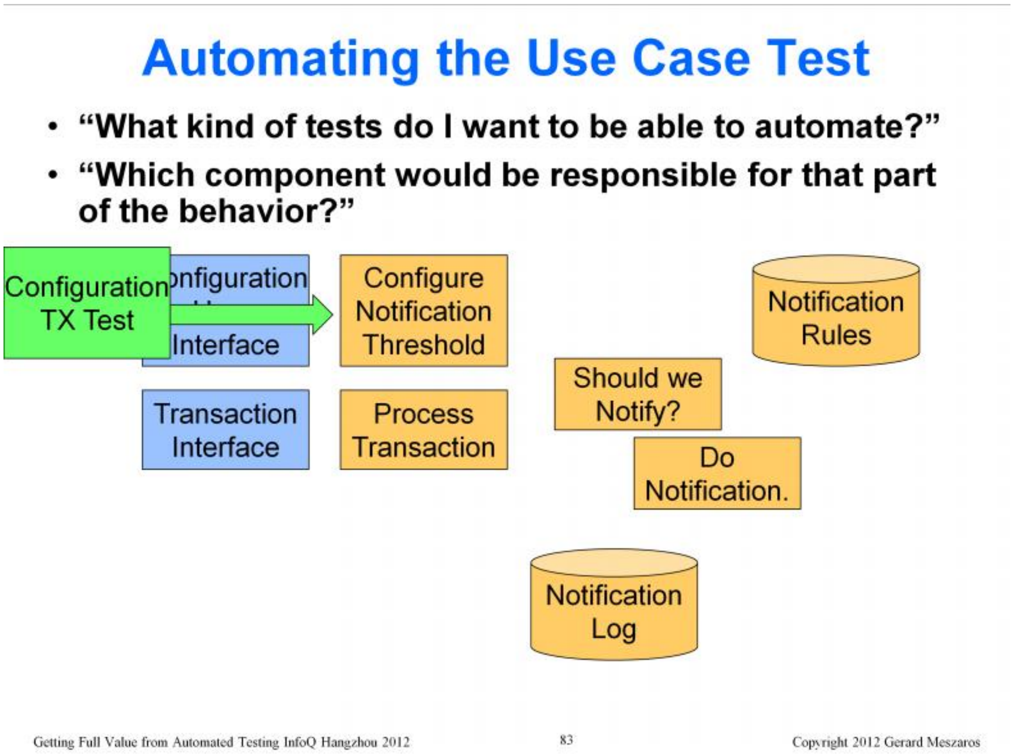
| Notification settings for account | 10035692877 | | | |
|---|---|---|---|---|
| transaction type | location where initiated | threshold amount | via | address |
| all | all | $10,000.00 | email | bobma@live.com |

Data to be shown on Manage Notifications Tab

Medium Detail; Medium Scope
Still no mention of User Interface!

The  overall workflow specification helps us understand the big picture but we want to make sure we understand how each transaction, such as configuring a notification rule, actually works. We can write a specification for this using a walk-through the UI as an inspiration. We want to ensure that the behavior of the system behind the UI at each step is clearly understood. This spec helps us do that. Each piece of data on the screen and each action accessible from it is represented in this transaction spec. That way we can be sure that the code behind the screen is implemented properly.

Note that there is a lot more detail shown here than in the workflow spec where it only took a single line to describe the equivalent of what we are doing here.

