Escape from the Spaghetti Code Jungle

“Big Ball of Mud”

Brian Foote
Joseph Yoder

The Refactory, Inc

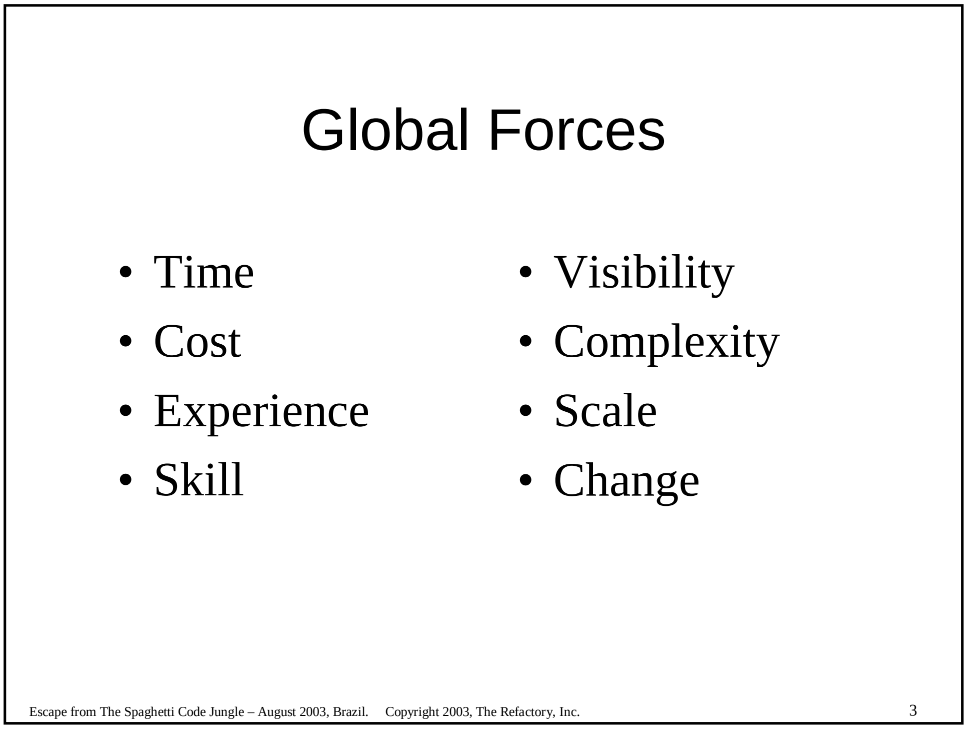
University of Illinois at Urbana-Champaign

Big Ball of Mud

Alias: Shantytown, Spaghetti Code

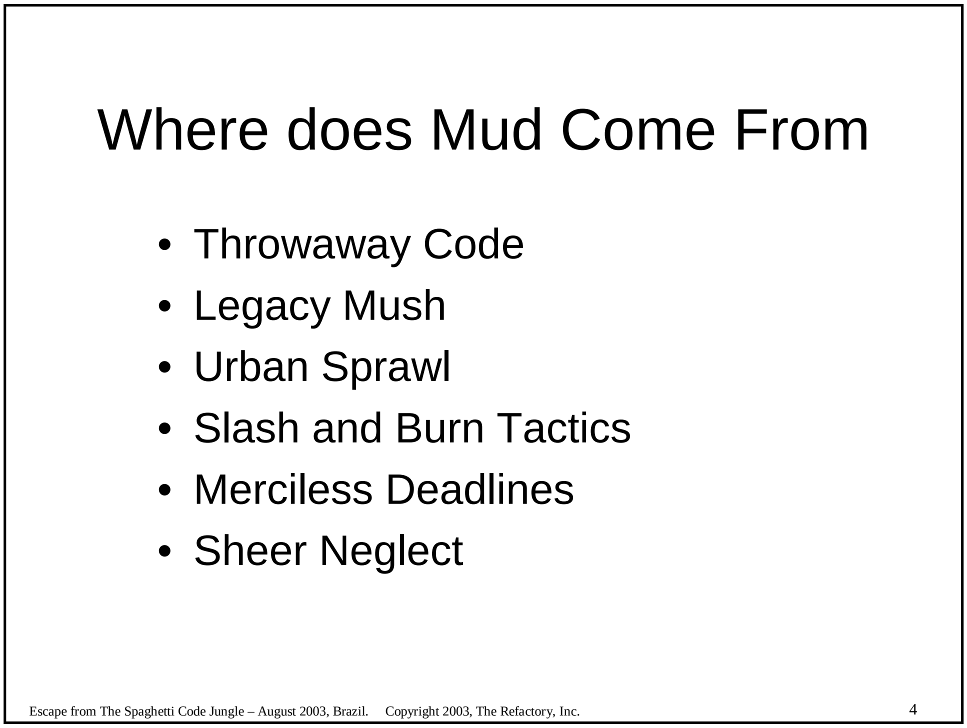
A BIG BALL OF MUD is haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle.

