Keeping Software Soft

Gerard Meszaros
India2011@gerardm.com

My Background

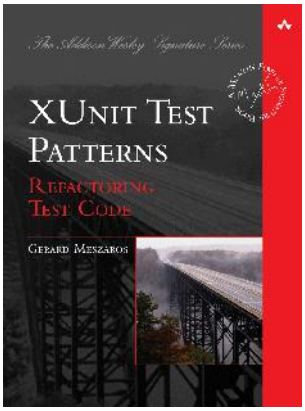
- Software developer
- Development manager
- Project Manager
- Software architect

Embedded Telecom

- OOA/OOD Mentor
- XP/TDD Mentor
- Agile PM Mentor

I.T.

- Test Automation Consultant
- Author
- Lean/Agile Coach/Consultant



Gerard Meszaros
Scrum2011@xunitpatterns.com

soft·ware

From: http://dictionary.com

Software

soft·ware [sawft-wair, soft-]

–noun

1. **Computers . the programs used to direct the operation of a computer, as well as documentation giving instructions on how to use them. Compare** hardware **(def. 5) .**
2. **anything that is not hardware but is used with hardware, especially audiovisual materials, as film, tapes, records, etc.: a studio fully equipped but lacking software.**
3. ~~**Television Slang . prepackaged materials, as movies or reruns, used to fill out the major part of a station's program schedule.**~~

From: http://dictionary.com

Ware

ware [wair]

–noun

1. **Usually, wares.**
   1. a. articles of merchandise or manufacture; goods: a peddler selling his wares.
   2. any intangible items, as services or products of artistic or intellectual creativity, that are salable: an actor advertising his wares.
2. ~~**a specified kind or class of merchandise or of manufactured article (usually used in combination): silverware; glassware.**~~
3. ~~**pottery, or a particular kind of pottery: delft ware.**~~

From: http://dictionary.com

Soft

soft [sawft]

–adjective

1. yielding readily to touch or pressure; easily penetrated, divided, or changed in shape; not hard or stiff: a soft pillow.
2. ~~relatively deficient in hardness, as metal or wood.~~
3. ~~smooth and agreeable to the touch; not rough or coarse: a soft fabric; soft skin.~~

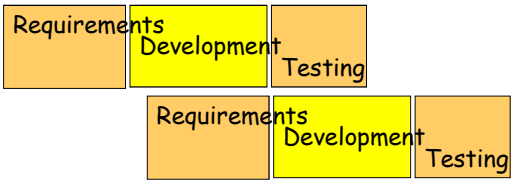
More Appropriate Names for Software?

- Slow-ware -- Slow to produce
- Finnicky-ware -- Hard to get right
- Brittle-ware -- Hard to change

- Any other suggestions?

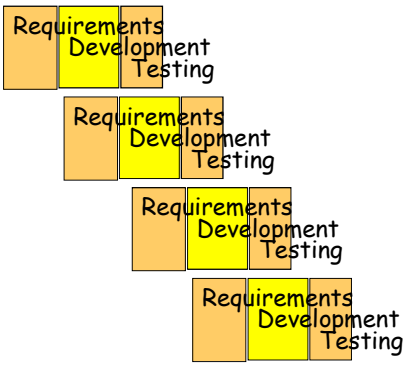
Why Do We Care (How "Soft" it is?)

Why Do We Care (How "Soft" it is?)



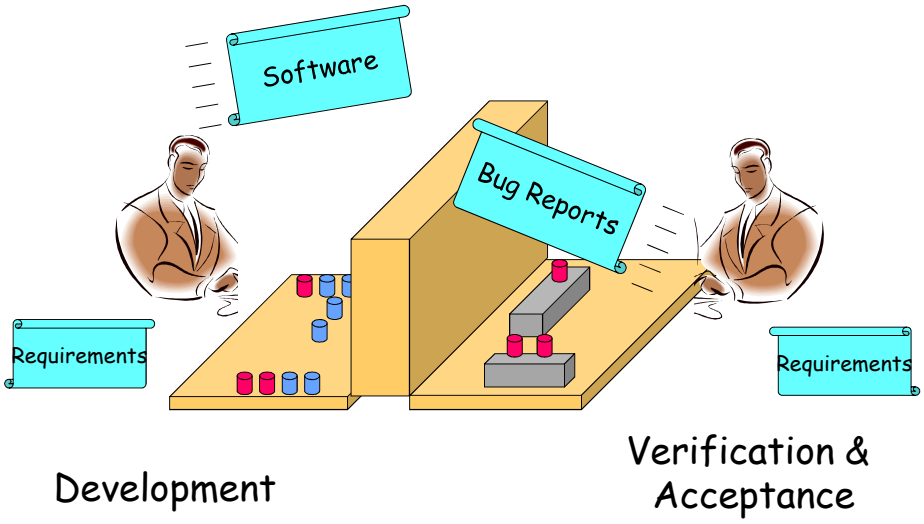
- **As development increments reduce in duration, testing needs to be reduced accordingly**

Why Do We Care (How "Soft" it is?)



