

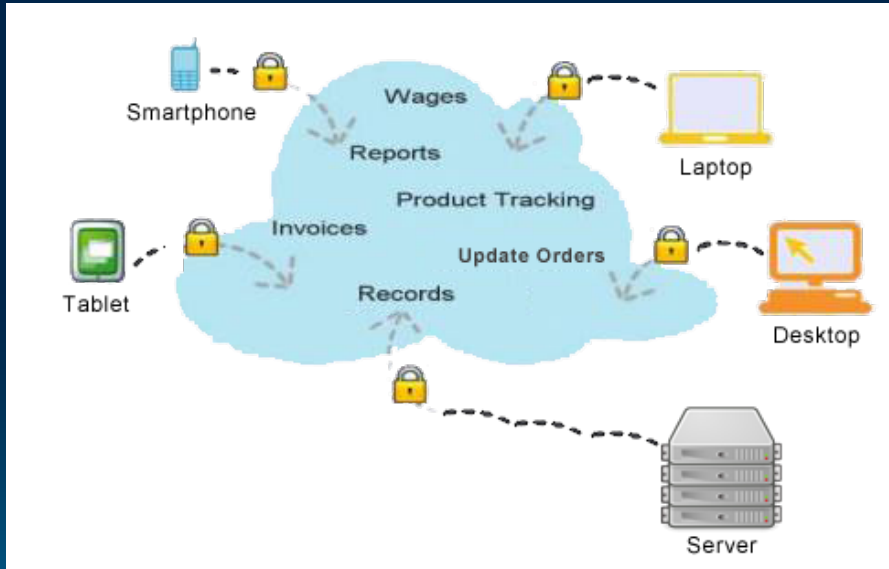
Taming Big Balls of Mud with Agile, Diligence and Lot's of Hard Work



SATURN— April 30th, 2015

***Joseph W. Yoder -- www.refactory.com
www.teamsthatinnovate.com***

Sustainable Architecture



Joseph W. Yoder -- www.refactory.com

Evolved from UIUC SAG

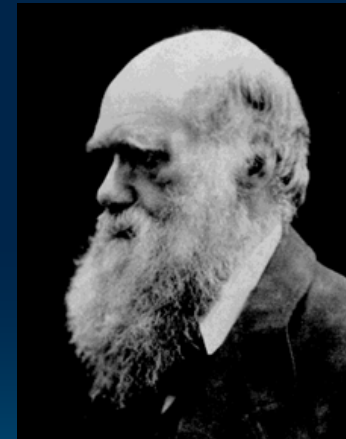
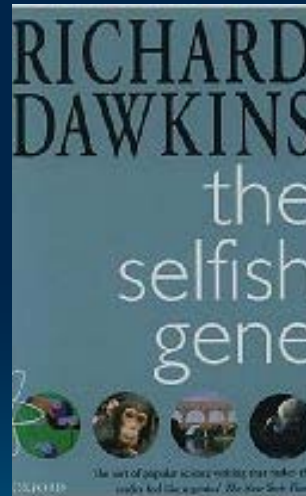
In the early 90's we were studying objects, frameworks, components, meta, refactoring, reusability, patterns, “good” architecture.



However, in our SAG group we often noticed that although we talk a good game, many successful systems do not have a good internal structure at all.

Selfish Class

Brian and I had just published a paper called Selfish Class which takes a *code's-eye view of software reuse and evolution*.



In contrast, our BBoM paper noted that in reality, a lot of code was hard to (re)-use.

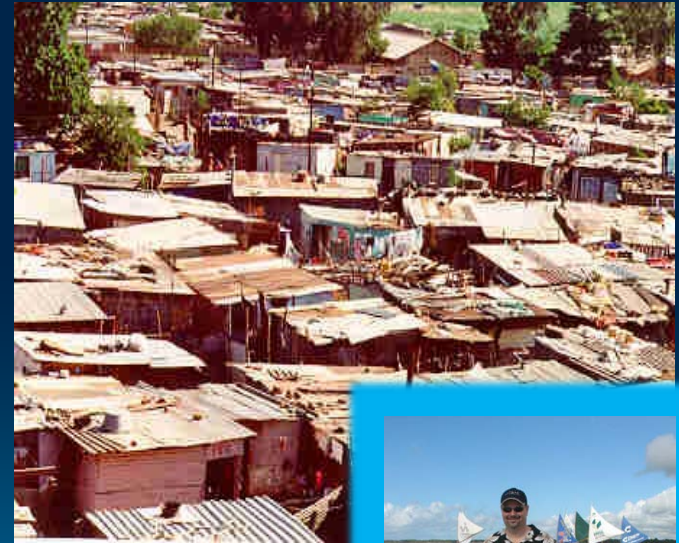
Big Ball of Mud

Alias: Shantytown, Spaghetti Code

A BIG BALL OF MUD is haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle.

The de-facto standard software architecture. Why is the gap between what we **preach** and what we **practice** so large?

We preach we want to build high quality systems but why are BBoMs so prevalent?



Worse is Better

Idea resembles Gabriel's 1991
"Worse is Better"

Worse is Better is an argument to release early and then have the market help you design the final product. It is taken as the first published argument for open source, among other things.

Do BBoM systems have a Quality?

What exactly do we mean by "Big"?

Well, for teams I consider $> 10^2$ big
and for code I consider $> 10^5$ big



```
; (a)
movf  PSP,w           ; Copy current top of stack frame address
movwf FSR             ; into the File Select register

; (b)
movf  MULTIPLICAND,w  ; Push the Multiplicand into the stack
movwf INDF             ; by copying the datum out
decf  FSR,f           ; and decrementing the FSR

; (c)
movf  MULTIPLIER,w    ; Push the Multiplier into the stack
movwf INDF             ; by copying the datum out
decf  FSR,f           ; and decrementing the FSR

; (d)
call  MUL_S           ; Call the subroutine
```


Legacy == Mud?



Legacy != Mud???

Does Legacy happen within months or a year after the first release?

Or is legacy after the second release?

What about Muddy code that is released on the first version? Is this a counterexample?

Is all Legacy Mud?

Messy Code

Have you been significantly impeded by messy code?



With permission from: Mrs D. <http://www.belly-timber.com/2005/10/03/embracing-our-inner-web-stat/>

Designs Naturally Degrade

- All designs evolve.
- Code rots as changes are made unless it is kept clean.
- Entropy increases over time.

```
class Pair
{
private:
    Object first_;
    Object second_;
    Object third_;

public:
    Pair() { }
    Pair( Object first, Object second, Object third )
        :first_(first), second_(second), third_(third)
    {}

    // etc.

};
```

Neglect Is Contagious

- Disorder increases and software rots over time.
- Don't tolerate a broken window.



http://www.pragmaticprogrammer.com/ppbook/extracts/no_broken_windows.html

Where Mud Comes From?



DILBERT © United Feature Syndicate, Inc.
Redistribution in whole or in part prohibited.

