Code Quality

And Modern Technical Practices

Code Quality

- What is code quality
- Code quality Metrics
- How to improve code quality
  - Practices
  - Design Principles
  - Tools

Code Quality

Quality code is

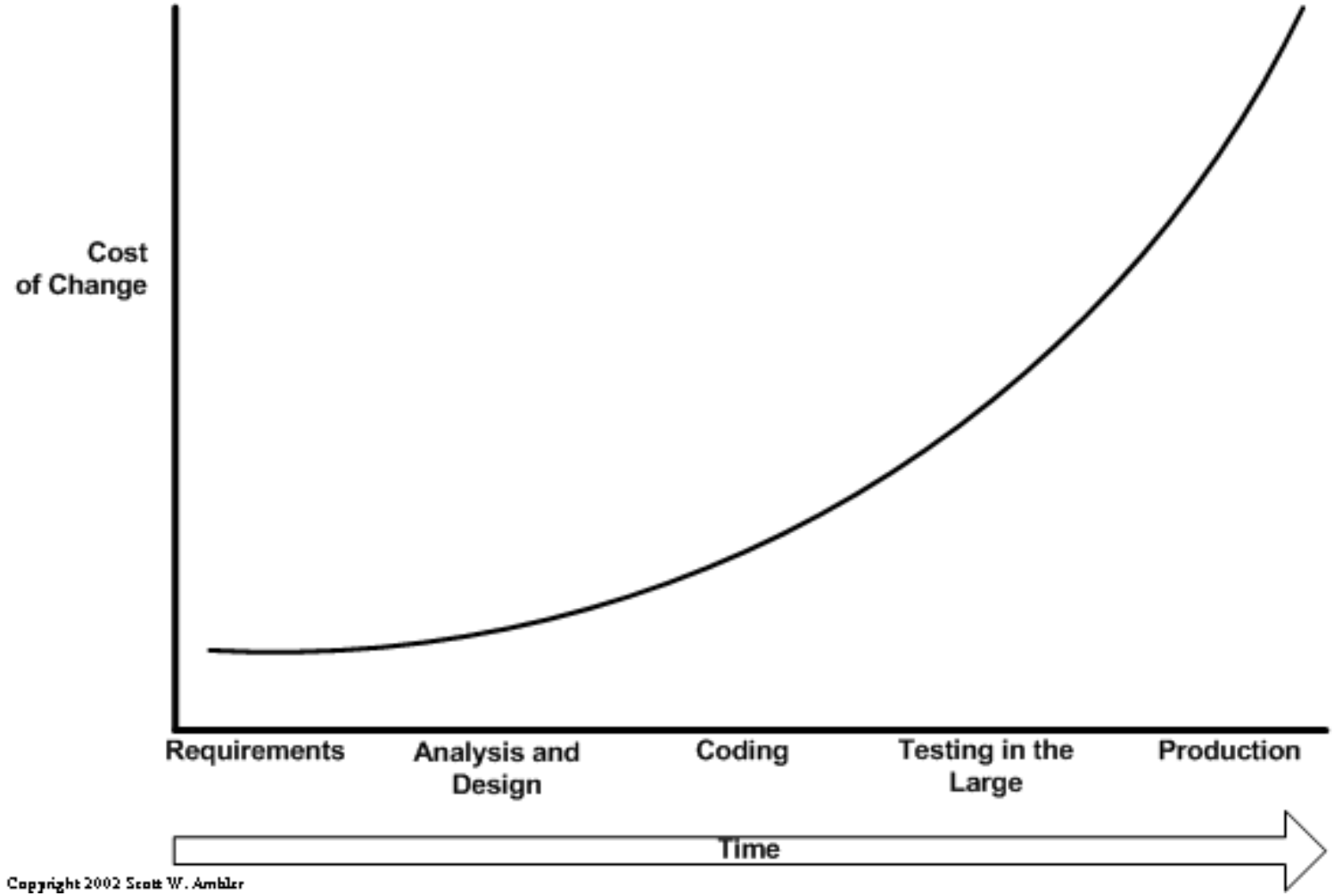
- Easy to understand
- Simple to maintain
- Easy to extend
- Reuse in the future
- Code must have unit test

Why we need to bother for code quality ?

Cost of change

One of the universal assumption of software engineering is that cost of changing a program rises exponentially over time .

Cost of change



Reduce Productivity

Code Quality Metrics

- Lines of code
- Depth of Inheritance Tree
- Coupling Between Object Classes
- Lack of Cohesion in Methods
- Maintainability Index

Lines Of Code

- Function
- Classes

Function

- Function should be small
- Functions should not have too many arguments
- Avoid flag arguments
- Command query separation
- Should follow SRP and OCP

Classes

- A class is a collection of data and routines that share a cohesive, well-defined responsibility.

- A class might also be a collection of routines that provides a cohesive set of services even if no common data is involved.

Classes

- Classes should be small

  So How many lines ?

- Instead of lines we count responsibilities .
- Classes should have one responsibility – one reason to change

Depth of inheritance tree

- Deeply nested type hierarchies can be difficult to follow, understand, and maintain

- Types where Depth Of Inheritance is higher or equal than 6 might be hard to maintain.

- However it is not a rule since sometime your classes might inherit from third-party classes which have a high value for depth of inheritance.

Coupling

- Coupling among classes or subsystems is a measure of how interconnected those classes or subsystems are.
- Tight coupling means that related classes have to know internal details of each other
- Changes ripple through the system, and the system is potentially harder to understand.
- It is a good practice to have types and methods that exhibit low coupling and high cohesion
- Types and methods that have a high degree of class coupling can be difficult to maintain.

Removing Coupling

- Factories
- Dependency Injection

Cohesion

Cyclomatic complexity

- The number of decisions made in our source code

- We use cyclomatic complexity to get a sense of how hard any given code may be to test, maintain, or troubleshoot as well as an indication of how likely the code will be to produce errors.

Maintainability Index

- **Maintainability Index** –An index value between 0 and 100 that represents the relative ease of maintaining the code. A high value means better maintainability.

- 0-9 = Red
- 10-19 = Yellow
- 20-100 = Green

Maintainability Index = MAX(0,(171 - 5.2 * ln(Halstead Volume) - 0.23 * (Cyclomatic Complexity) - 16.2 * ln(Lines of Code))*100 / 171)

Halstead Program Volume

- Halstead Program Length N = $N_1 + N_2$
- Halstead Program Volume V = $N * \log_2(n_1 + n_2)$

N_1	The total number of *operators* present.
N_2	The total number of *operands* present.
n_1	The number of distinct *operators* present.
n_2	The number of distinct *operands* present.