... and traditional approaches to development no longer work

Test&Fix Ping-Pong

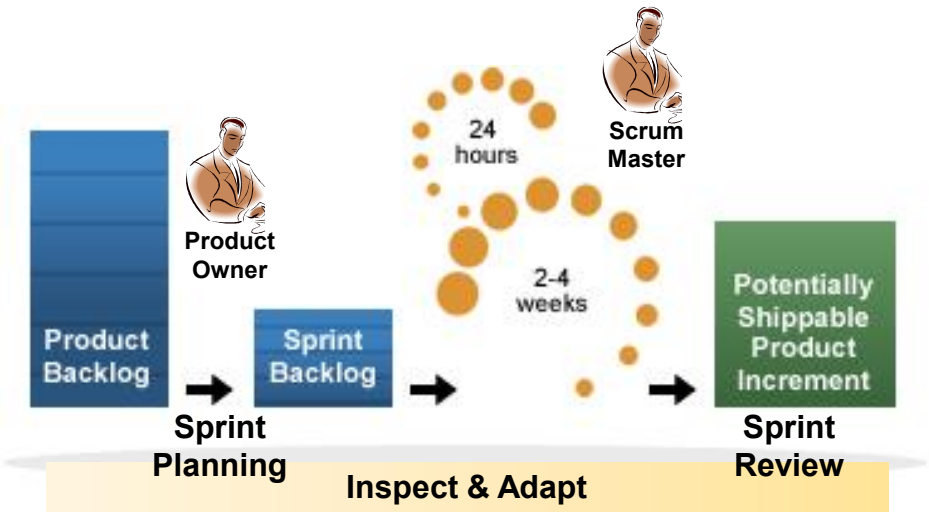


A Word of Cautionary:

- When the US automakers implemented Lean, they copied the practices
- Some of the culture & principles were skipped
- The results were less than ideal

Practices are not enough!

Scrum – The Most Popular Agile Method



 Added from Ron Jeffries

Why Is Scrum Successful?

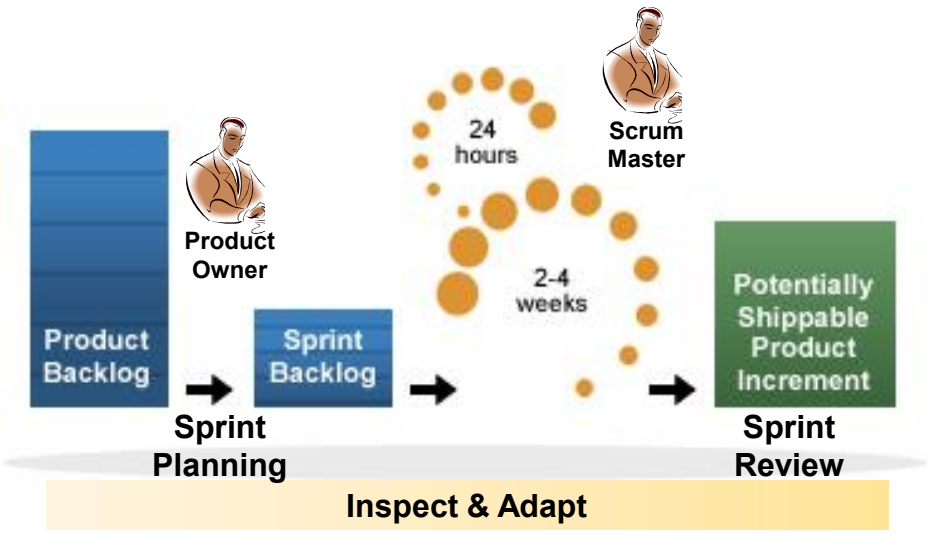
- **Encourages focus on delivery**
- **Encourages teamwork & self-determination**
- **Discourages management meddling**

- **Early to Certification Game**
  - Gained huge mindshare through thousands of CSM's
- **Doesn't impose engineering practices**
  - Assumes you have good ones!