People Write Code → People make Mud

Keep it Working, Piecemeal Growth, Throwaway Code



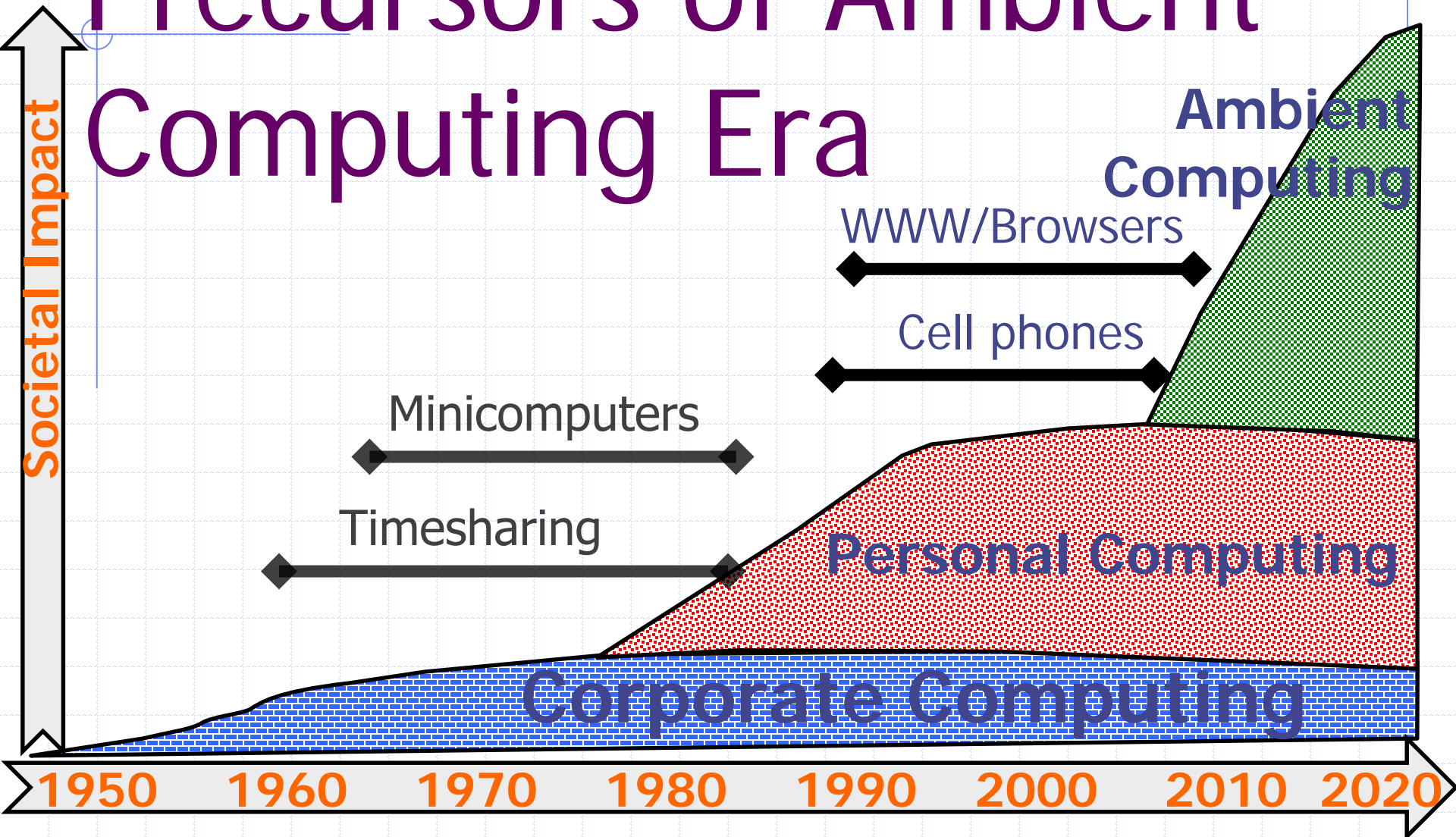
Sustaining Your Architecture

Copy 'n' Paste



Sustaining Your Architecture

Transitional Technologies Precursors of Ambient Computing Era



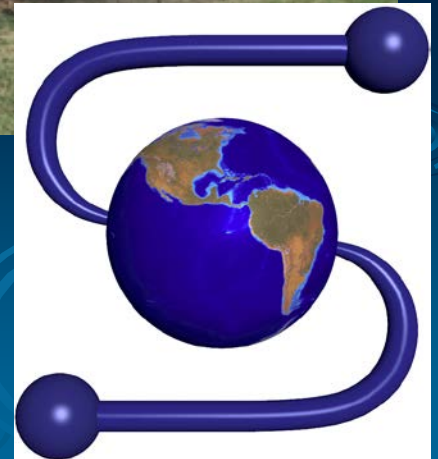
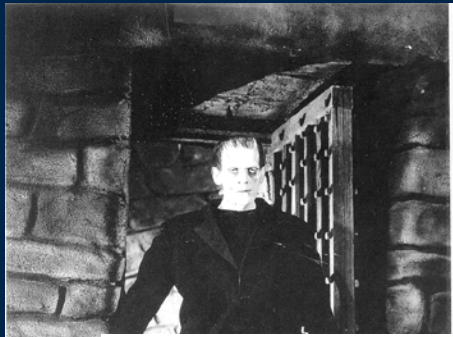
Apps on Browser Platforms



Apps are becoming the norm running on whatever platform



The Age of Sampling & Big Bucket of Glue



Sustaining Your Architecture

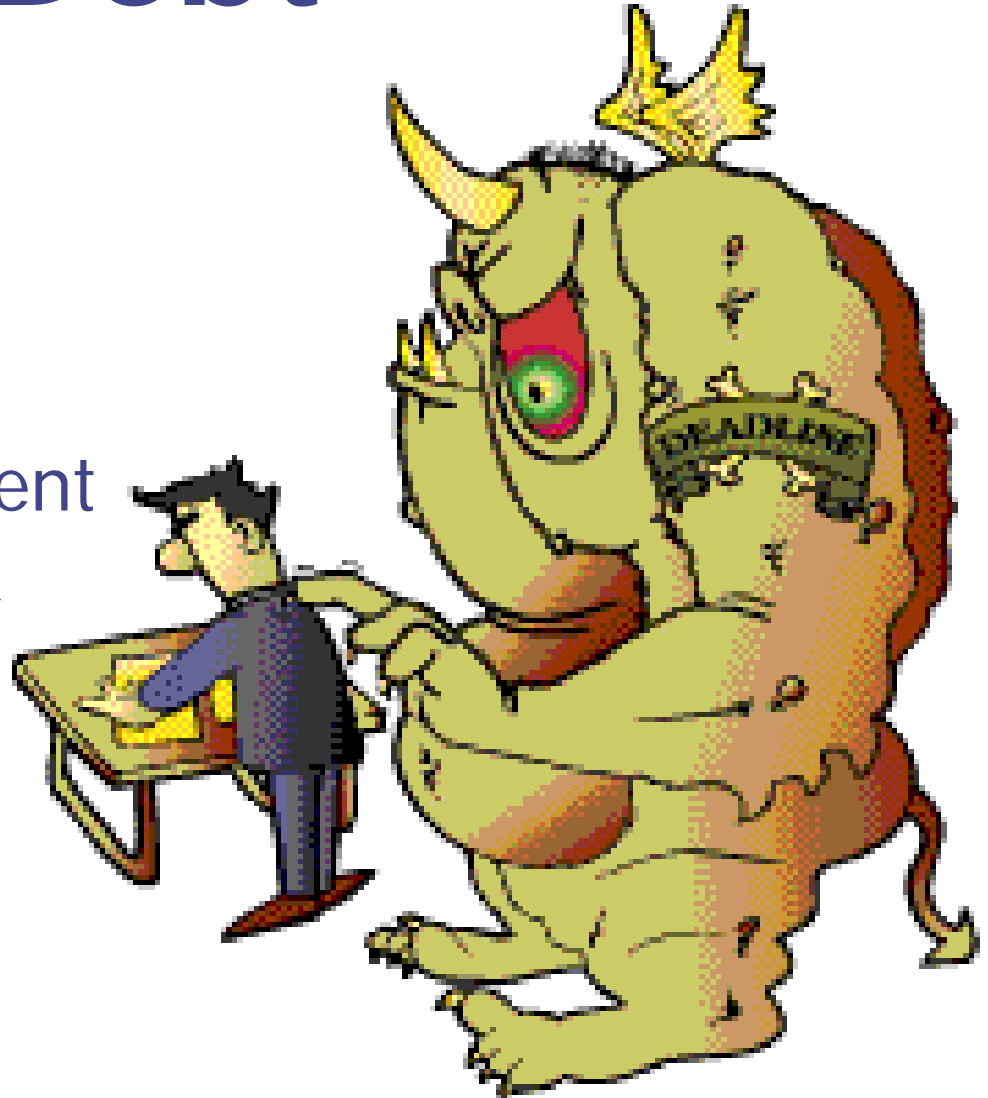
They Have a Name



Sustaining Your Architecture

Technical Debt

- ◆ Term invented by Ward Cunningham
- ◆ Piles up when you continually implement without going back to reflect new understanding
- ◆ Can have long term costs and consequences



Is Mud Normal?

Well, just read our paper....there are "normal" reasons why it happens. Maybe it is the best we can do right now.

If mud is such a bad thing, why do people keep making it?

Maybe if we accept it and teach it more then we can deal with it better and help prevent it from getting too bad.

Agile to the Rescue???

- 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.



...From the Agile Manifesto

Can Agile Help?

Scrum, TDD, **Refactoring**, Regular
Feedback, **Testing**, More Eyes,

Good People!!!

Continuous **attention** to **technical
excellence!**

Retrospectives!

Face-To-Face conversation.

Motivated individuals with the
environment and **support** they need.

Agile Design Values

➤ Core values:

- Design Simplicity
- Communication
- **Continuous Improvement**
- **Teamwork / Trust**
- **Satisfying stakeholder needs**

➤ **Keep learning**

➤ **Continuous Feedback**

➤ **Lots of Testing/Validation!!!**



Some Agile Myths

- Simple solutions are always best.
- We can easily adapt to changing requirements (new requirements).
- Scrum/TDD will ensure good Design/Architecture.
- Good architecture simply emerges from “good” development practices. Sometimes you need more.
- Make significant architecture changes at the last moment.



www.agilemyths.com

Do Some Agile Principles Encourage bad design?

Lack of Upfront Design?

Late changes to the requirements
of the system?

Continuously Evolving the Architecture?

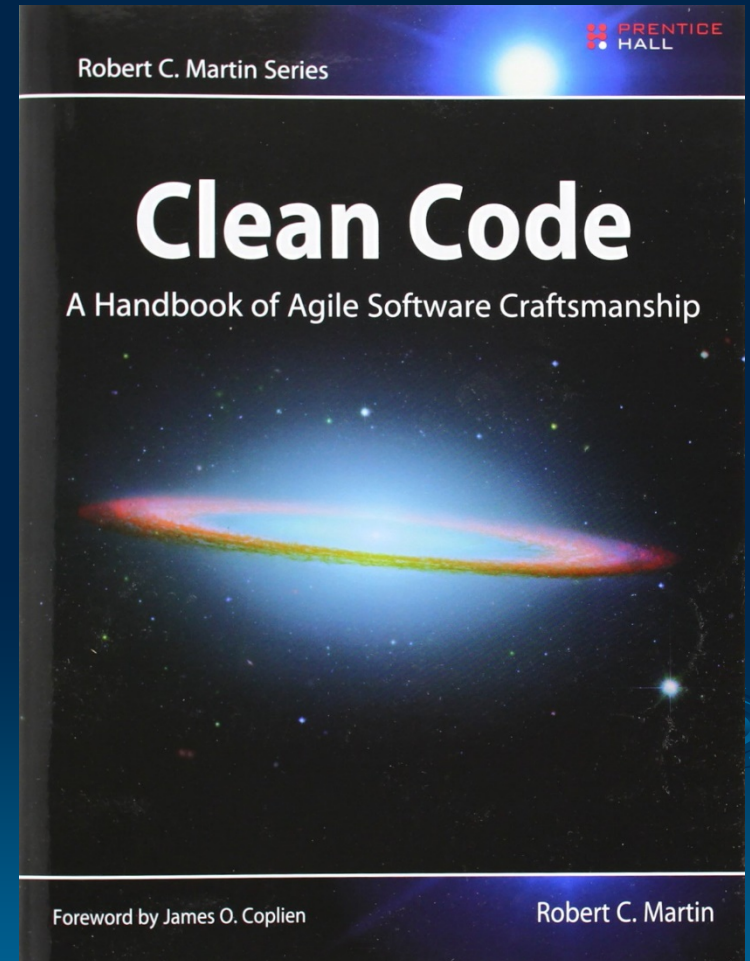
Piecemeal Growth?