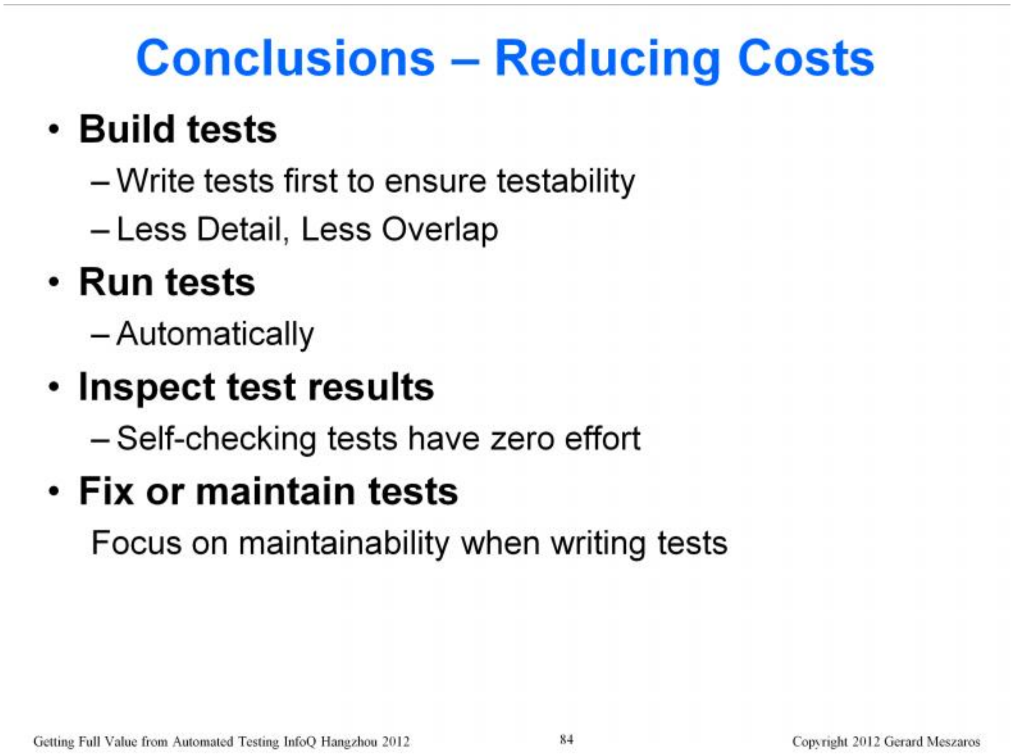


## Automating the Use Case Test

- “What kind of tests do I want to be able to automate?”
- “Which component would be responsible for that part of the behavior?”

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 83 Copyright 2012 Gerard Meszaros

To automate the use case tests, we expose the appropriate API on the configuration interface. This is most likely the same interface used by the User Interface. Depending on the nature of the test, we may choose to stub out the database or include it within the scope of the system.



## Conclusions – Reducing Costs

- **Build tests**
  - Write tests first to ensure testability
  - Less Detail, Less Overlap
- **Run tests**
  - Automatically
- **Inspect test results**
  - Self-checking tests have zero effort
- **Fix or maintain tests**

  Focus on maintainability when writing tests

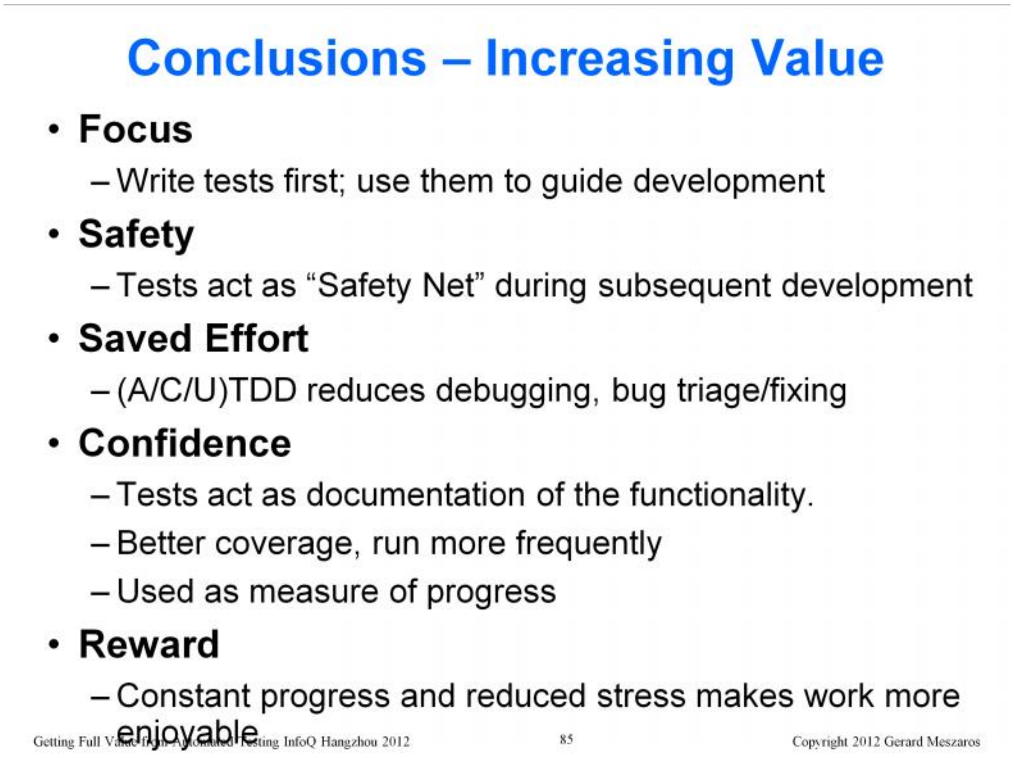
Maximizing Value for effort spent requires us to reduce cost and increase value.

We can reduce cost and effort by

•avoiding unnecessary detail in tests and reducing the overlap between tests.

•running tests automatically and ensuring that tests are self-checking so we don’t need to look at them every time they are run.

