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

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?
Why BBoM?

Why was this phenomenon so prevalent in our industry? We sure talk a good game.

We had seen where Lisp had failed, Smalltalk was starting to fail, Windows was winning. Why was this?

What is there about some systems that failed compared to systems that succeed, even when they seemed better?

Where Mud Comes From?

People Write Code → People make Mud
Where Mud Comes From!

Software Tectonics

Reconstruction
- Major Upheaval
- Throw it away

Incremental Change
- Evolution
- Piecemeal Growth

Throwaway Code
Legacy Mush
Urban Sprawl
Slash and Burn Tactics
Merciless Deadlines
Sheer Neglect

Keep it Working, Piecemeal Growth, Throwaway Code
Copy ‘n’ Paste

The Age of Sampling & Big Bucket of Glue
The Mystery of Dark Matter

Accidental Complexity?? Perhaps our current state of the art leads to Mud!

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.
Do Some Agile Principles

Encourage mud?

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!!!

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.
Worse is Better

Ideas 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 is the Payoff?

The question that keeps getting asked is what value does the customer get from paying back this technical debt? What value does the customer get from simplifying this design? What value does the customer get from cleaning this code?

…

The answer is almost universally – none!!!

…Daniel Hinz comment on Brian Marick’s Blog
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.

Some Answers to Mud!?!?

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

Can refactoring, patterns, frameworks, components, agile, and objects help with mud?
Total Code Makeover

Can we just Refactor out of Mud?
Sweep the Mess Under the Rug?

Total Code Makeover

Escape From The Spaghetti Code Jungle
If we have a BBoM

How can we even start?

How can we cordon off the mess?

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.

- The platform
- Infrastructure
- Data schema
- Standard frameworks, components, services
- Abstract classes and interfaces
- Classes
- Code
- Data

Sweep It Under the Rug

Cover it up to keep other areas clean
(Façade and other Wrapper Patterns)
**Put a rug at the Front Door**

**Protect Important Components!**

Keep other parts of the system clean.

Sometimes Glue code (Mediators) helps keep others parts of the system cleaner.

(Anti-Corruption Layer -- Eric Evans)

---

**Wipe your Feet at the Front Door**

**ALIAS: ENCAPSULATE AND IGNORE KEEPING THE INTERNALS CLEAN**

Patterns for Sustaining Architecture

PLoP 2012 Paper
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

Code Make Over

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

Refactoring to Better Design (Patterns)…
Refactorings

Behavior Preserving Program Transformations

- Rename Instance Variable
- Promote Method to Superclass
- Move Method to Component

Always done for a reason!!!

Refactoring is key and integral to most Agile processes!!!

A Simple Refactoring

Create Empty Class

Borrowed from Don Roberts, The Refactory, Inc.

Escape From The Spaghetti Code Jungle
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.

Escape From The Spaghetti Code Jungle

At every step, the tests should be executed to verify if everything is still working!

Refactoring is performed in small steps to remove bad smells and reach the desired design
Testing is Key!!!

You Must Test

When you find smelly code, you often apply refactorings to clean your code.

Testing is a key principle for safe refactoring!
Transform Design with Small Steps

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.

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

--- Ward’s Wiki

Escape From The Spaghetti Code Jungle
Ten Most Putrid List

1) Sloppy Layout,
2) Dead Code,
3) Lame 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 …

Bad Formatting

```
public class SyntaxHighlighterTest {
    public static void main(String[] args) {
        System.out.println("Nice highlighting!");
    }
}
```

```
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); bar(x, c, z);
    }
}
```

```
void bar(int i[], int j, int k)
{ return i[j] = int k}
```
Dead Code

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];
                return;
                x[c] = d;
            }
        }
        x[b] = a;
    }
    y = 5; // set y equal to 5
    int e = x[--b]; x[b] = x[y];
    /* used to use bar,
     * might need it again
     * bar(x, c, z); */
}

void bar(int i[], int j, int k) {
    /* bar method returning nothing */
    if (j > k) {
        return k;
        i[k] = i[j];
    }
    if (j == k) {
        return i[j] = int [k];
    }
}

Fix the Layout and Remove Useless Items

Format the Code Consistently
Agree on a standard format
Set the tools for consistent formatting
Run the tools over the code base
Remove Unreachable Code
Delete useless comments
Delete commented out code
Remove code that can’t be reached,
Lame Names

```c
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);
    }
}
```

```c
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);
    }
}
```


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!**
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
Switch Statements

Many switch statements or nested conditionals throughout methods

```java
double getSpeed() {
    switch (_type) {
        case EUROPEAN:
            return getBaseSpeed();
        case AFRICAN:
            return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
        case NORWEGIAN_BLUE:
            return (_isNailed) ? 0 : getBaseSpeed(_voltage);
    }
    throw new RuntimeException("unreachable");
}
```

Replace Conditional with Polymorphism

Rather than switching use method names to do the cases (double dispatch)

```java
double getSpeed() {
    switch (_type) {
        case EUROPEAN:
            return getBaseSpeed();
        case AFRICAN:
            return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
        case NORWEGIAN_BLUE:
            return (_isNailed) ? 0 : getBaseSpeed(_voltage);
    }
    throw new RuntimeException("unreachable");
}
```

Use polymorphism or overriding of hook methods (new cases do not change existing code)
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!!!
Refactoring Addresses Some Key Leverage Points

Refactoring is a technique that works with Brooks’ “promising attacks” (from “No Silver Bullet”):

- buy rather than build: restructuring interfaces to support commercial SW
- grow don’t build software: software growth involves restructuring
- requirements refinements and rapid prototyping: refactoring supports such design exploration, and adapting to changing customer needs
- support great designers: a tool in a designer’s tool chest

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)

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

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”

Thinking Fast vs. Slow

- Fast thinking: decisions based on intuition, biases, ingrained patterns, and emotions
- Slow thinking: Reasoning, logical thinking, contemplating
Take Time For Both

- **Slow thinking**
  - Pairing and discussing options or why you want to implement something a certain way
  - Sketching, noodling, design spikes

- **Fast thinking**
  - Following your intuition, deciding on the fly
  - Fast turns of coding, testing and quick fixes… (Red/Green)

Another View of Test-Driven Development

- Requirements Envisioning (days/weeks/…)
- Architecture Envisioning (days/weeks/…)
- Iteration Modeling (hours)
- Model Storming (minutes)
- Fast TDD (hours)
- Iteration n: Development

Conceptual Modeling

Example Driven Design

*a little bit of modeling then a lot of coding*
Other Techniques for Improving Quality

Steve McConnell

Average is 40% for any one technique!

Combining techniques gives you quality (> 90%)

Can tools Help?

What is the role of tools in draining these swamps?

What kinds of tools and practices might forestall software entropy; is mud preventable?

Tools can help, but too often too much is put on tools as the solution to all our problems.

Refactoring Tools, Testing Tools, XUnit, Lint Tools, Code Critics, …
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 ***
Mud is Here…

*It isn’t always bad!*
*It can be contained!*
*It can be cleaned up!*

Our code can be more habitable!

---

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…*
Dogmatic

Synonyms: bullheaded, dictative, doctrinaire, fanatical, intolerant

Antonyms: amenable, flexible, manageable

Pragmatic

Synonyms: common, commonsense, logical, practical, rational, realistic, sensible

Antonyms: idealistic, unrealistic

Be Agile do you?

Agile Mindset
It Takes a Village

Dziękuję!!!

joe@refactory.com
Twitter: @metayoda
Dziękuję!!!

joe@refactory.com
Twitter: @metayoda