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



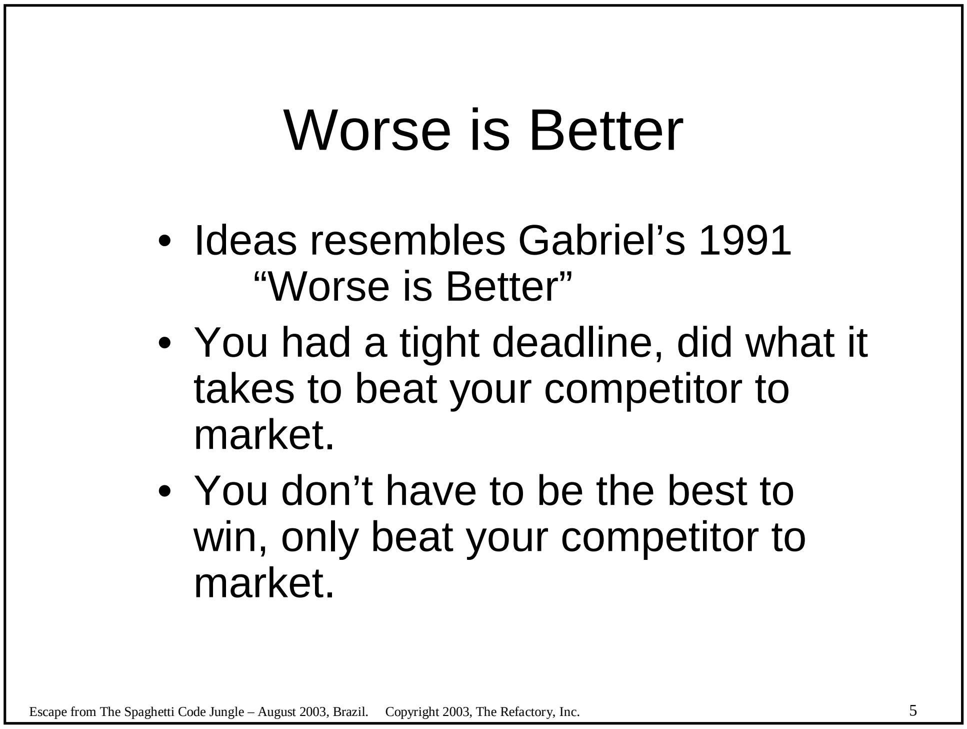
Global Forces

- Time
- Cost
- Experience
- Skill
- Visibility
- Complexity
- Scale
- Change



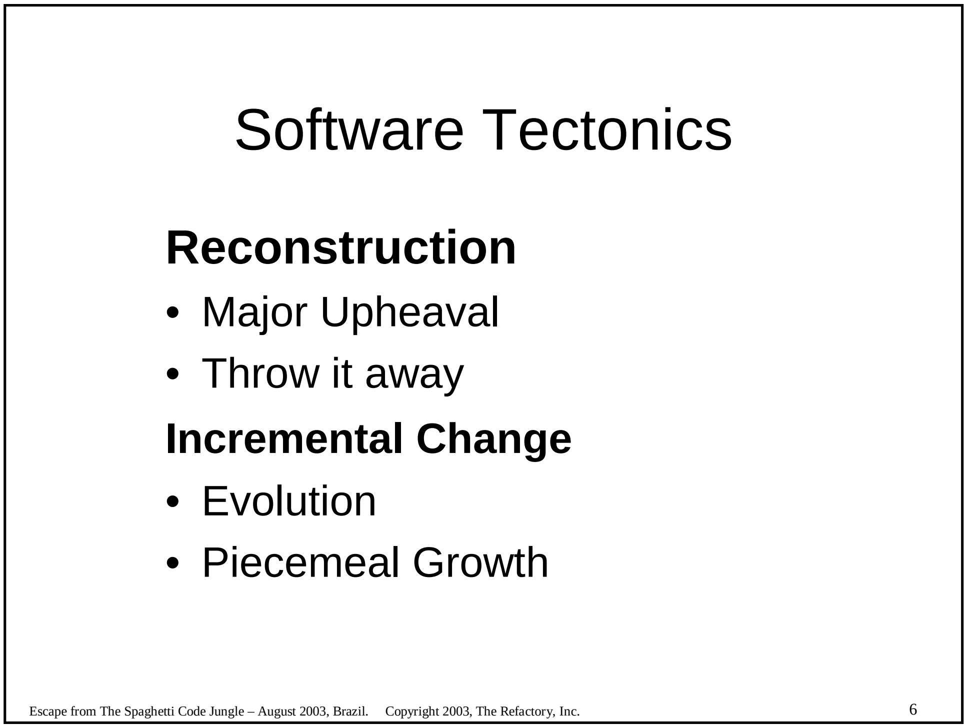
Where does Mud Come From

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



Worse is Better

- Ideas resembles Gabriel’s 1991 “Worse is Better”
- You had a tight deadline, did what it takes to beat your competitor to market.
- You don’t have to be the best to win, only beat your competitor to market.