•We can reduce the Cost of Ownership of the product and it’s tests by focusing on maintainability when writing the tests.



## Conclusions – Increasing Value

- **Focus**
  - Write tests first; use them to guide development
- **Safety**
  - Tests act as “Safety Net” during subsequent development
- **Saved Effort**
  - (A/C/U)TDD reduces debugging, bug triage/fixing
- **Confidence**
  - Tests act as documentation of the functionality.
  - Better coverage, run more frequently
  - Used as measure of progress
- **Reward**
  - Constant progress and reduced stress makes work more enjoyable

Getting Full Value from Automated Testing InfoQ Hangzhou 2012 85 Copyright 2012 Gerard Meszaros

To summarize the value provided by our automated tests:

•We write the tests first so that they guide us during development

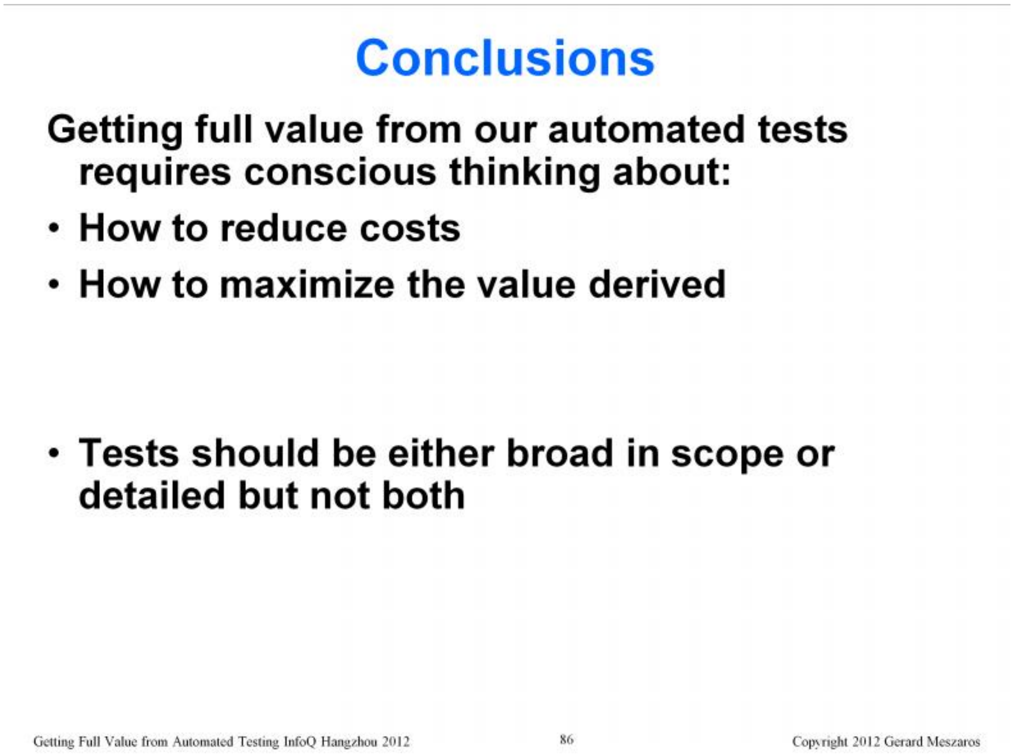
•The tests act as a safety net during subsequent development.

•Driving development with Acceptance, Component and Unit tests reduces the number of defects we put into the code thereby reducing the amount of debugging we need to do and the number of founds that are found, have to be triaged and managed until they are fixed.

•The Tests act as documentation of the functionality giving us confidence in our knowledge of what the product does and how it works.

•Running the tests frequently gives us better confidence in the quality of our product. And we can use the number of acceptance and component tests passing as a measure of progress.

•The constantly visible process and the reduced stress of working with a safety net of tests make work more enjoyable.



## Conclusions

Getting full value from our automated tests requires conscious thinking about:

- How to reduce costs
- How to maximize the value derived

- Tests should be either broad in scope or detailed but not both

XUnit Test Patterns
Refactoring Test Code
Gerard Meszaros
Acceptance Test Engineering
How to Decide if Software is Ready for You or Your Customers
patterns & practices

## Thank You!

**Gerard Meszaros**

**infoQ2012hz@gerardm.com**

**http://www.xunitpatterns.com**

**http://GettingFullValue.gerardm.com**

Jolt Productivity Award winner - Technical Books

http://testingguidance.codeplex.com/

**Call me when you:**

- **Want to transition to Agile or Lean**
- **Want to do Agile or Lean better**
- **Want to teach developers how to test**
- **Need help with test automation strategy**
- **Want to improve your test automation**