Focus on Process rather than Architecture?

Working code is the measure of success!

I'm sure there are more!!!

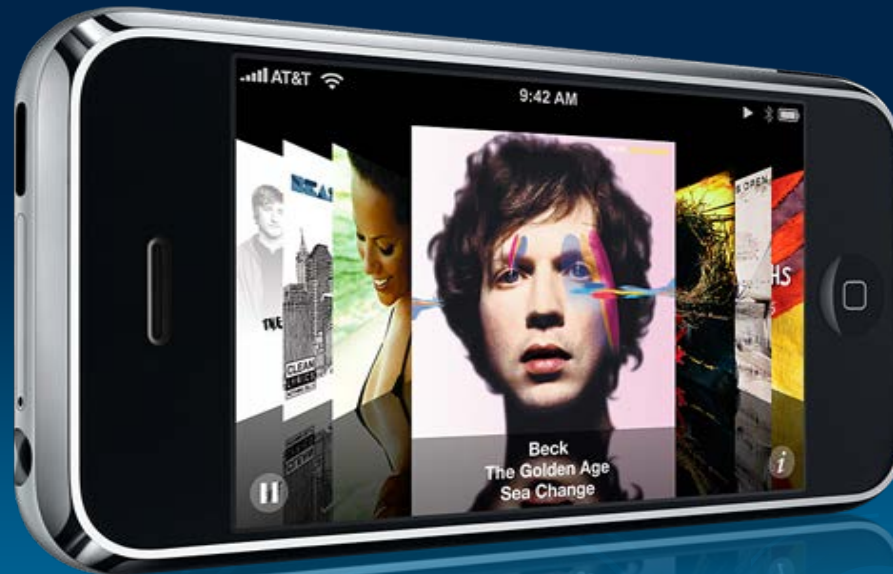
Craftsmanship?



Sustaining Your Architecture

Quality

Quality Definition: a peculiar and essential character or nature, an inherent feature or property, a degree of excellence or grade, a distinguishing attribute or characteristic



Quality (Who's perspective)

Artist important/boring	Scientist true/false
Designer cool/uncool	Engineer good/bad

“The Four Winds of Making”...Gabriel

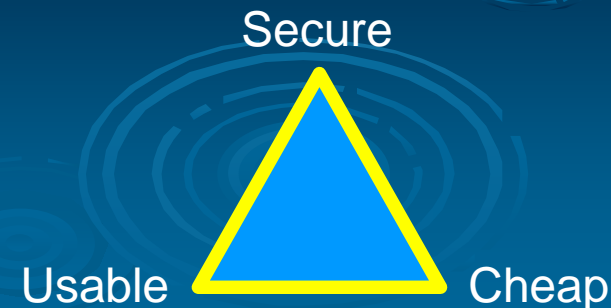
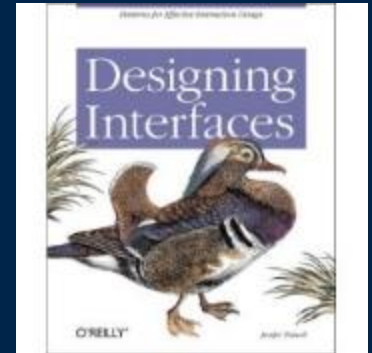
An architect
might have
perspectives
from an artist
to a design or
an engineer!

Rich Gold "The Plenitude: Creativity, Innovation & Making Stuff
(Simplicity: Design, Technology, Business, Life)"

Triggers and Practices – Richard Gabriel <http://www.dreamsongs.com>

Design is about Tradeoffs

- Usability and Security often have *orthogonal* qualities
 - **Designing Interfaces: Patterns for Effective Interaction Design**
 - **Security Patterns: Integrating Security and Systems Engineering**
- Performance vs Small Memory
 - Quality of being good enough

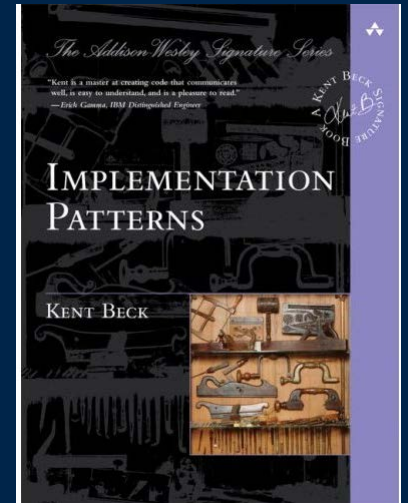


Being Good Enough

- Quality of being good enough.
- Does it meet the minimum requirements
- Quality has many competing forces...are we designing a system for online orders or for controlling the space shuttle, they have different qualities, thus different patterns and solutions apply.
- Perfection is the enemy of ***Good Enough!***
- Maybe Quality without a Number.

Does Quality Code Matter?

Patterns about creating quality code that communicates well, is easy to understand, and is a pleasure to read. Book is about patterns of “Quality” code.



But...Kent states, “...***this book is built on a fragile premise: that good code matters. I’ve seen too much ugly code make too much money to believe that quality of code is either necessary or sufficient for commercial success or widespread use. However I still believe quality of code matters.***”

Patterns assist with making code more bug free and easier to maintain and extend.



Cleaning House

Can we gentrify, rehabilitate,
or make-over code helping
clean up the mud?



Can **refactoring**, patterns, frameworks,
components, agile, and objects help
with mud?

Is it possible to do some **Mud Prevention**
and keep our **Architecture Clean**?

If we have a BBoM

How can we even start?

How can we cordon off the mess?

Clean Code Doesn't Just Happen

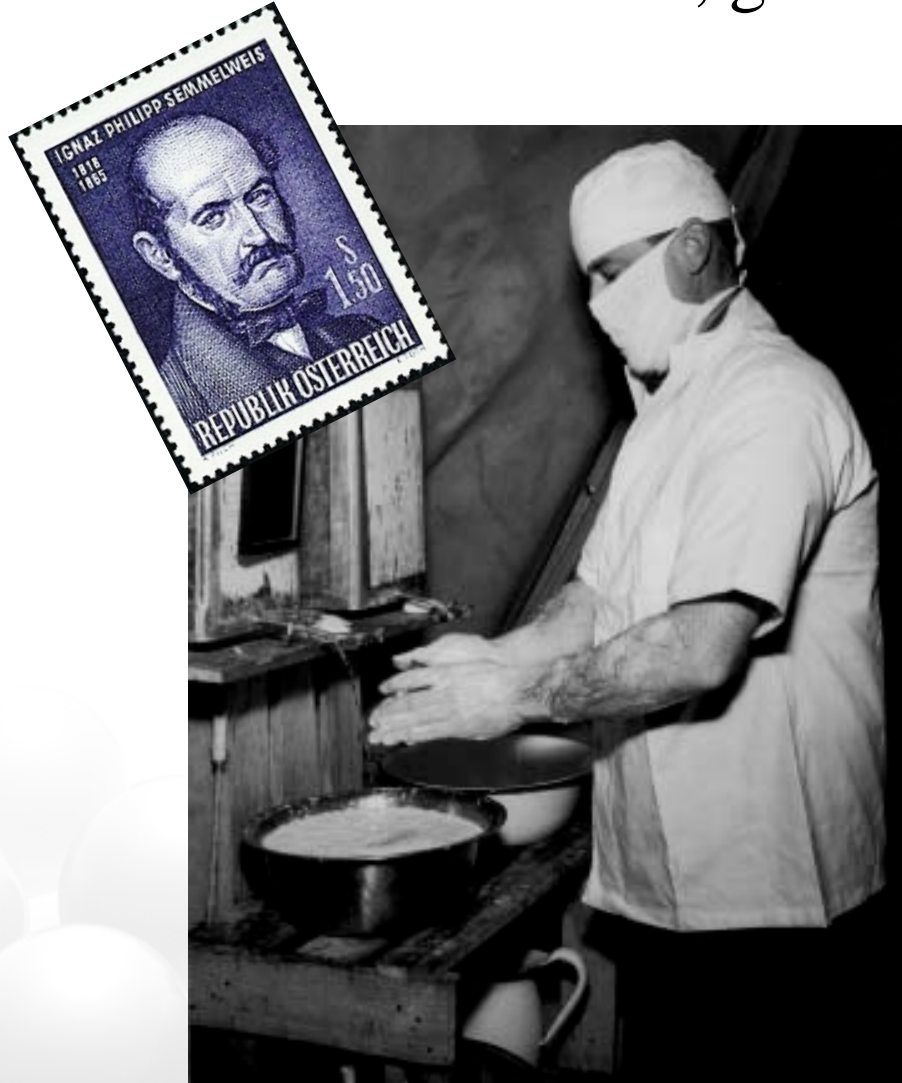
- You have to craft it.
- You have to maintain it.
- You have to make a professional commitment.

“Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.”

– Martin Fowler

Professional Responsibility

There's no time to wash hands, get to the next patient!