Duplicate Code

- Identical code at different place – Extract method
- Switch/case or if/else chain that appears again and again in various modules , always testing for the same set of conditions - Replaced with polymorphism
- Same algorithm , but that don’t share similar lines of code – Template method or Strategy pattern
- Same code in two sibling classes

Improving Code Quality

Practices

- Coding Standard
- Code Review
- Unit testing
- Agile
- Pair Programming
- Extreme Programming
- BDD
- Test Driven Development
- Refactoring
- Mentorship
- Continuous Integration
- Acceptance test driven development
- Communication & Collaboration
- Collective Ownership
- Small cycle feedback

Design Principals

- GOF Design Patterns
- SOLID Design Principal
- GRASP

Tools

- Visual Studio 2012
- Stylecop
- FxCop
- Resharper
- Ncover
- Ndepend
- Sonar
- Code Coverage

General Responsibility Assignment Software Patterns or Principles (GRASP)

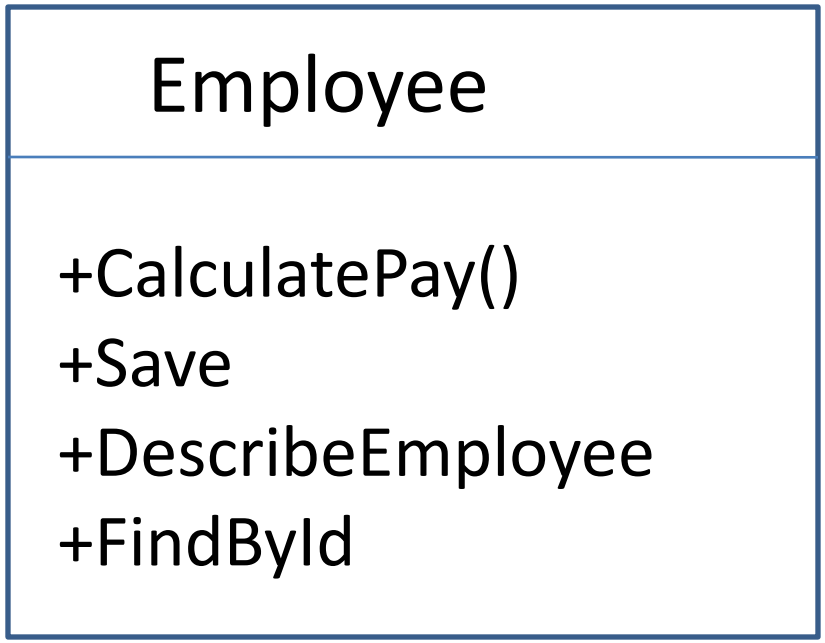
Pattern/ Principle	Description
Information Expert	A general principle of object design and responsibility assignment? Assign a responsibility to the information expert—the class that has the information necessary to fulfill the responsibility.
Creator	Who creates? (Note that Factory is a common alternate solution.) Assign class B the responsibility to create an instance of class A if one of these is true: 1. B contains A 2. B aggregates A 3. B has the initializing data for A 4. B records A 5. B closely uses A
Controller	What first object beyond the UI layer receives and coordinates ("controls") a system operation? Assign the responsibility to an object representing one of these choices: 1. Represents the overall "system," a "root object," a device that the software is running within, or a major subsystem (these are all variations of a *facade controller*). 2. Represents a use case scenario within which the system operation occurs (a use-case or *session controller*)
Low Coupling (evaluative)	How to reduce the impact of change? Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.
High Cohesion (evaluative)	How to keep objects focused, understandable, and manageable, and as a side-effect, support Low Coupling? Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives.
Polymorphism	Who is responsible when behavior varies by type? When related alternatives or behaviors vary by type (class), assign responsibility for the behavior—using polymorphic operations—to the types for which the behavior varies.
Pure Fabrication	Who is responsible when you are desperate, and do not want to violate high cohesion and low coupling? Assign a highly cohesive set of responsibilities to an artificial or convenience "behavior" class that does not represent a problem domain concept—something made up, in order to support high cohesion, low coupling, and reuse.
Indirection	How to assign responsibilities to avoid direct coupling? Assign the responsibility to an intermediate object to mediate between other components or services, so that they are not directly coupled.
Protected Variations	How to assign responsibilities to objects, subsystems, and systems so that the variations or instability in these elements do not have an undesirable impact on other elements? Identify points of predicted variation or instability; assign responsibilities to create a stable "interface" around them.

SOLID

- Single responsibility principle
- Open/closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

Single responsibility principle

Class should have only one reason to change


Employee
+CalculatePay()
+Save
+DescribeEmployee
+FindById

Open/closed principle

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

```
public void DrawAllShapes(IList shapes)
 {
foreach(Shape shape in shapes)
        shape.Draw();
}
```

Liskov substitution principle

Subtypes must be substitutable for their base types

LSP Example

```
public class Rectangle
{
  private double width;
  private double height;

  public virtual double Width
  {
    get { return width; }
    set { width = value; }
  }

  public virtual double Height
  {
    get { return height; }
    set { height = value; }
  }
}
```

```
public class Square : Rectangle
{
  public override double Width
  {
    set
    {
      base.Width = value;
      base.Height = value;
    }
  }
  public override double Height
  {
    set
    {
      base.Height = value;
      base.Width = value;
    }
  }
```

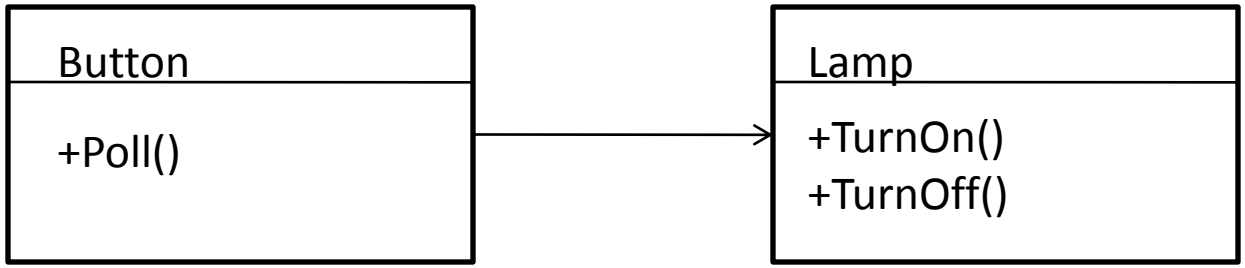
```
void g(Rectangle r)
      {
        r.Width = 5;
        r.Height = 4;
        if(r.Area() != 20)
          throw new Exception("Bad area!");
      }
```