Why Does Scrum Often Fail?

- **Incomplete adoption of Scrum(But)**
- **Focus on the practices**
  - all management
- **Unsustainable engineering/technical practices**

Scrum – The Most Popular Agile Method



 Added from Ron Jeffries

Key Practice: Inspect & Adapt

- Make problems visible (Inspect)
- Change the process to address them (Adapt)
- Repeat Forever

- Probably the most important part of Scrum
- And the least well implemented.

But What Do We Change?

- What would an appropriate highly incremental development process (that keeps software soft) look like?

- How long would it take us to evolve there using Inspect & Adapt?

Design a New Process from Scratch

- **Determine the Characteristics we Desire**
- **Pick the Practices that will Give us These Characteristics**
- **Integrate Them Into a Methodology**

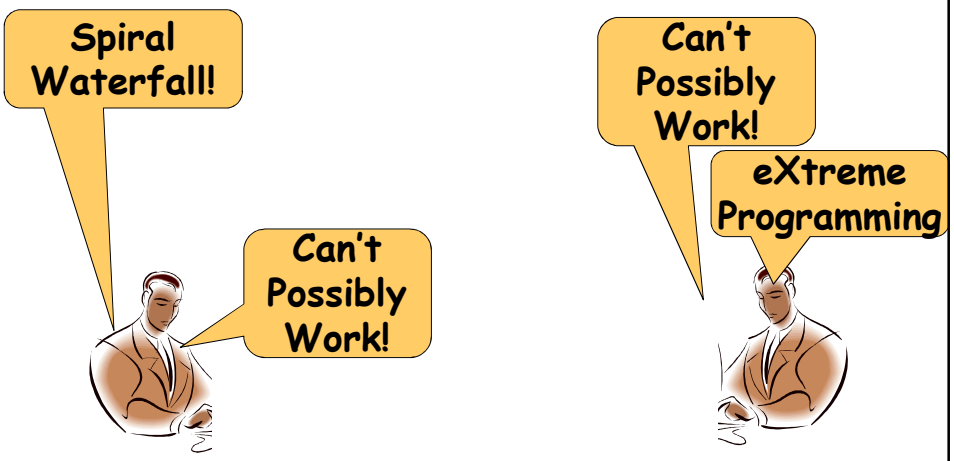
- **Takes a detailed understanding of:**
  - Each practice, and
  - How the Practices Interact
- **Cannot be acquired without actual experience**
- **The people with the power (process police) don' t have the experience**

Adopt and Inspect

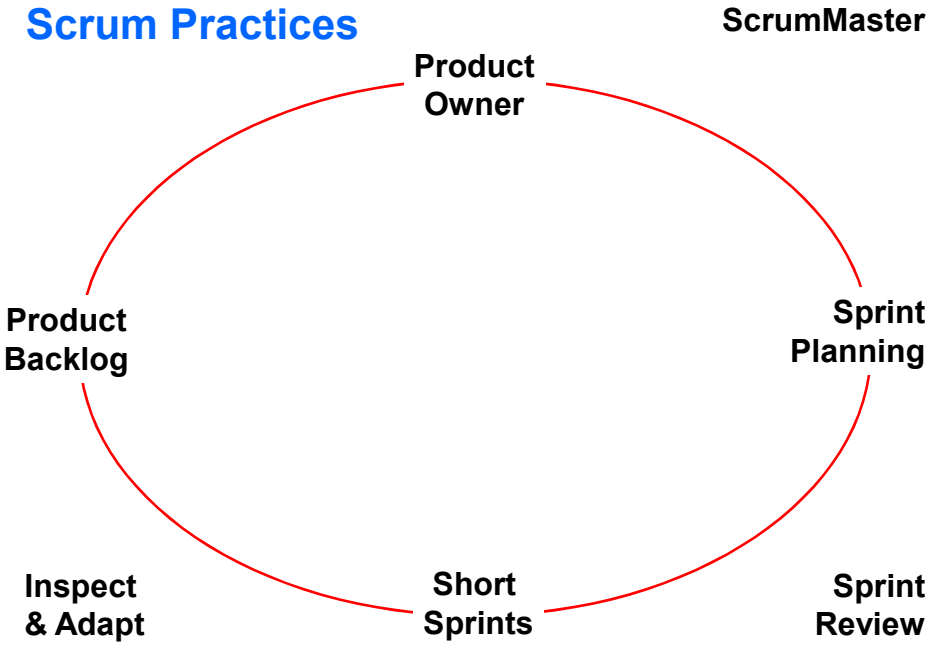
- **Find a Development Process that's known to work**
- **Adopt it**
- **Inspect the Results & Adapt**