Professionalism

Make it your responsibility to create code that:

- Delivers business value
- Is clean
- Is tested
- Is simple
- Follows good design principles



When working with existing code:

- If you break it, you fix it
- You never make it worse than it was
- You always make it better

Legacy Code

Definition:

- Code without tests, Code from last year, Code I didn't write

The goal is to make small, relatively safe changes to get tests in place, so that we can feel comfortable making larger changes

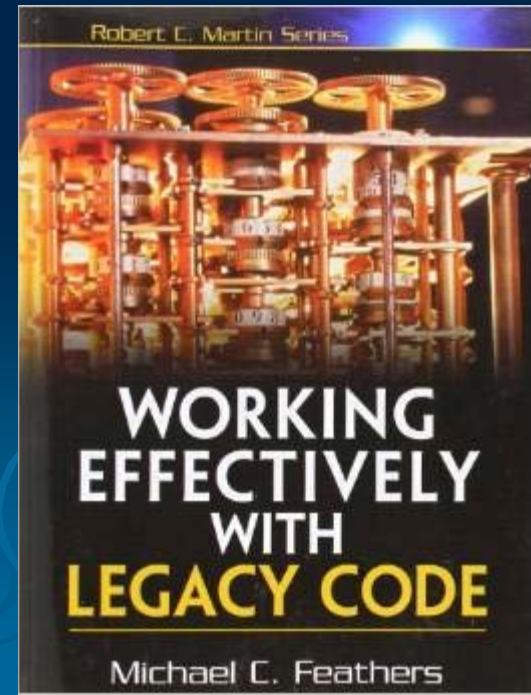
Take a stance - we will not let our code base get any worse

- New code does not have to go into class as existing (untested) code
 - Try to add new code, test driven, in new classes
 - Sprout Method, Sprout Class from VideoStore
- Bugs are opportunities
 - Identify the bug
 - Write a failing test to expose the bug
 - Fix the bug
 - Test now passes



Legacy Code Change Algorithm

- Identify change points
- Find test points
- Break dependencies
- Write tests
- Make changes
and refactor



Sweep It Under the Rug

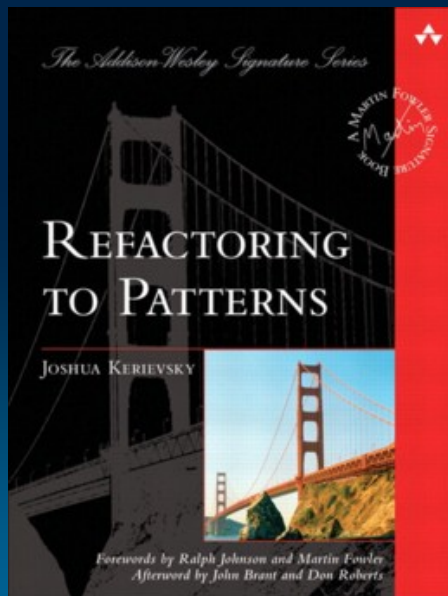


Cover it up to keep other areas clean
(Façade and other Wrapper Patterns)

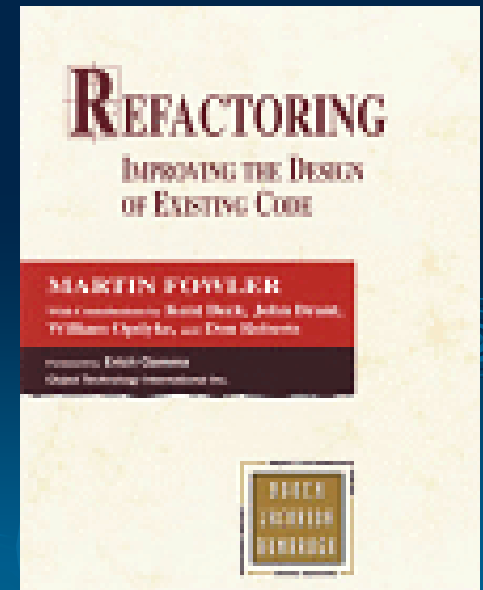
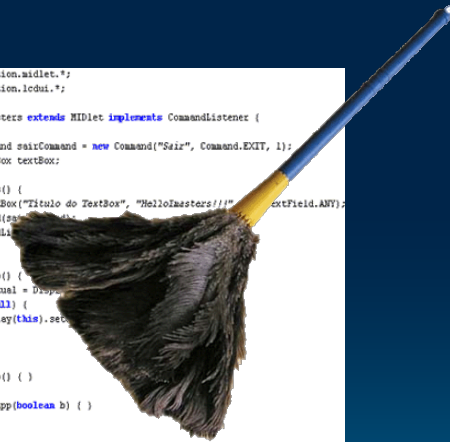
Code Make Over

Refactoring can help reverse some mud. The tradeoff is cost and time....maybe with technology

Refactoring to Better Design (Patterns)...



```
1 import javax.microedition.midlet.*;
2 import javax.microedition.lcdui.*;
3
4 public class HelloMasters extends MIDlet implements CommandListener {
5
6     private final Command sairCommand = new Command("Sair", Command.EXIT, 1);
7     private final TextBox textBox;
8
9     public HelloMasters() {
10         textBox = new TextBox("Título do TextBox", "HelloMasters!!!", TextField.ANY);
11         textBox.addCommand(sairCommand);
12         textBox.setCommandListener(this);
13     }
14
15     public void startApp() {
16         Displayable telaAtual = Displayable.getDisplayable(this);
17         if(telaAtual == null) {
18             Display.getDisplay(this).setCurrent(telaAtual);
19         }
20     }
21
22     public void pauseApp() { }
23
24     public void destroyApp(boolean b) { }
25
26     void sair() {
27         destroyApp(false);
28         notifyDestroyed();
29     }
30
31     public void commandAction(Command c, Displayable d) {
32         if (c == sairCommand) {
33             sair();
34         }
35     }
36 }
```



Code Smells

A *code smell* is a **hint** that something has **gone wrong** somewhere in your code. Use the smell to **track** down the **problem**... **Kent Beck**

Bad Smells in Code was an essay by Kent Beck and Martin Fowler, published as Chapter 3 of:
Refactoring Improving The Design Of Existing Code.

----Ward's Wiki

Have you ever looked at a piece of code that doesn't smell very nice?



Ten Most Putrid List

- 1) Sloppy Layout,
- 2) Dead Code,
- 3) Lamé Names,
- 4) Commented Code,
- 5) Duplicated Code,
- 6) Feature Envy,
- 7) Inappropriate Intimacy,
- 8) Long Methods & Large Class,
- 9) Primitive Obsession & Long Parameter List,
- 10) Switch Statement & Conditional Complexity ...



Lame Names

```
void foo(int x[], int y, int z)
{
  if (z > y + 1)
  {
    int a = x[y], b = y + 1, c = z;
    while (b < c)
    {
      if (x[b] <= a) b++; else {
        int d = x[b]; x[b] = x[--c];
        x[c] = d;
      }
    }
    int e = x[--b]; x[b] = x[y];
    x[y] = e; foo(x, y, b);
    foo(x, c, z);
  }
}
```

```
void quicksort(int array[], int begin, int end) {
  if (end > begin + 1) {
    int pivot = array[begin],
    l = begin + 1, r = end;
    while (l < r) {
      if (array[l] <= pivot)
        l++;
      else
        swap(&array[l], &array[--r]);
    }
    swap(&array[--l], &array[begin]);
    sort(array, begin, l);
    sort(array, r, end);
  }
}
```

<http://dreamsongs.com/Files/BetterScienceThroughArt.pdf>

Sustaining Your Architecture



Fixing Names

Names should *mean something*.

Standards improve communication
- know and follow them.

Standard protocols

object ToString(), Equals()

ArrayList Contains(), Add(), AddRange()

Remove(), Count, RemoveAt(),

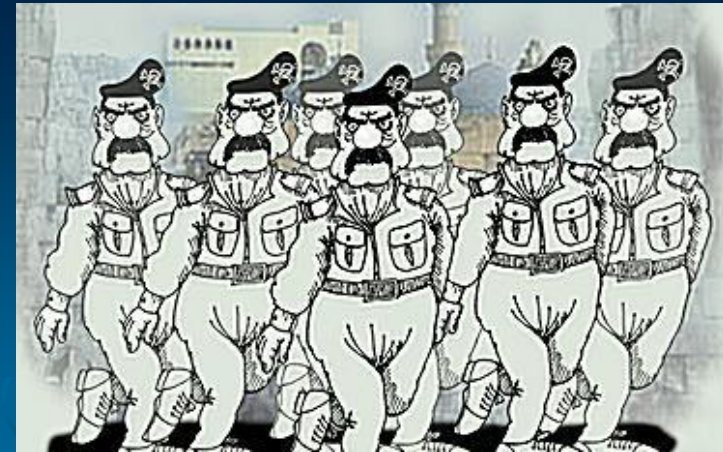
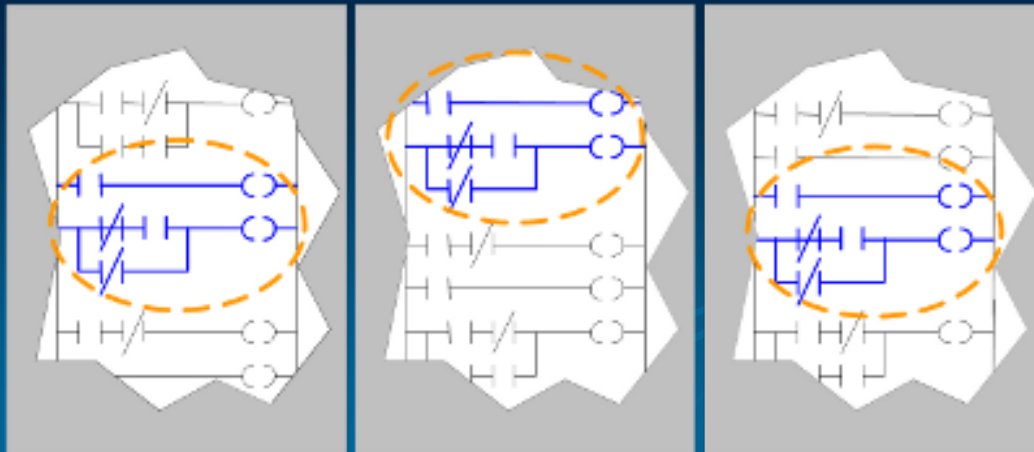
HashTable Keys, ContainsKey(),

ContainsValue()

Standard naming conventions

Duplicate Code

- Do everything exactly once
- Duplicate code makes the system harder to understand and maintain
 - Any change must be duplicated
 - The maintainer has to change every copy



Fixing Duplicate Code

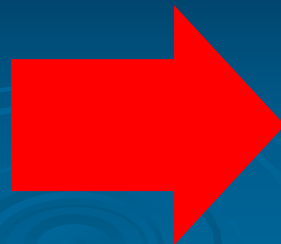
- Do everything exactly once!!!