Dependency inversion principle

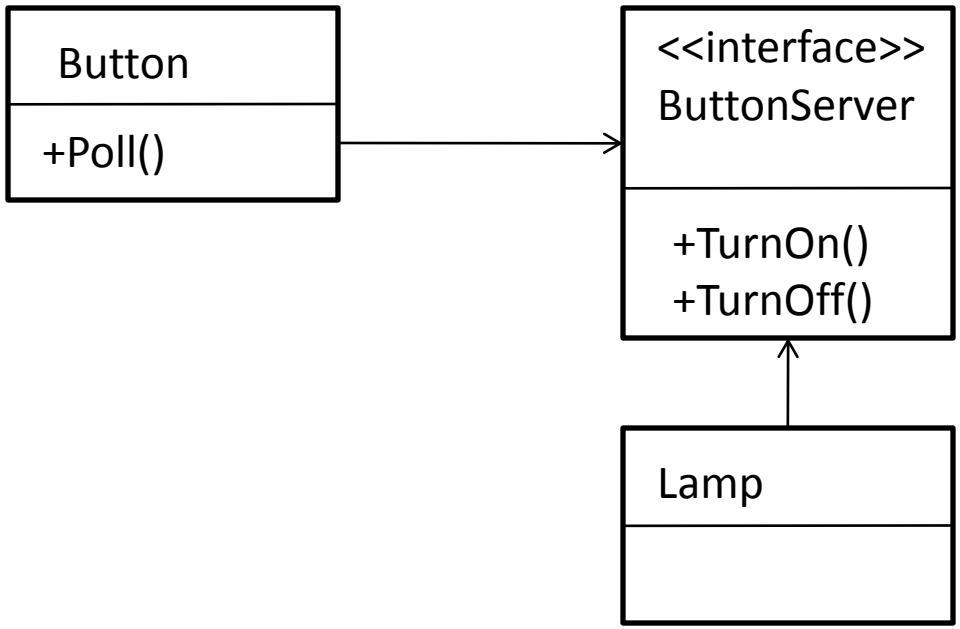
- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions.

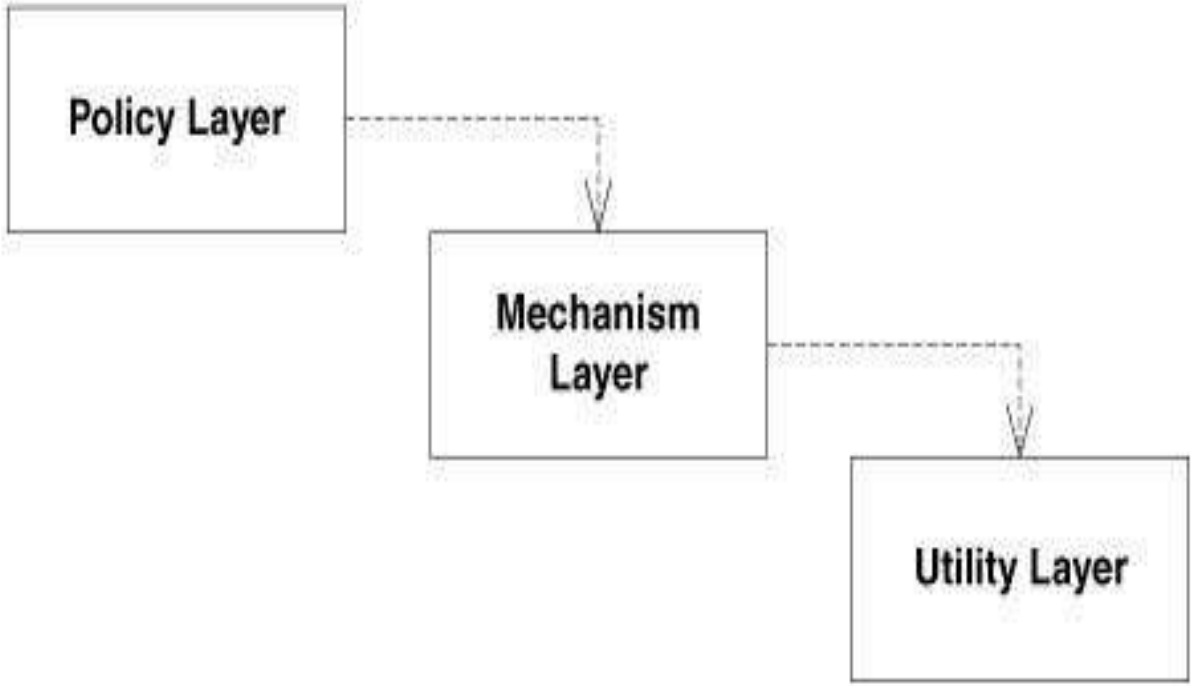
DIP Example

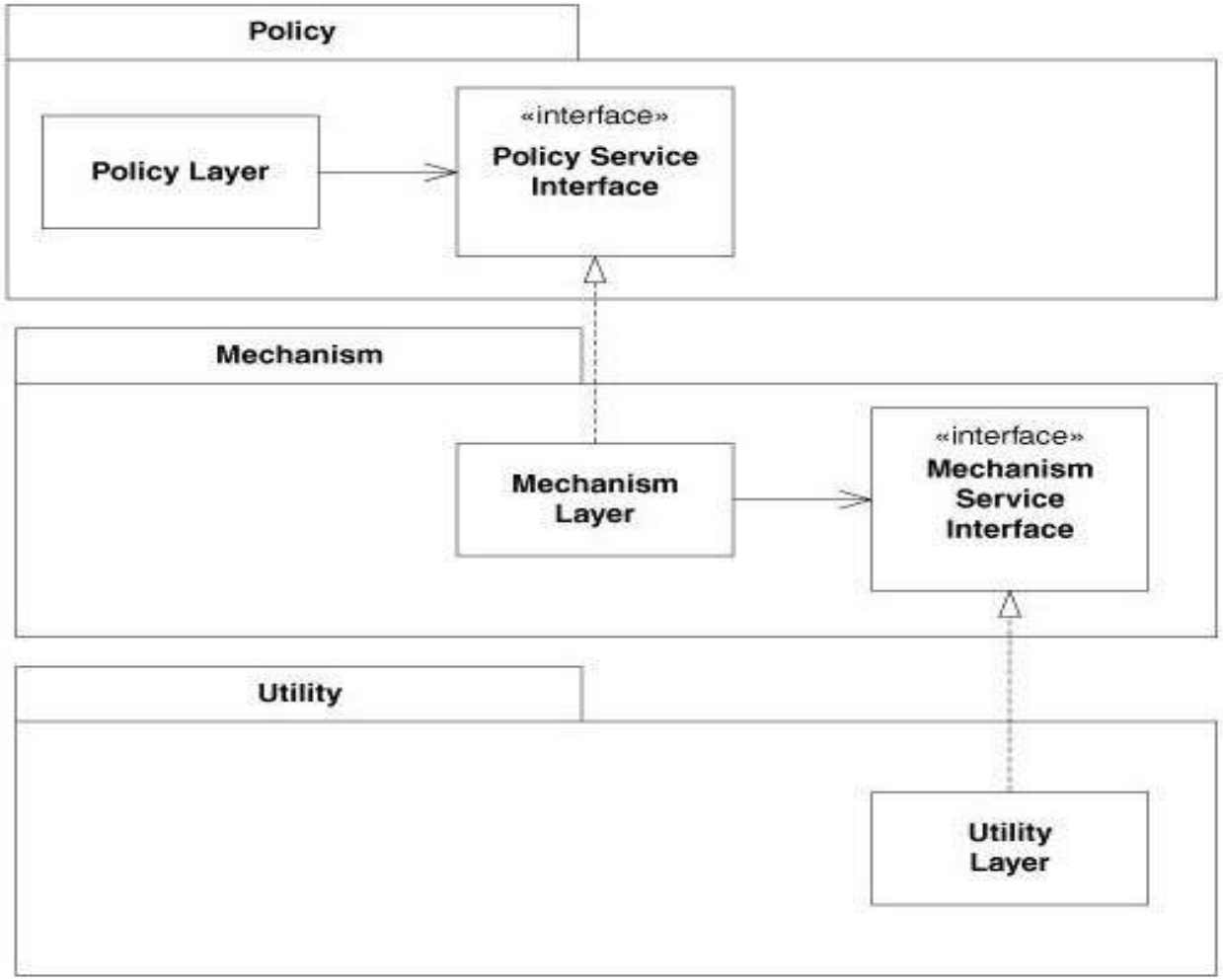
- Consider a **Button** and **Lamp** object



```
Public class Button
{
    Private Lamp lamp;
    Public void poll()
    {
        If(/*some condition */)
            lamp.TurnOn();
    }
}
```