- **Example: Scrum with XP Inside™**
  - XP = eXtreme Programming

Scrum with XP Inside™

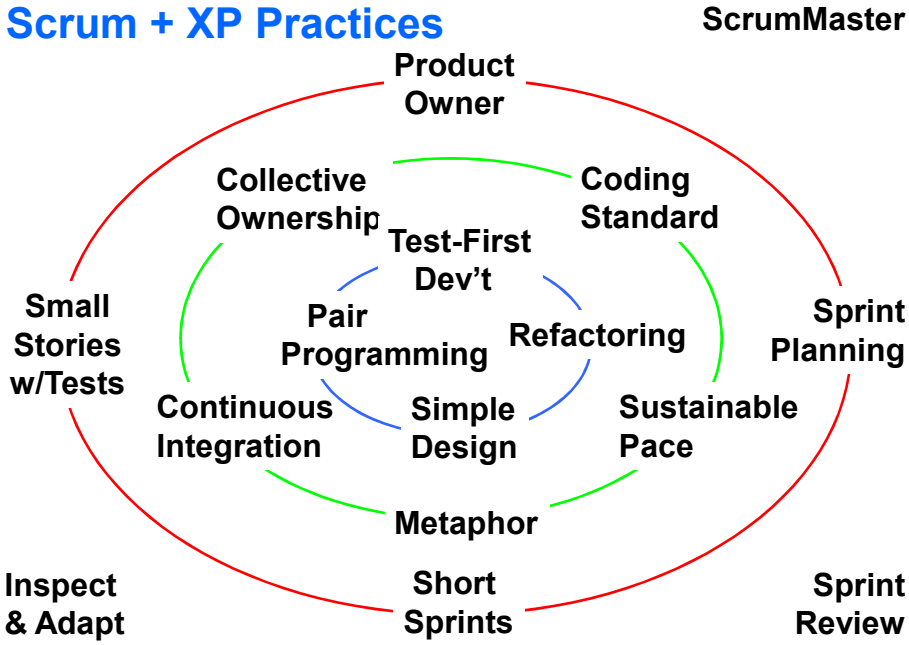


Scrum Practices



 Added from Ron Jeffries

Scrum + XP Practices



 Courtesy of Ron Jeffries

Key Requirement Practices

- **Small Increments of Functionality**
  - Small, testable user stories
  - Enables continuous flow of functionality
  - Can be finished in a single sprint
- **Acceptance (Story) Test Driven Development**
  - Also known as Example-Driven Development
  - Concrete examples of expected results
  - Avoids Test&Fix Ping Pong

I talk about these in my session: User Stories – The Whole Story

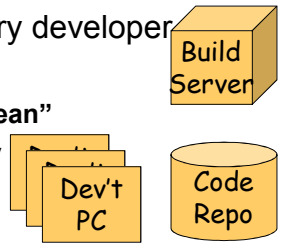
Key Engineering Practices

- **Continuous Integration**
  - Frequent check-ins reduce integration debt
- **Automated Functional Testing**
  - Detect changes in behaviour quickly
  - Ensures same tests run every time
- **Automated Unit Testing**
  - Improves automated test coverage
  - Detects changes faster with less effort
- **Refactoring**
  - Improving the design of code incrementally

Continuous Integration

Consists of 3 essential components:

1. **Build Server**
   - With software to rebuild the system every time code is checked in.
2. **Automated Tests**
   - To verify the code works (compile+link isn't enough)
3. **Frequent Check-ins**
   - At least once a day, on average, by every developer
   - Run all unit tests before checking in.
     - » **"Keep the Bar Green to Keep the Code Clean"**
     - » **Requires automated tests that run quickly**



Automated Testing

- **Required to keep the cost of regression testing low enough to do for every build.**
- **Need functional tests to ensure functionality valued by stakeholders still works.**
- **Need unit or component tests to get good enough test coverage**
  - Code valued by developers/testers (error/exception scenarios) often cannot be hit by pure functional tests
- **But, we need all these tests *before* we check in the code**
  - So that CI can catch any errors on future check-in

Implies TDD

Where Does This Leave Us?

- **We can add new code**
- **We can change existing code**
- **We can find out whether we broke anything**
- **Anyone can change anything, safely**

But Does this make the Software "soft"

And Where Does the Software Design Happen?