DRY Principle

- Fixing Code Duplication
 - Move identical methods up to superclass
 - Move methods into common components
 - Break up Large Methods



Do not
duplicate!



REUSE

Inappropriate Intimacy

When classes depend on other's implementation details ...

Tightly coupled classes -
you can't change one without changing the other.

Boundaries between
classes are not
well defined.



Feature Envy

When a class uses a lot the functionality or features of another class

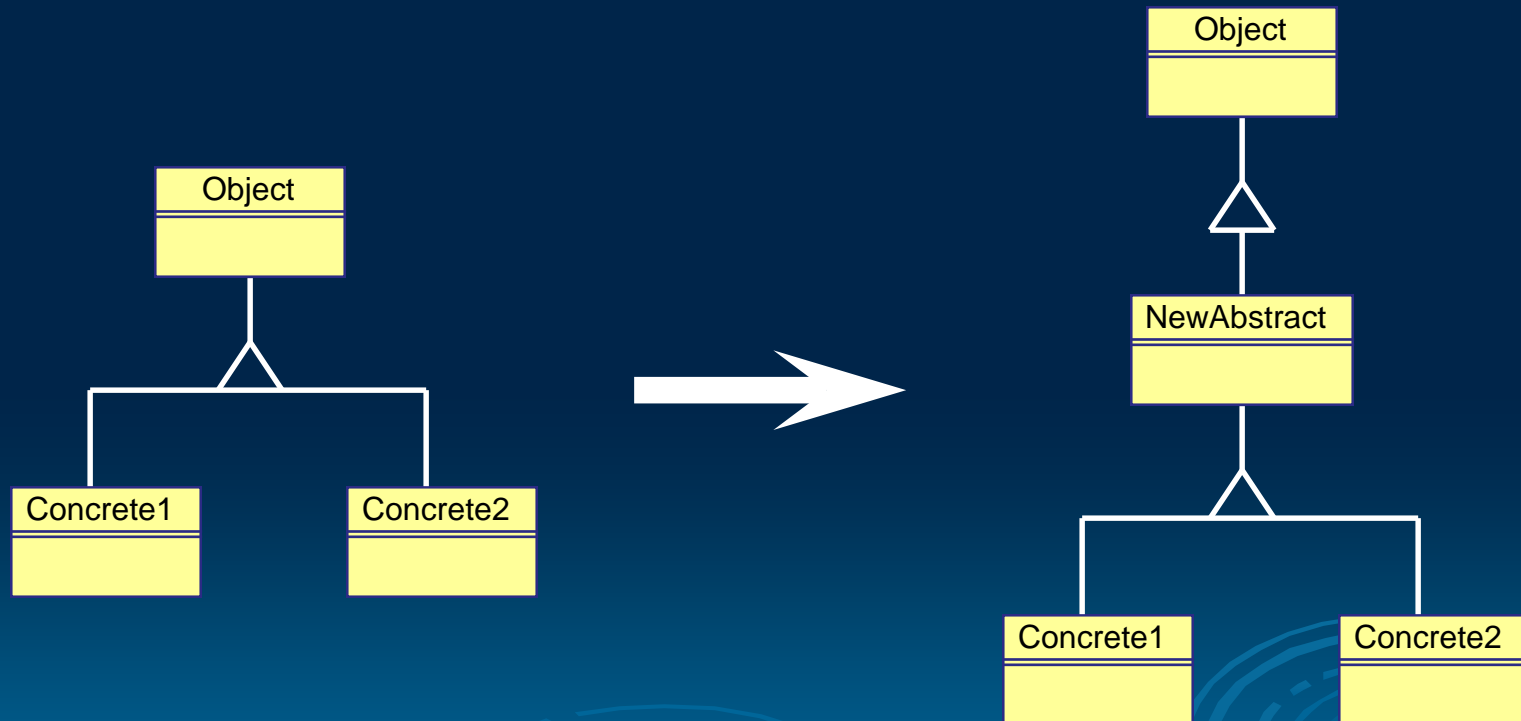
Indicates that some functionality is in the wrong class ... “Move Method”

It creates a tight coupling between these two classes



A Simple Refactoring

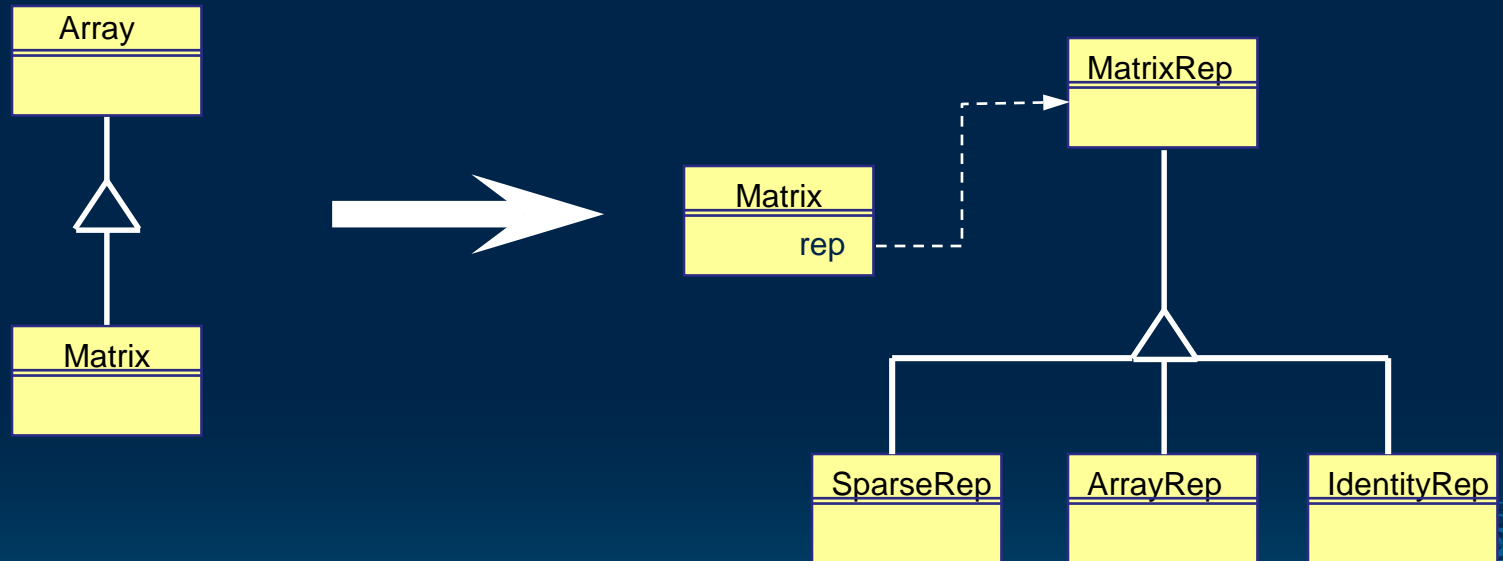
Create Empty Class



Borrwed from Don Roberts, The Refactory, Inc.