Button
+Poll()
<<interface>>
ButtonServer
+TurnOn()
+TurnOff()
Lamp


Policy Layer
Mechanism Layer
Utility Layer


Policy
Policy Layer
«interface»
Policy Service Interface
Mechanism
Mechanism Layer
«interface»
Mechanism Service Interface
Utility
Utility Layer

Interface segregation principle

Clients should not be forced to depend upon interfaces that they do not use

Coding standards

Programmers write all code in accordance with rules.

Code Review

Problems are found later, when they are more expensive to fix

It is well documented that the later a bug is found the more expensive it is to fix. Before submitting something for a code review, a conscientious developer will have designed a solution, coded it, written unit and integration tests for it, and probably done some manual acceptance testing as well, all in preparation for checking in. If a problem is found in a subsequent code review, that whole process has to start again, at least to some degree.

By contrast, pair programming finds those problems earlier on in the life cycle, thereby requiring less rework to fix. Pair programming is defect prevention. You find problems earlier, when they are cheaper to fix.

Context switching is required, which lowers productivity

Code reviews don't happen instantly. They take time. Other developers are busy. What are you going to do while you are waiting for the review to complete? Hopefully you are not going to sit around doing nothing - you are going to start on the next story. Then the next day (or after several days, in some situations I have been in) you are going to receive feedback from the review. In general you can respond in one of two ways: you can defend the code the way it is, or you can agree to make the suggested change. Either way, you need to get back to the state the code was in at the time of the review. But you have since moved on; you have made other changes to the code.

Sure, there are strategies for handling this. You can create a patch file for your current changes and revert them. You can create and work in a separate branch for each story. But there is overhead in these approaches. In addition to the context switching of the actual code, there is the mental context switching in your head. You have moved on to other problems, but now have to get your brain wrapped around the previous issues again. I suspect that of the two types of context switching, this one (the mental one) has the greater cost.

With pair programming, on the other hand, you are finding and addressing problems right while you are in the midst of them - no context switching required.

Quality is lower, because of resistance to changing working, tested code

When asked what he thought of code reviews, I heard one developer say "It just feels like it's too late". In talking further, what he was getting at was that by the time the code review happens, all this work of coding and testing has already been done. To suggest that we tear it all up and rework it doesn't seem all that productive at that point. Sure, if something is really broken it will have to be addressed, but otherwise, people tend to want to leave it as is.

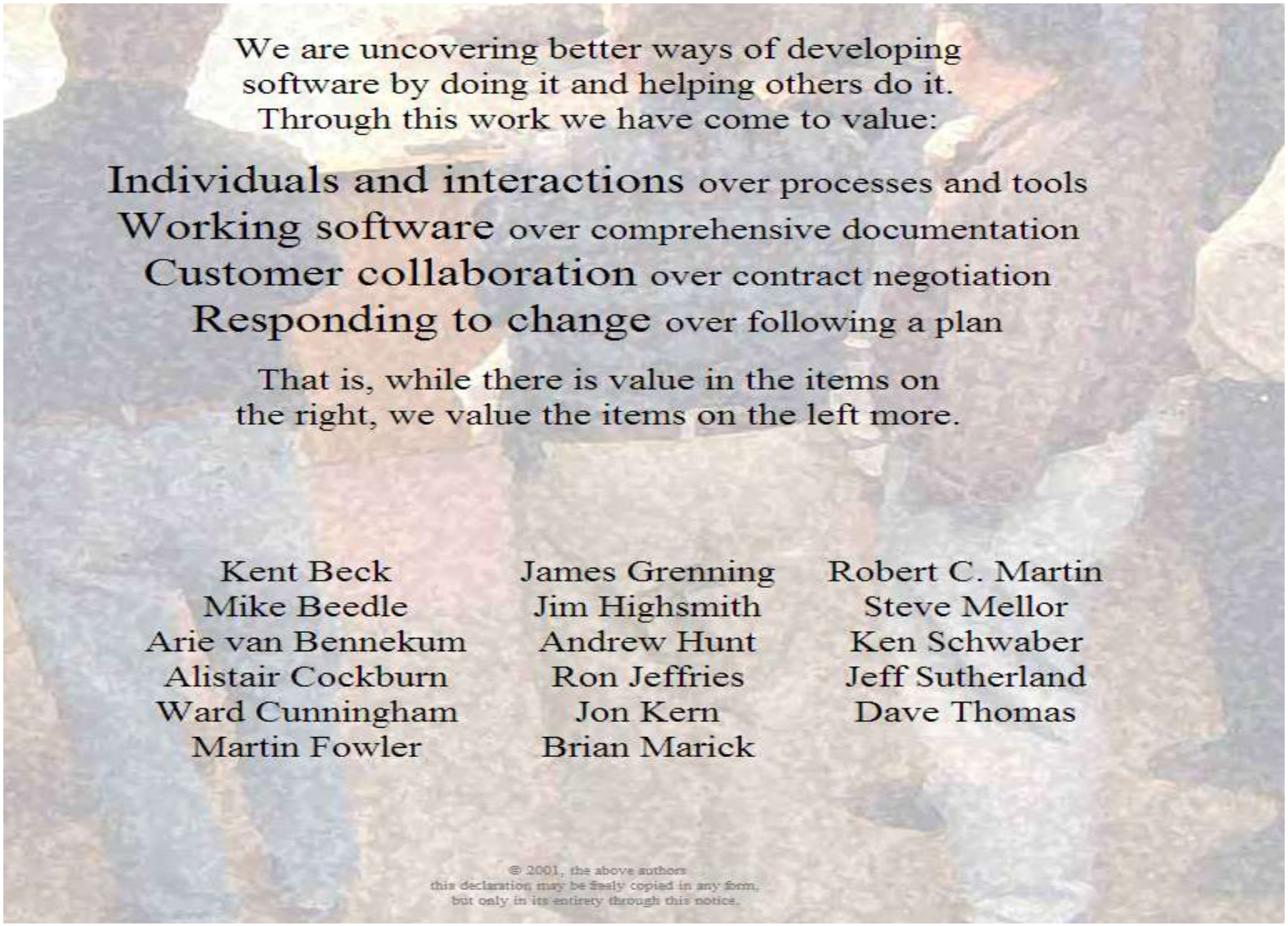
Another aspect of this is that even if some changes are made to fix a problem, they often tend to be band-aid solutions, because you are now retrofitting existing code; the end result won't be of the same quality as a pair working together from the beginning. It's like the difference between building a house, then having an inspection, and then fixing the problems the inspector found, versus building a house with an inspector alongside you the entire time. No band-aids placed over things, just quality code from the ground up.