Refactoring (from a Simple Design)

Improving the design of existing code without changing its functionality

- **Should be done in small steps so system always works (never broken)**

Requires understanding of:

- **What good design looks like (e.g. Patterns),**
- **And the transformations (refactoring moves)**

Is a High Skill Activity

Less Risky Than Re-Architecting!

Refactoring vs. Prefactoring

- **"Architecture" or "Design Up Front" may work for known requirements**
  - But speculative frameworks are often hard to use
- **For yet unknown requirements, only refactoring will help**
- **Emergent frameworks are easier to use**
  - because they are simpler (based on actual usage patterns)

You Will Need to Refactor;
So Learn It Beforehand!

Contentious Engineering practices

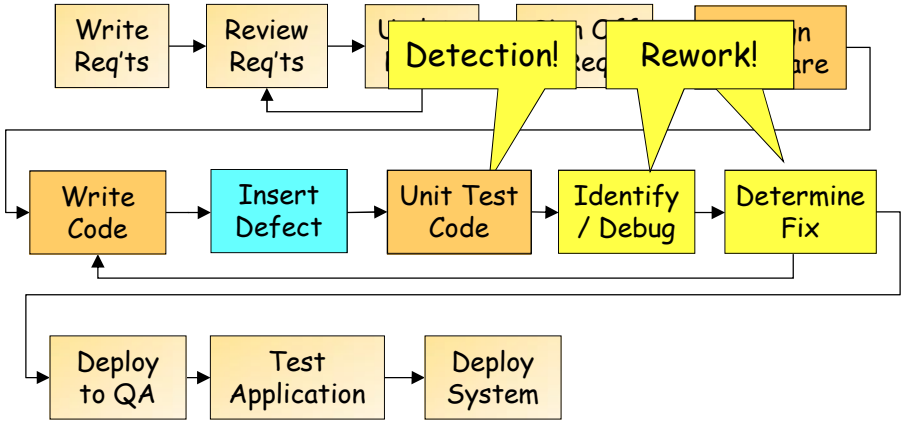
- **Test-Driven Development**
  - Ensures unit tests are written
  - Avoids untestable code
  - Improves the design
  - Reduces the amount of time wasted debugging
- **Pair Programming**
  - Avoids wasting time on dead ends
  - Ensures required discipline occurs

Goals of Automated Developer Tests

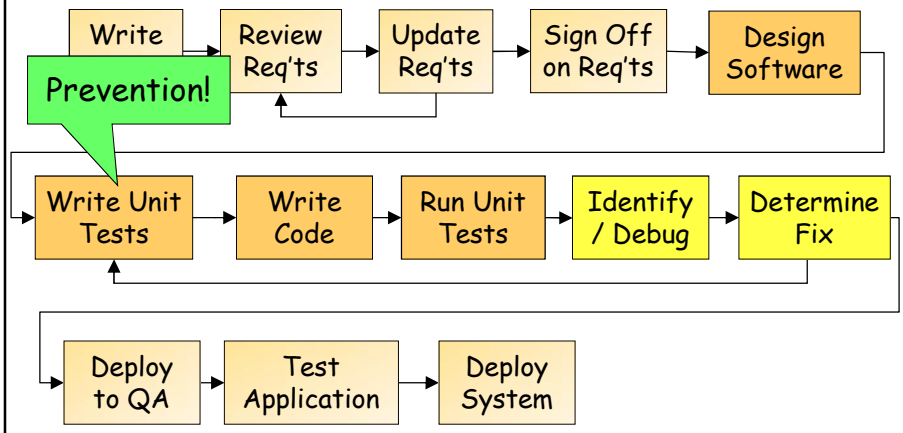
- **Before code is written**
  - Tests as Specification
- **After code is written**
  - Tests as Documentation
  - Tests as Safety Net (Bug Repellent)
  - Defect Triangulation (Minimize Debugging)
- **Minimize Cost of Running Tests**
  - Fully Automated Tests
  - Repeatable Tests
  - Robust Tests

Requires writing tests *before* code (TDD)