Sustaining Your Architecture

A Complex Refactoring

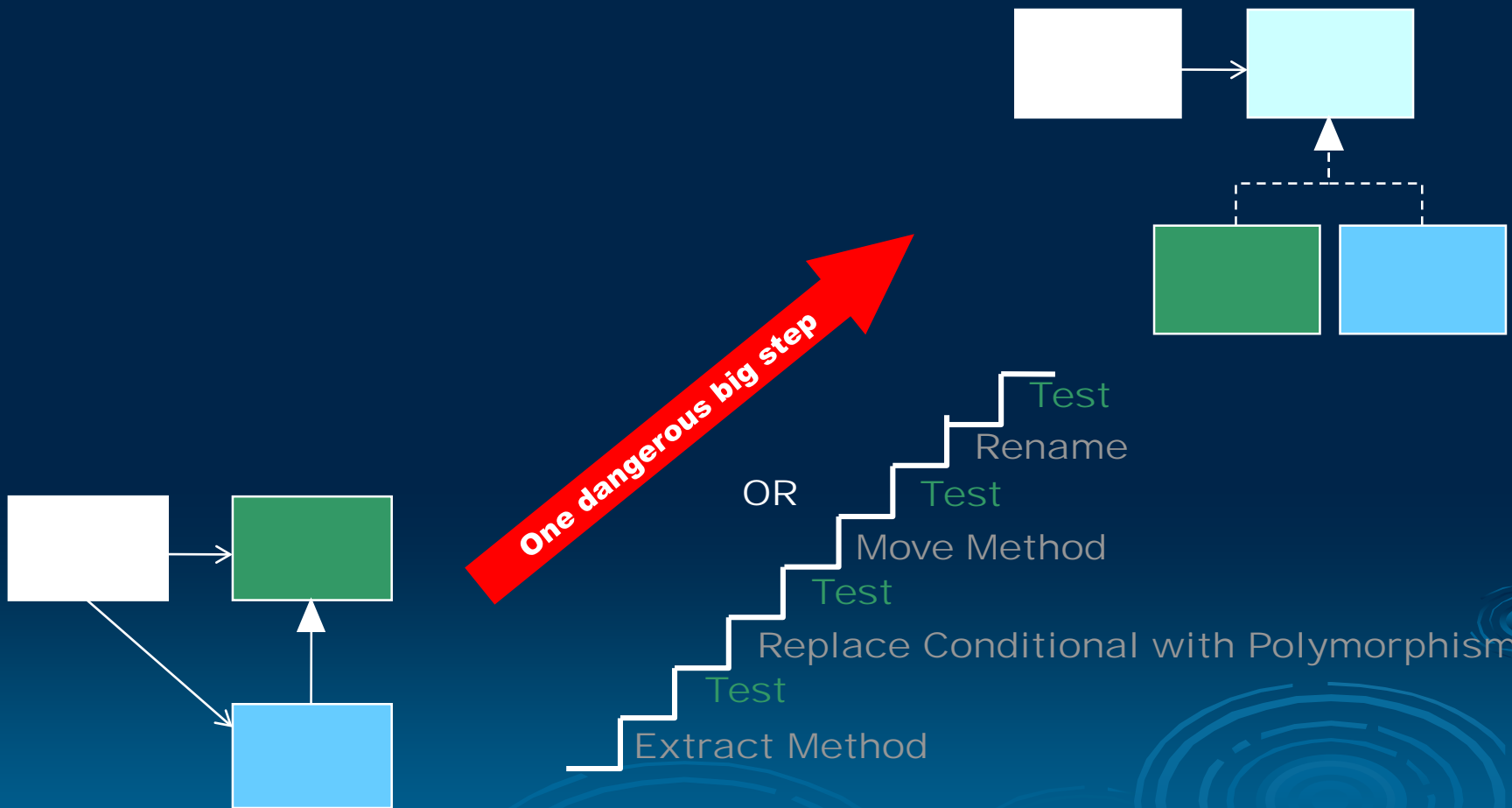


Refactoring can be hard but there are a lot of small steps that lead to big gains in mud busting

Borrowed from Don Roberts, The Refactory, Inc.

Sustaining Your Architecture

Transform Design with Small Steps



Two Refactoring Types*

- Floss Refactorings—frequent, small changes, intermingled with other programming (daily health)
- Root canal refactorings — infrequent, protracted refactoring, during which programmers do nothing else (major repair)



* Emerson Murphy-Hill and Andrew Black in
“Refactoring Tools: Fitness for Purpose”

<http://web.cecs.pdx.edu/~black/publications/IEEESoftwareRefact.pdf>

Sustaining Your Architecture

Common Wisdom

Work refactoring into your daily routine...

“In almost all cases, I’m opposed to setting aside time for refactoring. In my view refactoring is not an activity you set aside time to do.

Refactoring is something you **do all the time** in little bursts.” — Martin Fowler



Refactoring: When to do it

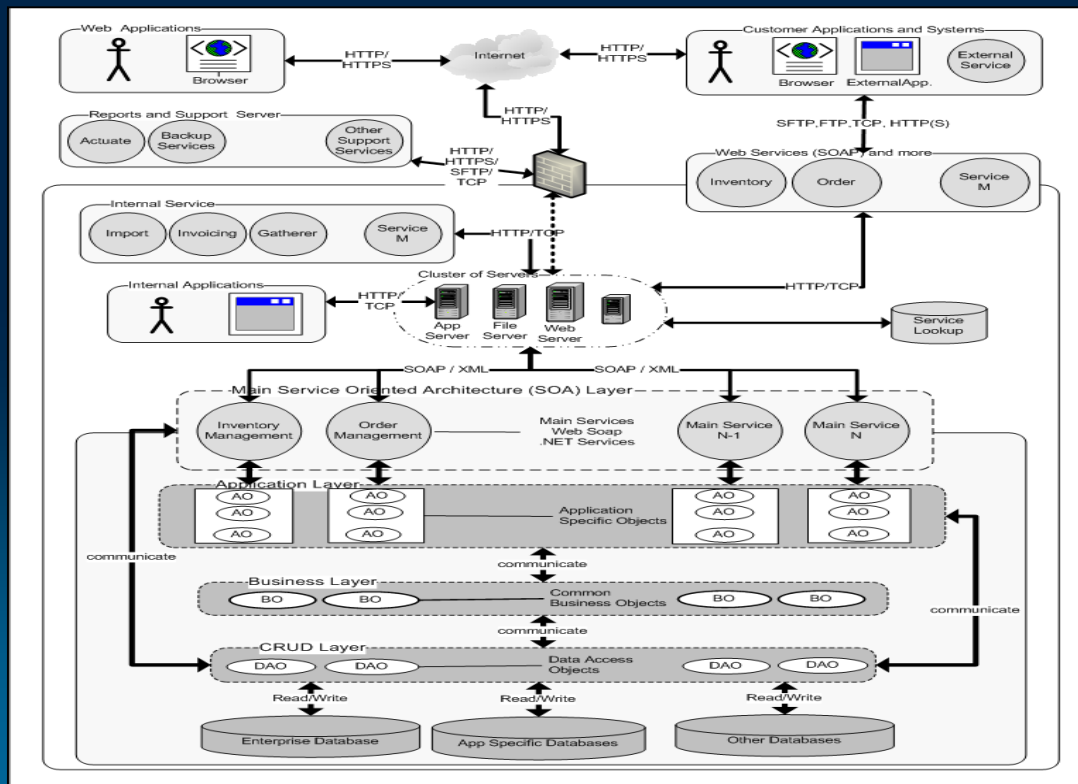
- Regular refactorings make it fairly safe and easy to do anytime. Especially when you have good **TESTS**.
- When you are fixing bugs
- Adding new features
- Right after a release
- Might have to **Refactor Tests** too!!!

Large Refactorings

- Four Big Refactorings
 - Tease Apart Inheritance
 - Convert Procedural Design to Objects
 - Separate Domain from Presentation
 - Extract Hierarchy
- Refactoring and Databases
- Architecture Smells
- Keeping the System Going

Sustaining Architecture

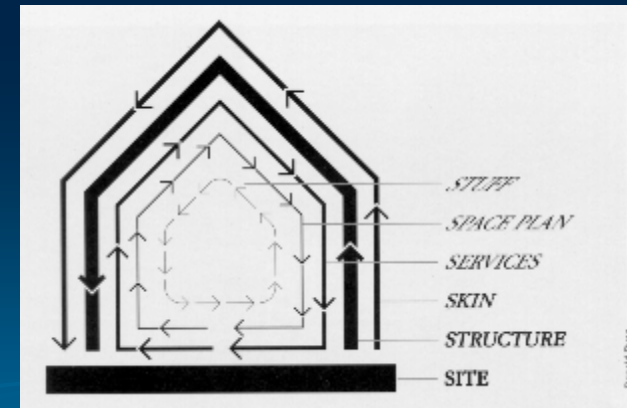
What can be done to help sustain our architecture in the long run?



Sustaining Your Architecture

Stuart Brand's Shearing Layers

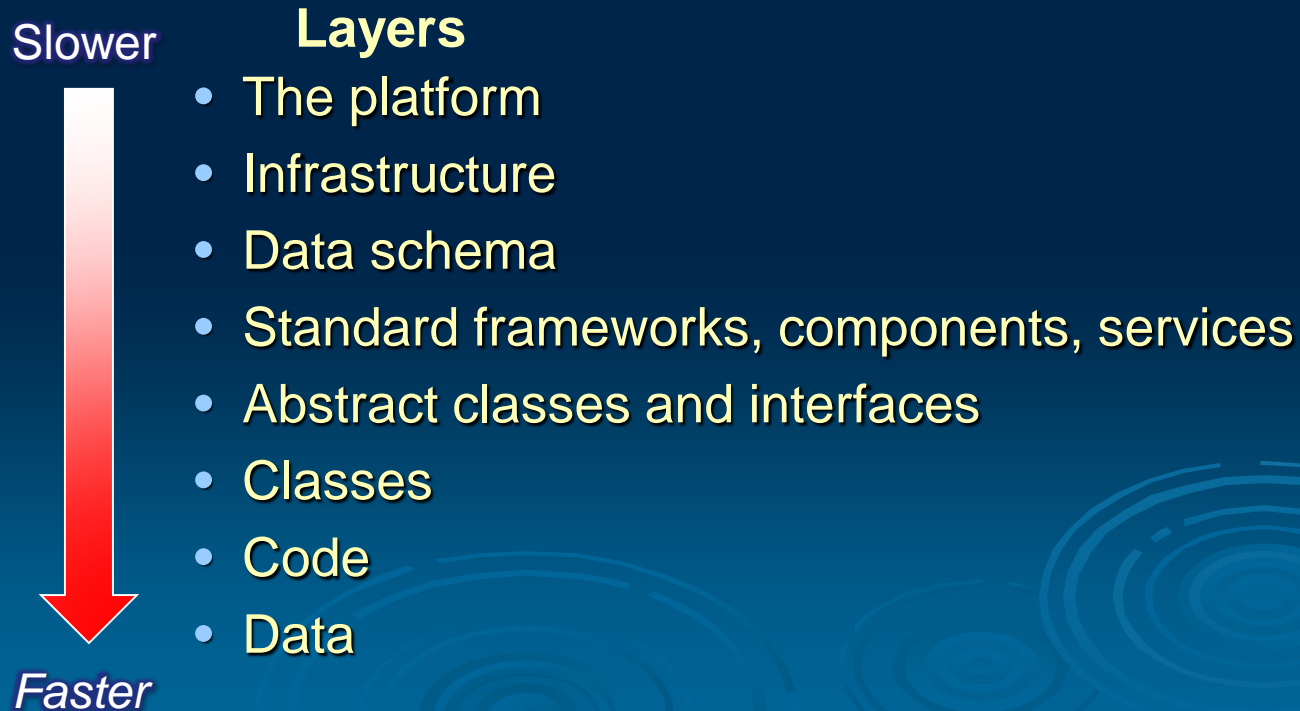
- Buildings are a set of components that evolve in different timescales.
- Layers: site, structure, skin, services, space plan, stuff. Each layer has its own value, and speed of change (pace).
- Buildings adapt because faster layers (services) are not obstructed by slower ones (structure).