Bad feelings can arise between team members

This one is less common than the others, but I have seen it happen. While we would like to think that we are all objective-minded and don't become emotionally invested in our code, that isn't always the case. And after designing, coding, and testing it we are more invested in it than if it was still just in the idea phase. In a code review, it is our working, tested, finished product that is being critiqued. When pairing, there is still a lot of critiquing going on (probably even more than in a code review), but it is happening earlier in the process; usually in the idea phase when we are less invested in a specific solution. Overall, code reviews can feel more adversarial than pair programming (which feels very collaborative).

It's all about collaboration - don't just write something and throw it over the wall for a code review. Pair program!

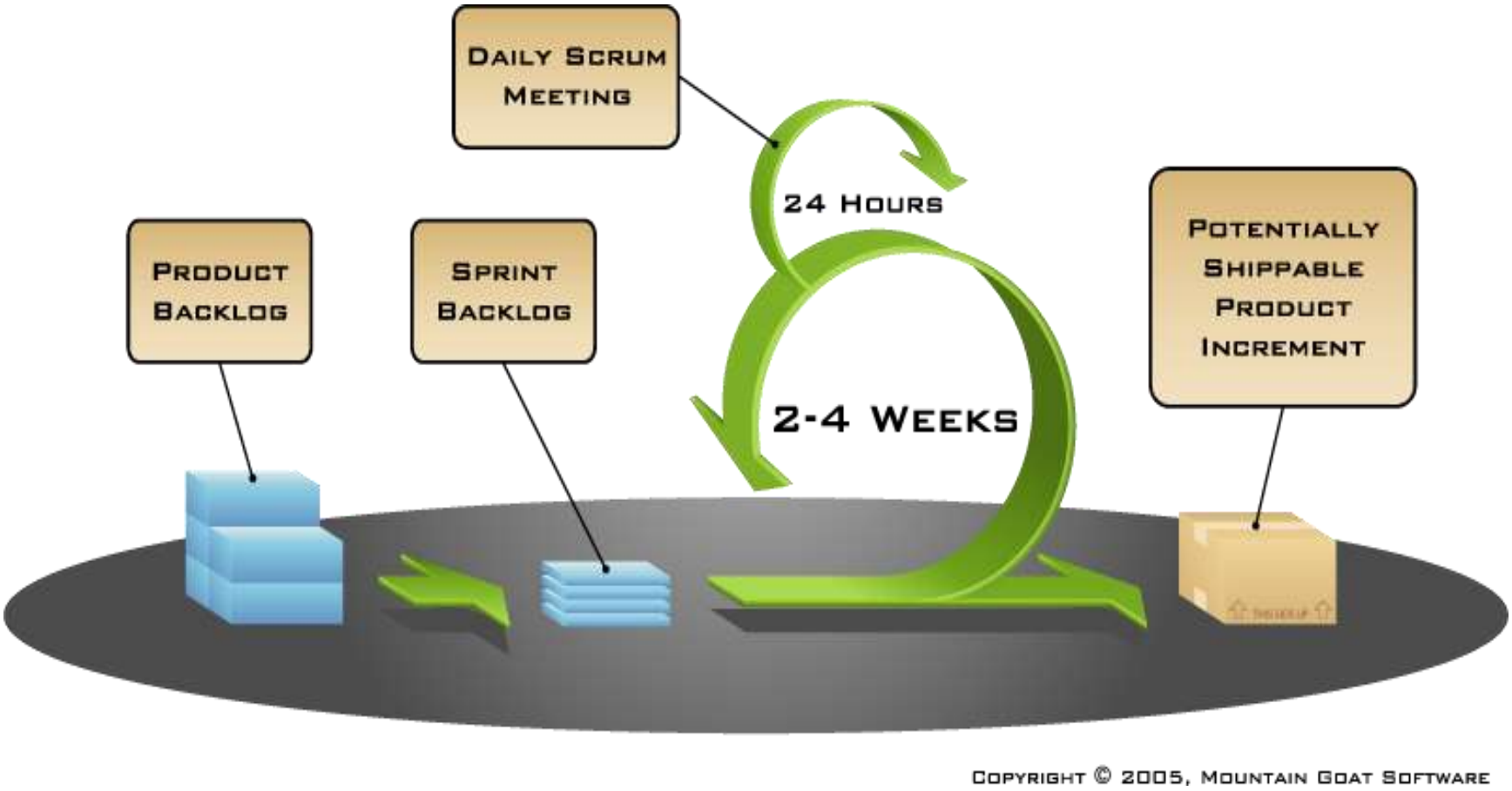
Agile


We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan
That is, while there is value in the items on
the right, we value the items on the left more.
Kent Beck
Mike Beedle
Arie van Bennekum
Alistair Cockburn
Ward Cunningham
Martin Fowler
James Grenning
Jim Highsmith
Andrew Hunt
Ron Jeffries
Jon Kern
Brian Marick
Robert C. Martin
Steve Mellor
Ken Schwaber
Jeff Sutherland
Dave Thomas
© 2001, the above authors
this declaration may be freely copied in any form,
but only in its entirety through this notice.