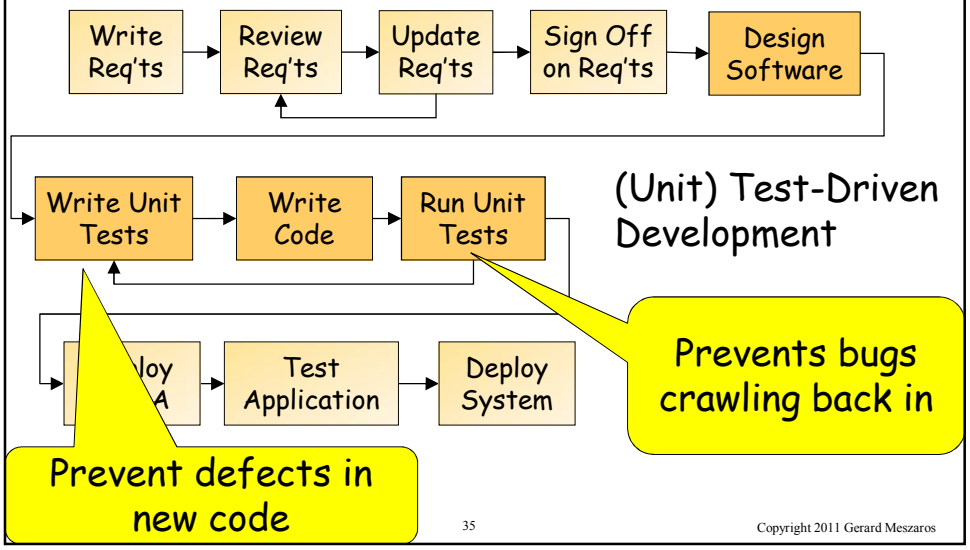
Preventing Coding Defects
(Building the Right Product)



Preventing Coding Defects
(Building the Right Product)



Preventing Coding Defects
(Building the Right Product)



Isn't TDD Redundant?

- **Expect to write at least as much test code as production code!**
- **Won't that double the cost of building the product?**
- **No! If you do it right, it will reduce the cost.**

TDD Rhythm

Test Code

- First Test
- Second Test
- Third Test
- Fourth Test

Product Code

- Hard-coded method
- Introduce variable
- Introduce conditional
- Surround with a loop

Just Like Double-Entry Booking:

- An entry on the test side for each entry on the Prod side

Where Does This Leave Us?

Continuous Integration (all 3 parts)

+ Automated Tests

+ Test-Driven Development

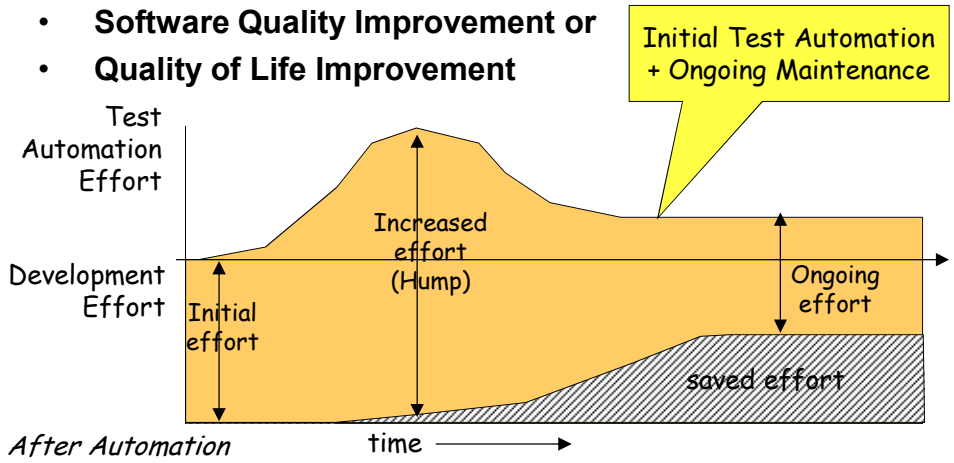
+ Refactoring

~~"Soft" Software~~

Economics of Maintainability

Test Automation is a lot easier to sell on

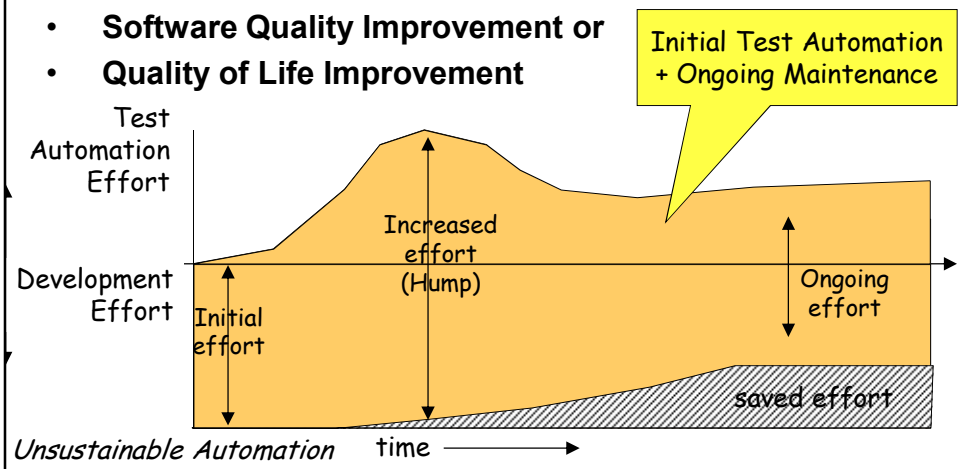
- **Cost reduction than**
- **Software Quality Improvement or**
- **Quality of Life Improvement**