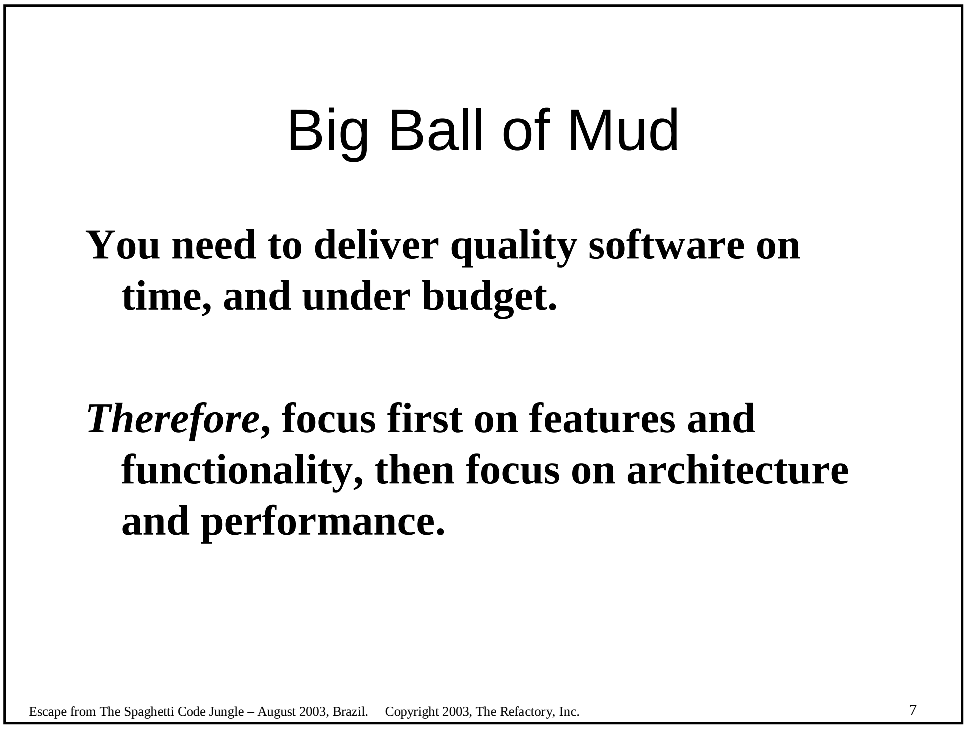
Software Tectonics

Reconstruction

- Major Upheaval
- Throw it away

Incremental Change

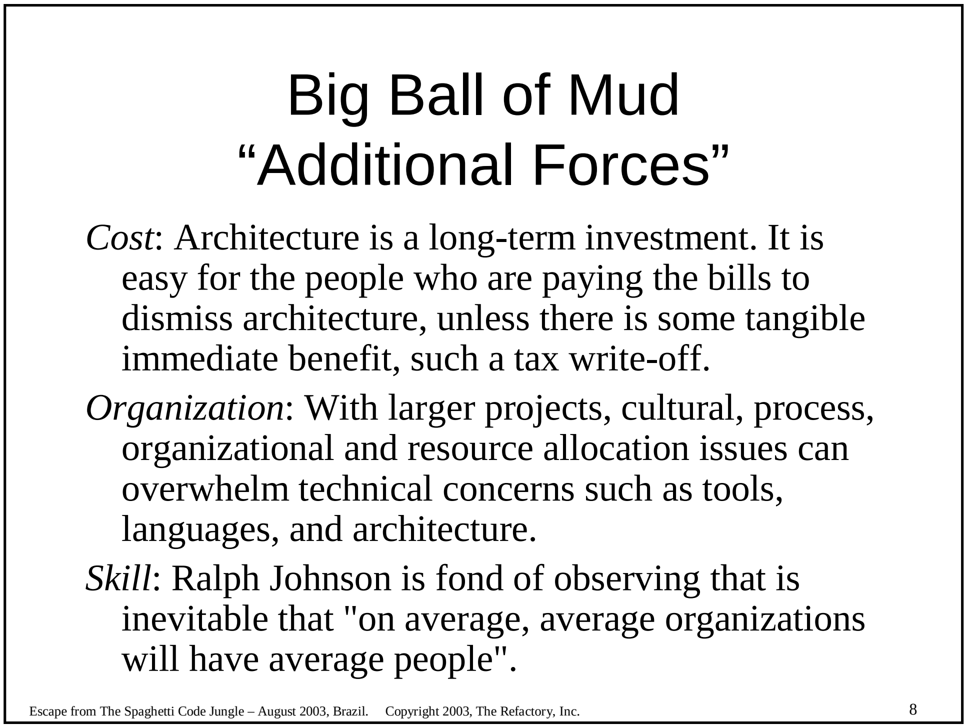
- Evolution
- Piecemeal Growth



Big Ball of Mud

You need to deliver quality software on time, and under budget.

***Therefore*, focus first on features and functionality, then focus on architecture and performance.**



Big Ball of Mud
“Additional Forces”

Cost: Architecture is a long-term investment. It is easy for the people who are paying the bills to dismiss architecture, unless there is some tangible immediate benefit, such a tax write-off.

Organization: With larger projects, cultural, process, organizational and resource allocation issues can overwhelm technical concerns such as tools, languages, and architecture.

Skill: Ralph Johnson is fond of observing that is inevitable that "on average, average organizations will have average people".

Big Ball of Mud

Why does so much software, despite the best intentions and efforts of developers, turn into BIG BALLS OF MUD? Why do slash-and-burn tactics drive out elegance? Does bad architecture drive out good architecture?