—Stuart Brand, *How Buildings Learn*

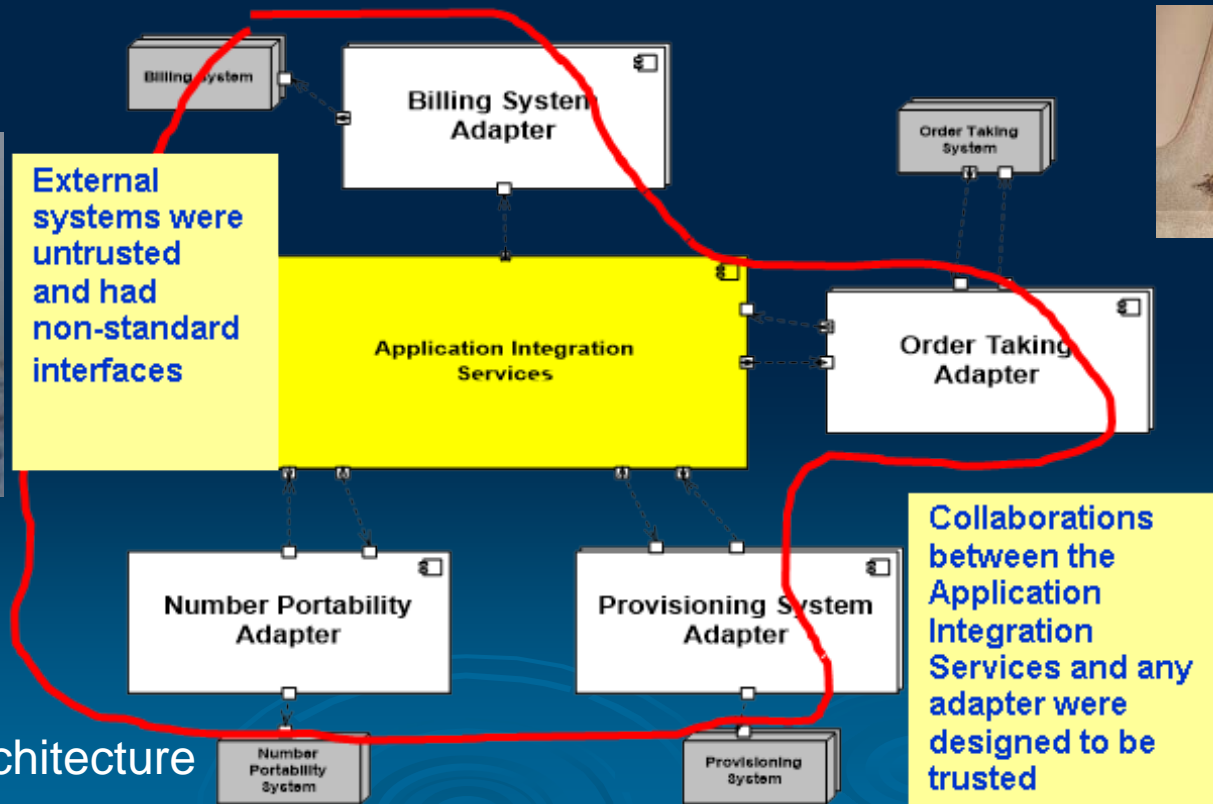
Yoder and Foote's Software Shearing Layers

“Factor your system so that artifacts that change at similar rates are together.”—Foote & Yoder, Ball of Mud, PLoPD4.



Put a Rug at the Front Door

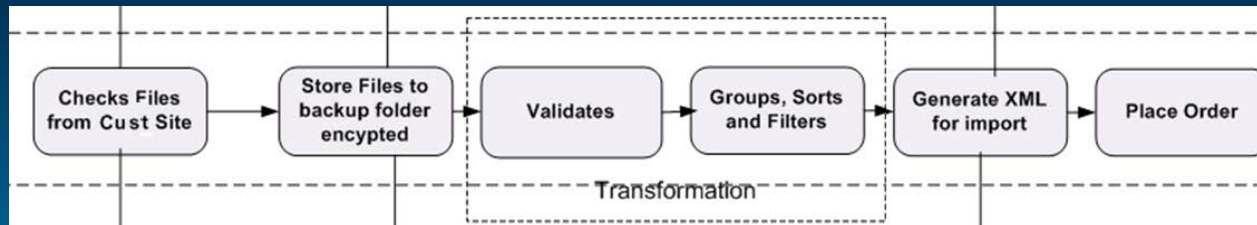
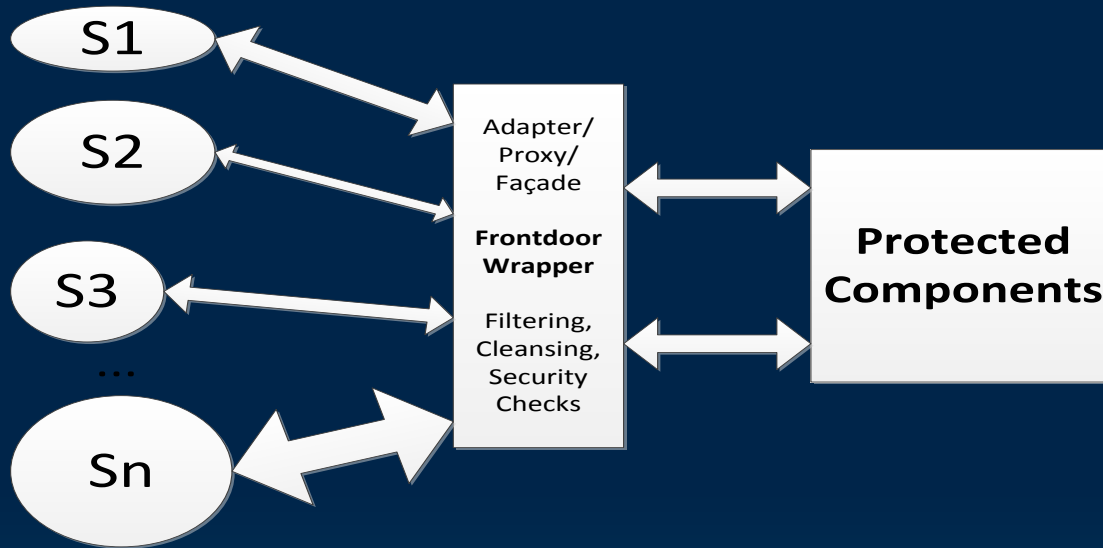
*ALIAS: ENCAPSULATE AND IGNORE
KEEPING THE INTERNALS CLEAN*