Economics of Maintainability

Test Automation is a lot easier to sell on

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



(Re)Factoring 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(
        assertEquals(invoice,
             actualLineItem.getInvoic
        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_severalQuantity
() {
  QUANTITY = 5 ;
  product = createProduct();
  invoice = createAInvoice();
  // Exercise SUT
  invoice.addItemQuantity(
          product, QUANTITY);
  // Verify Outcome
  expectedItem = newLineItem(
    invoice, product, QUANTITY,
    product.getPrice()*
    QUANTITY);
  assertExactlyOneLineItem( invoic
    expectedItem );
}
```


Refactored Test Code

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

The Least Utilized Practice: Pairing

Pairing is two people working together at the same computer

- **A Developer + Tester writing tests**
- **An Analyst + Developer defining requirements**
- **Two PM's defining inter-dependencies**
- **Two Developers writing code & tests**

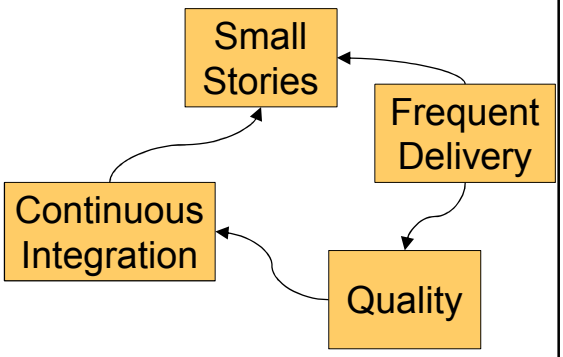
- **Isn't this wasteful?**

Isn't Pairing wasteful?

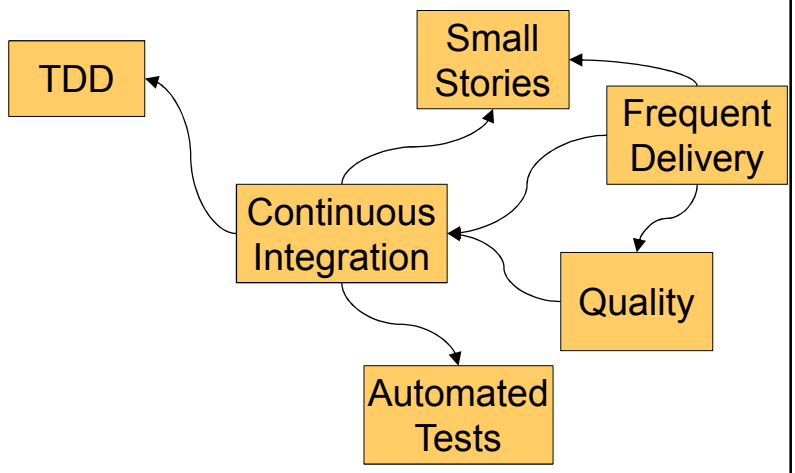
Two minds produce better product, faster

- **Pilot + Co-pilot**
  - "50 metres (elevation), 40 metres, ..."
- **Driver + Navigator**
  - The next turn is 110degrees at 80 km/h