What does this muddy code look like to the programmers in the trenches who must confront it? Data structures may be haphazardly constructed, or even next to non-existent. Everything talks to everything else. Every shred of important state data may be global.

Big Ball of Mud

As per CONWAY’S LAW [Coplien 1995], architects depart in futility, while engineers who have mastered the muddy details of the system they have built in their images prevail. [Foote & Yoder 1998] went so far as to observe that inscrutable code might, in fact, have a survival advantage over good code, by virtue of being difficult to comprehend and change. This advantage can extend to those programmers who can find their ways around such code.

Throwaway Code

(Killer Demo, Quick Hack, Scripting, Killer Demo)

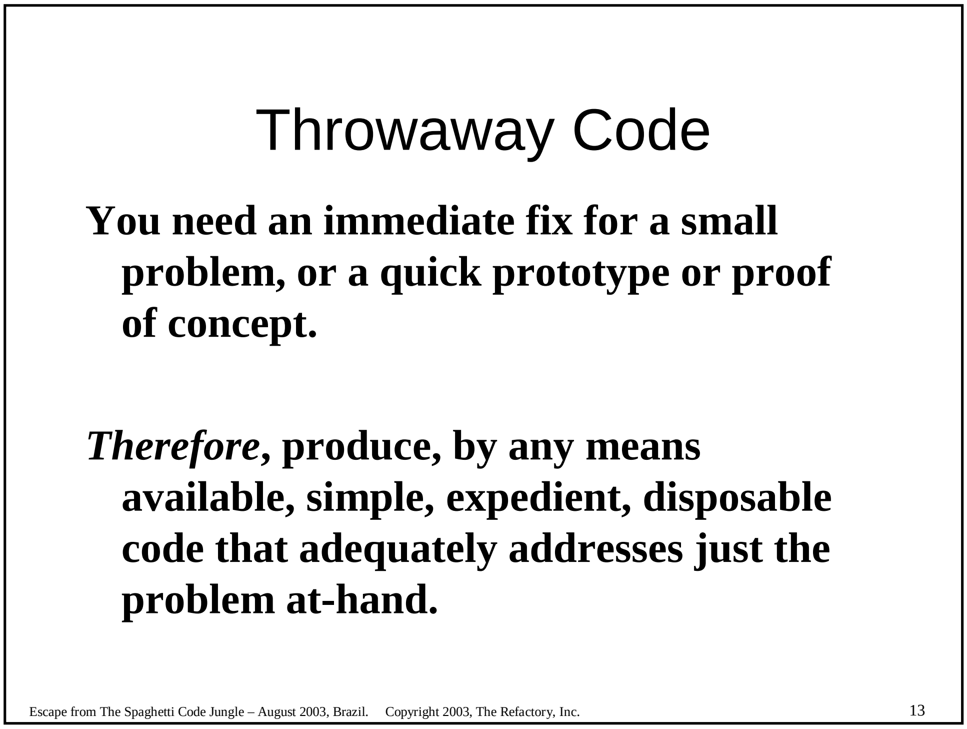
Sometimes this is the *right* approach

There is the danger that such code will take on a life of its own

The original Wiki was an example of Throwaway Code.