Patterns for
Sustaining Architecture
PLoP 2012 Paper

Sustaining Your Architecture

Wipe your Feet at the Front Door



**Filtering and Cleansing Sequence to keep
Place Order Interface Clean**

Sustaining Your Architecture

Paving over the Wagon Trail



Patterns for Sustaining Architecture
PLoP 2012 Paper

ALIAS: MAKE REPETITIVE TASKS EASIER
STREAMLINING REPETITIVE CODE TASKS

Create simple examples, templates, & scripts

Develop a tool that generates code

Identify and use existing tools or frameworks

Develop a framework &/or runtime environment

Develop a domain-specific language

Continuous Inspection



Asian PLoP 2014 Paper

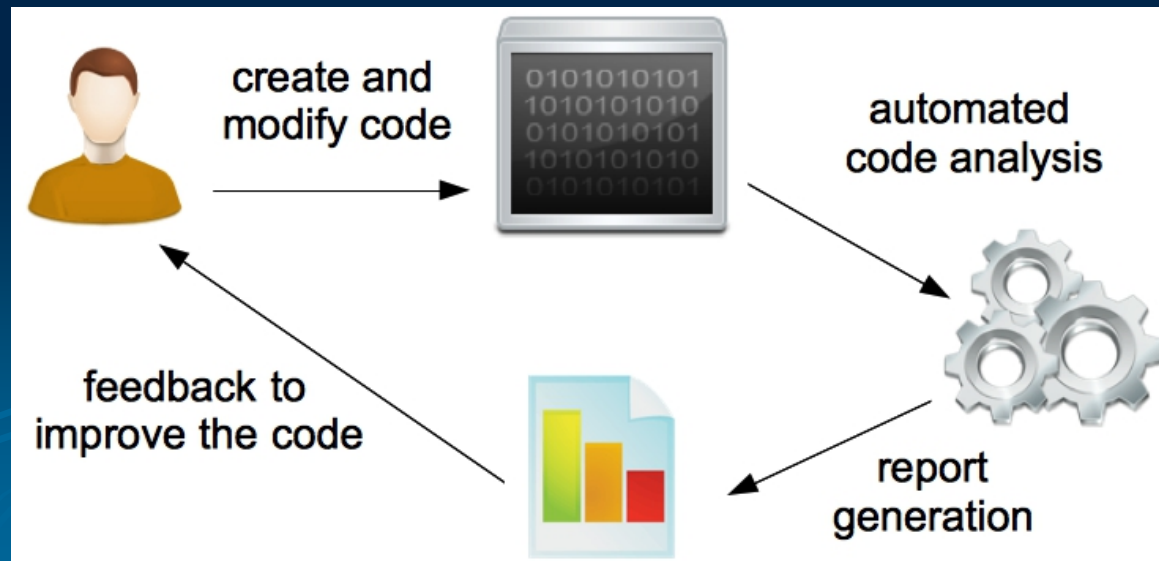
CODE SMELL DETECTION

***METRICS (TEST COVERAGE, CYCLOMATIC COMPLEXITY,
TECHNICAL DEBT, SIZES, ...)***

SECURITY CHECKS

***ARCHITECTURAL
CONFORMANCE***

***AUTOMATE WHERE
YOU CAN!!!***



Sustaining Your Architecture

Sustainable Architecture

➤ Stewardship

- Follow through
- Ongoing attention
- Not ignoring the little things that can undermine our ability to grow, change and adapt our systems



Architectural Practice: Reduce Technical Debt

- Integrate new learning into your code
 - Refactoring
 - Redesign
 - Rework
 - Code clean up
- Unit tests (functionality)
- Test for architectural qualities (performance, reliability,...)



Agile Values Drive Architectural Practices

- Do something. Don't debate or discuss architecture too long
- Do something that buys you information
- Prove your architecture ideas
- Reduce risks
- Make it testable
- Prototype realistic scenarios that answer specific questions
- Incrementally refine your architecture
- Defer architectural decisions that don't need to be immediately made



Do
something!
Prove &
Refine.

Indicators You've Paid Enough Attention to Architecture

- Defects are localized
- Stable interfaces
- Consistency
- Developers can easily add new functionality
- New functionality doesn't "break" existing architecture
- Few areas that developers avoid because they are too difficult to work in
- Able to incrementally integrate new functionality





Be Agile
or you will

Agile Mindset

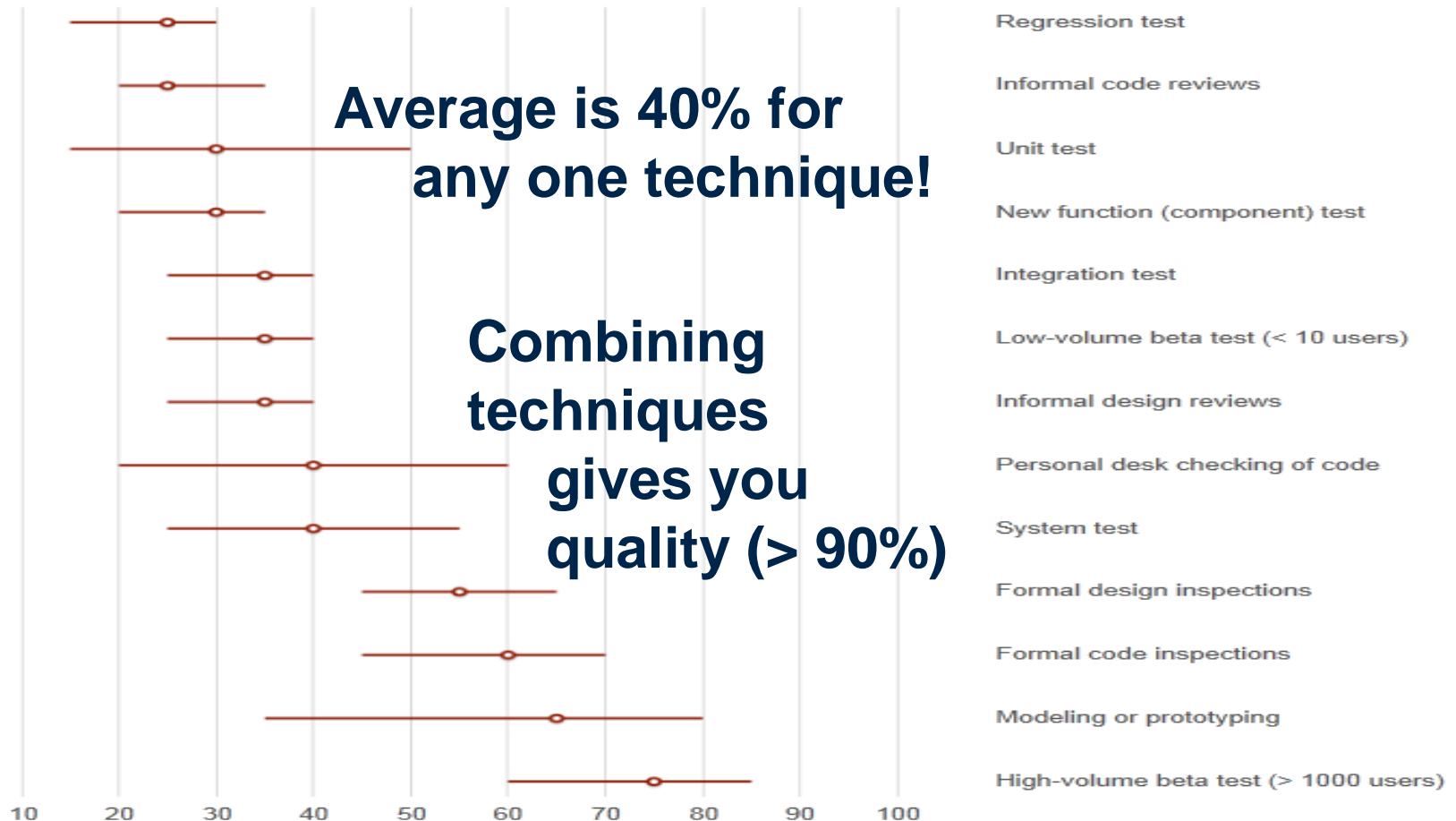
Other Techniques for Improving Quality

Steve McConnell

<http://kev.inburke.com/kevin/the-best-ways-to-find-bugs-in-your-code/>

**Average is 40% for
any one technique!**

**Combining
techniques
gives you
quality (> 90%)**



Draining the Swamp

You can escape from the
“*Spaghetti Code Jungle*”

Indeed you can transform the landscape.
The key is not some magic bullet, but a
long-term commitment to **architecture**,
and to cultivating and refining “*quality*”
artifacts for your domain (**Refactoring**)!

Patterns of the best practices can help!!!

Silver Buckshot

There are no silver bullets

...Fred Brooks

But maybe some silver buckshot

...promising attacks

Good Design

Frameworks

Patterns

Architecture

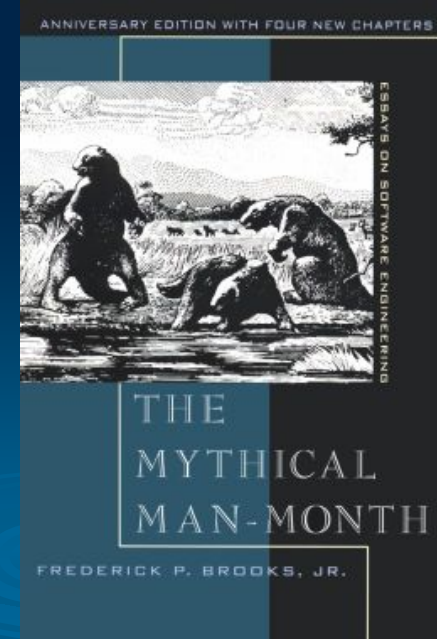
Process/Organization

Tools and Support

Refactoring

Good People* **

Sustaining Your Architecture



So There is Some Hope!!!

Testing (TDD), **Refactoring**, Regular Feedback, **Patterns**, More Eyes, ...

Good People!!!

Continuous attention to **technical excellence!**

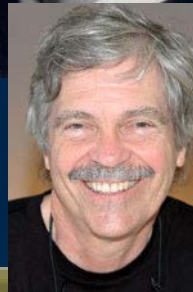
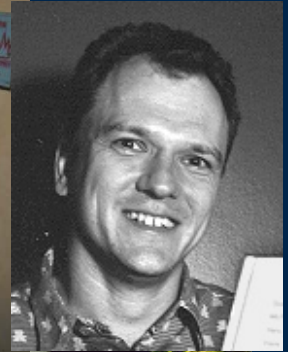
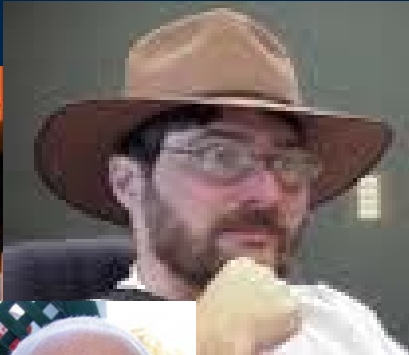
Retrospectives! Face-To-Face conversations.

Diligence and Hard Work!

Motivated individuals with the *environment* and *support* they need.

But, Maybe Mud is why we have Agile...

It Takes a Village



Thanks!

Questions?



joe@refactory.com
Twitter: @metayoda