- **Plumber + apprentice/helper**
  - "I'll hold this pipe while you attach that end."
- **Two Developers:**
  - "Shouldn't we write a test first?"
  - "Wouldn't an Iterator be a better way of doing that?"

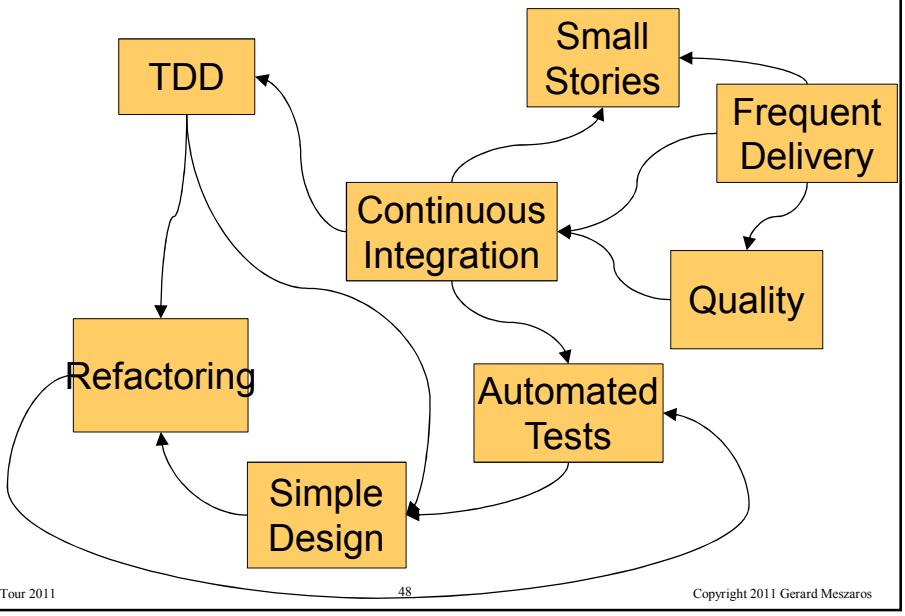
An Ecosystem of Practices



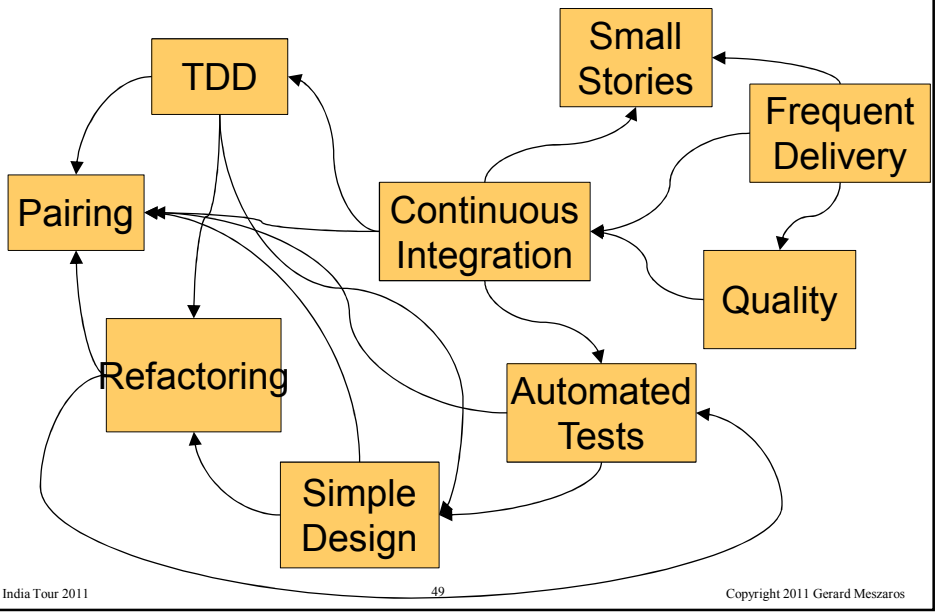
An Ecosystem of Practices



An Ecosystem of Practices



An Ecosystem of Practices



Conclusions

Highly Incremental Development

Requires a Change in How We Build Software

- **To Deliver Continuous Stream of Value**
- **To Reduce the Cost of Change**
- **By Reducing the Likelihood of Inserting Defects**
- **And to Speed Up the Detection of New Defects**

Conclusions (2)

Highly Incremental Development

Requires:

- **Smaller Stories/Features**
- **Continuous Integration (all 3 parts!)**
- **Test-Driven Development (Acceptance & Unit)**
- **Automated Test Execution**
- **Close Teamwork including Pairing**

Conclusions (3)

Highly Incremental Development

may be:

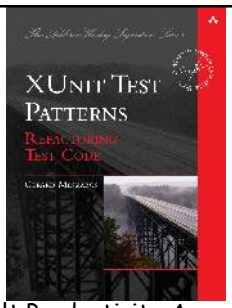
- **Evolved using Inspect & Adapt**
- **Designed from scratch based on deep understanding**
- **Adopted & Evolved (e.g. Scrum + XP)**

Thank You!

Gerard Meszaros

India2011@gerardm.com

http://www.xunitpatterns.com



http://KeepingSoftwareSoft.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**