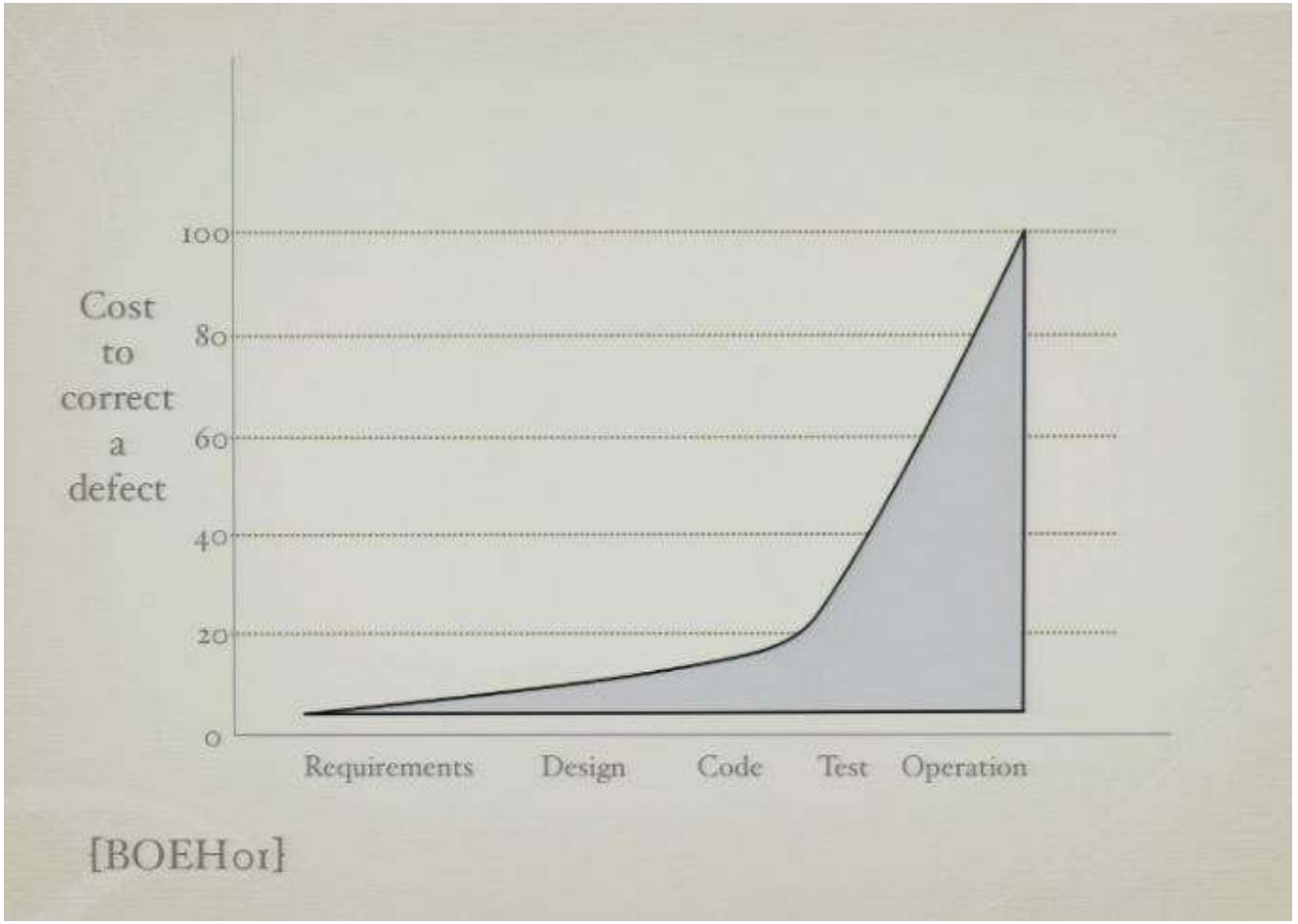
Scrum

- Scrum is an agile process that allows us to focus on delivering the highest business value in the shortest time.
- It allows us to rapidly and repeatedly inspect actual working software (every two weeks to one month).
- The business sets the priorities. Teams self-organize to determine the best way to deliver the highest priority features.
- Every two weeks to a month anyone can see real working software and decide to release it as is or continue to enhance it for another sprint

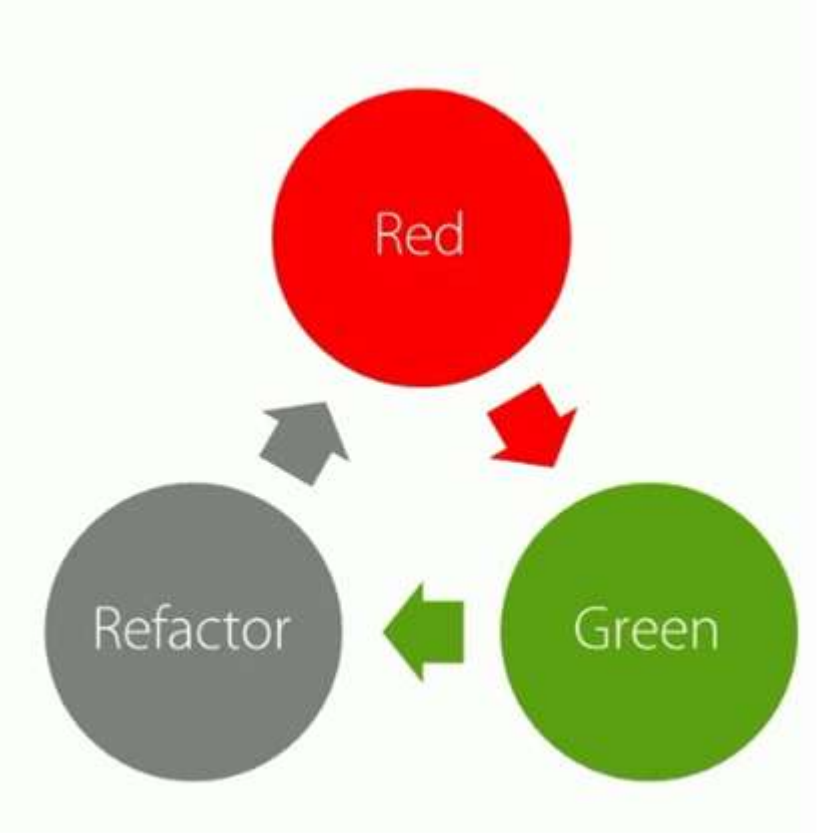
Scrum


Daily Scrum Meeting
24 Hours
Product Backlog
Sprint Backlog
2-4 Weeks
Potentially Shippable Product Increment
Copyright © 2005, Mountain Goat Software

Cost Of Defect



Test Driven Development

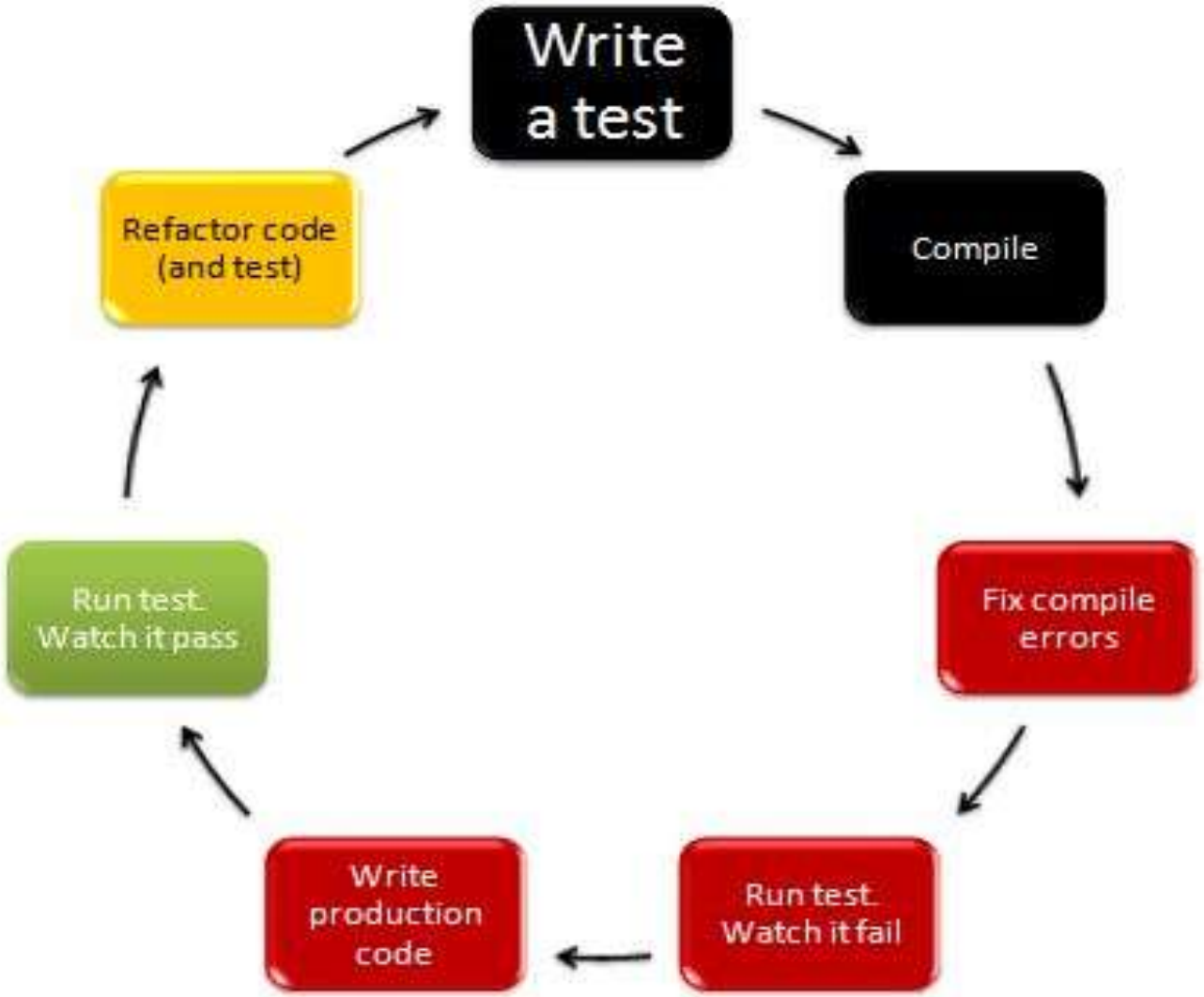


Create a failing test – Red

Make the test pass – Green

Refactor – Improve the internal implementation without changing the external contract or behavior

Test Driven Process

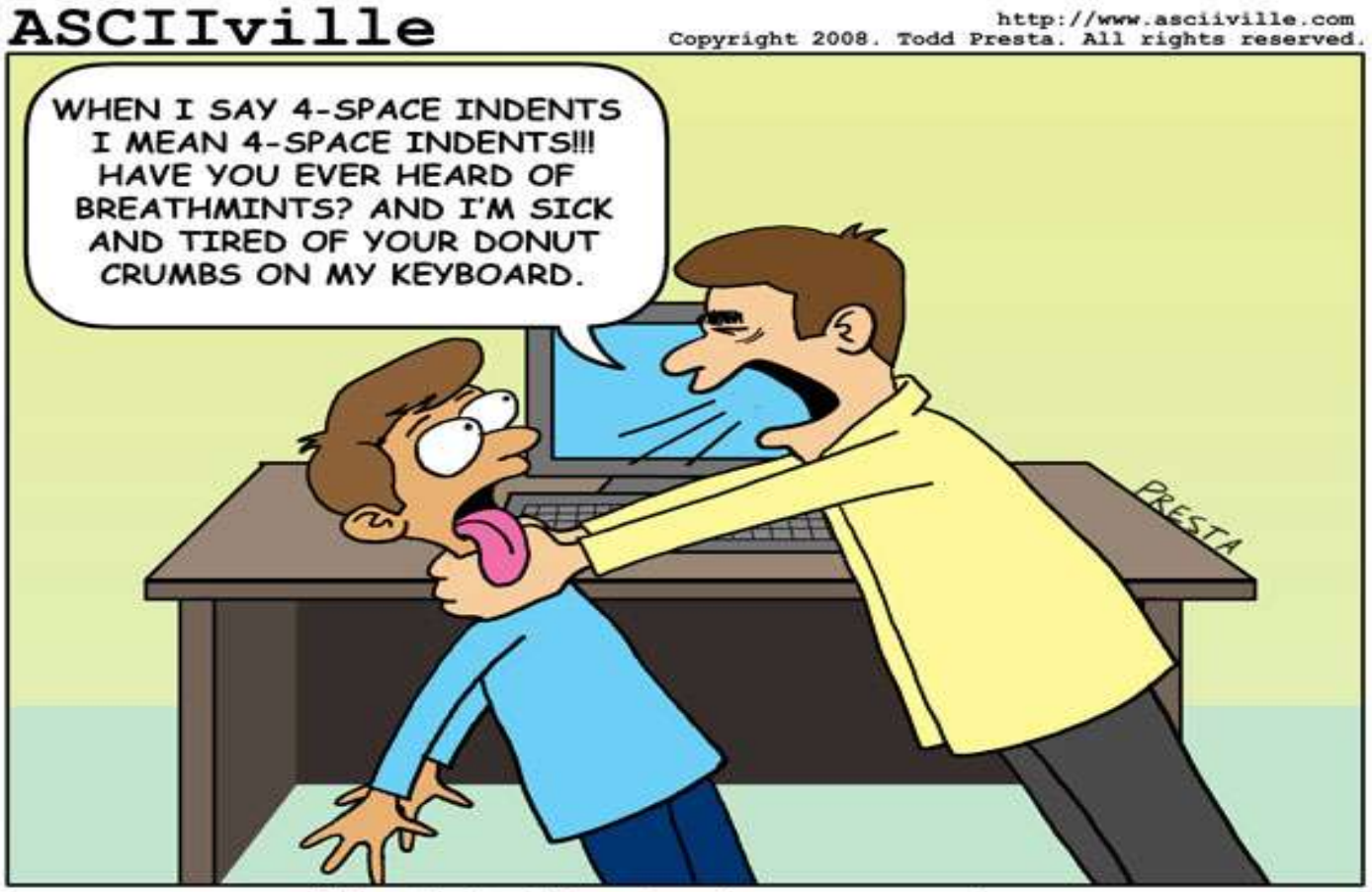


Pair Programming

All production code is written with two programmers at one machine.

If people program solo they are more likely to make mistakes, more likely to overdesign, and more likely to blow off the other practices, particularly under pressure.


You missed a semicolumn
Spelling mistakes
Don't try to be a compiler
A



The dark side of pair programming.

Economics of pair programming

Pair programming will double code development expenses

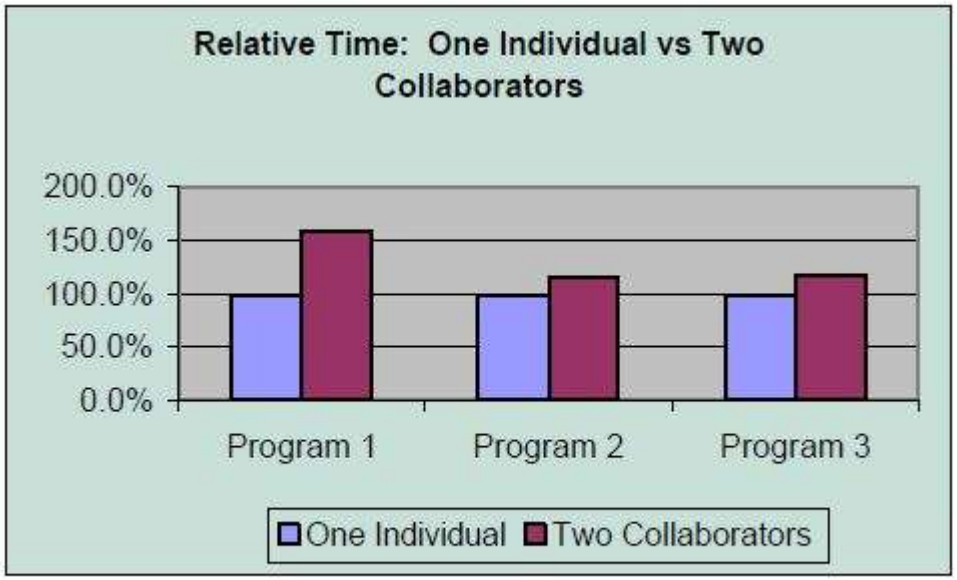


Figure 1: Programmer Time

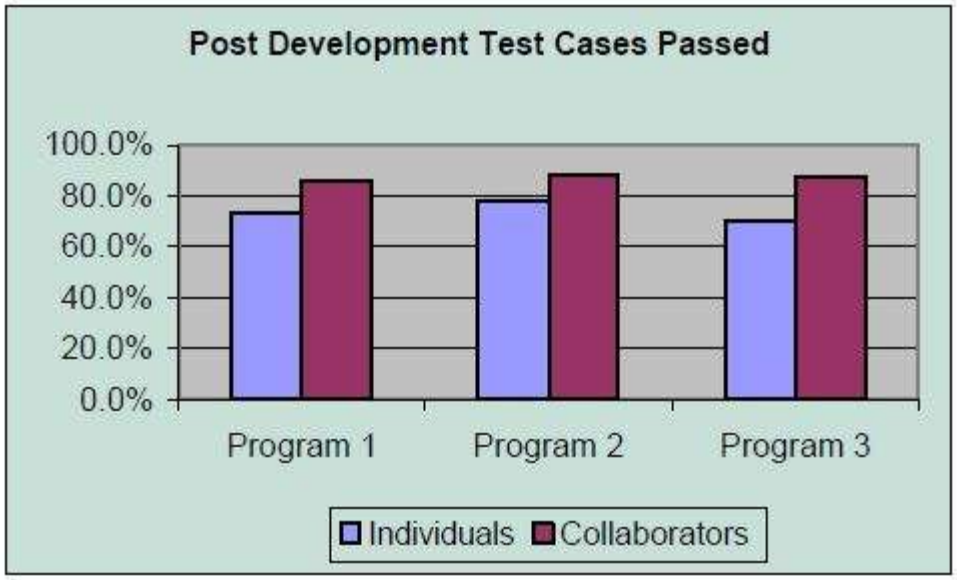


Figure 2: Code Defects

Extreme Programming

Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, courage, and respect. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation.

- Core Practices
  - Whole Team
  - Pair Programming
  - Incremental Design
  - Continuous Integration
  - Test-First Programming
  - Weekly Cycle
  - Ten-Minute Build

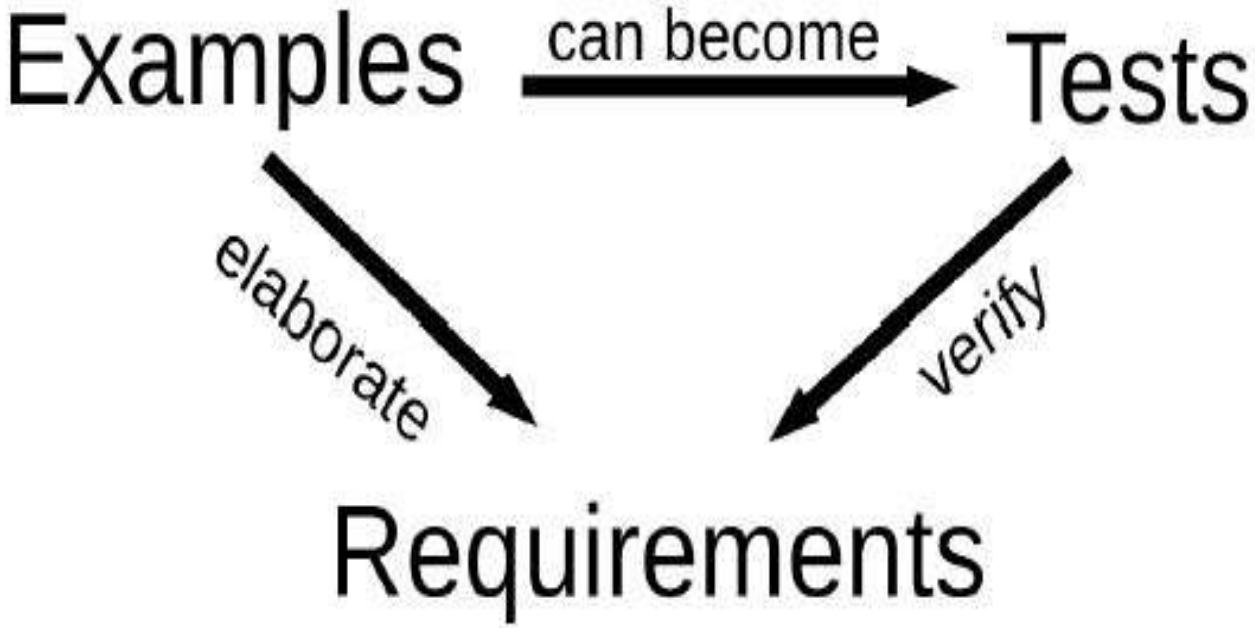
ATDD

Drive implementation of a requirement through a set of automated , executable acceptance tests

About

- Communication
- Collaboration
- Process improvements
- Better Specification

Acceptance test driven development



Code Coverage

Refactoring

Programmers restructure the system without changing its behaviour to remove duplication, improve communication, simplify, or add flexibility.

Refactoring

- Improves the design of Software
- Makes software easier to understand
- Helps you find bugs
- Helps you program faster

When we should refactor

All the time

Tools

- Visual Studio 2012
- Stylecop
- FxCop
- Resharper
- Ncover
- Ndepend
- Sonar

Visual Studio 2012

- **Maintainability Index**
- **Cyclomatic Complexity**
- **Depth of Inheritance**
- **Class Coupling**
- **Lines of Code**
- **Duplicate Code**
- **Code Coverage**

Exercise

Revisit your previous coding efforts, try to find ways you can improve the old code.

Reference

- http://www.wikispeed.com/Affordable