Throwaway Code

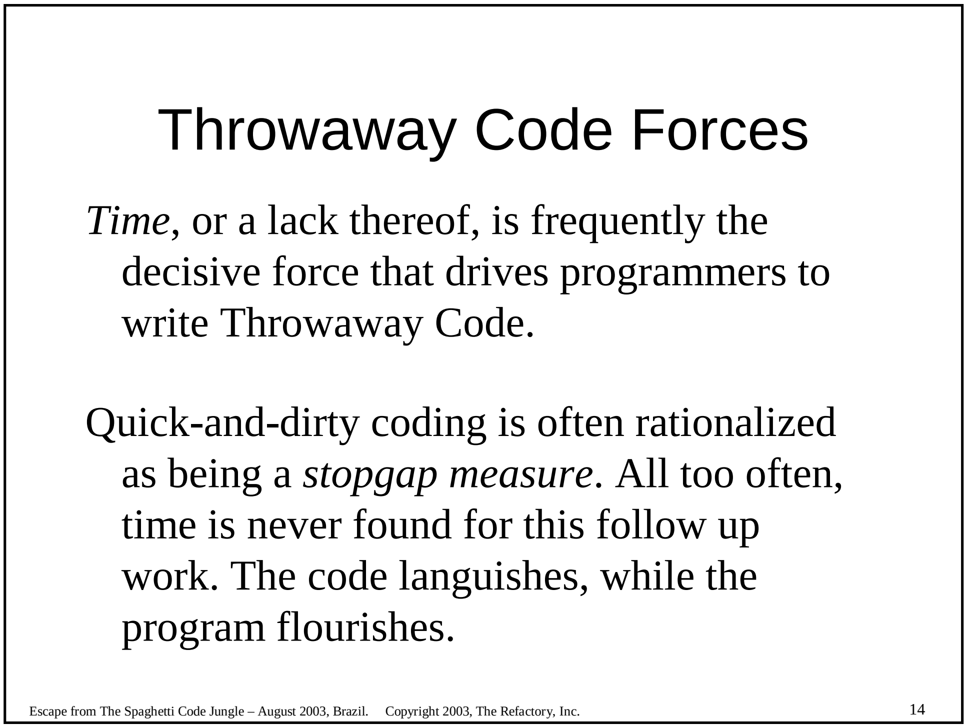
When you are prototyping a system, you are not usually concerned with how elegant or efficient your code is. You know that you will only use it to prove a concept. Once the prototype is done, the code will be thrown away and written properly. As the time nears to demonstrate the prototype, the temptation to load it with impressive but utterly inefficient realizations of the system’s expected eventual functionality can be hard to resist. Sometimes, this strategy can be a bit too successful. The client, rather than funding the next phase of the project, may slate the prototype itself for release.



Throwaway Code

You need an immediate fix for a small problem, or a quick prototype or proof of concept.

***Therefore*, produce, by any means available, simple, expedient, disposable code that adequately addresses just the problem at-hand.**



Throwaway Code Forces

Time, or a lack thereof, is frequently the decisive force that drives programmers to write Throwaway Code.

Quick-and-dirty coding is often rationalized as being a *stopgap measure*. All too often, time is never found for this follow up work. The code languishes, while the program flourishes.

Throwaway Code

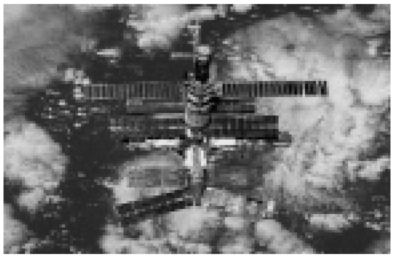
THROWAWAY CODE is often written as an alternative to reusing someone else's more complex code. When the deadline looms, the certainty that you can produce a sloppy program that works yourself can outweigh the unknown cost of learning and mastering someone else's library or framework.

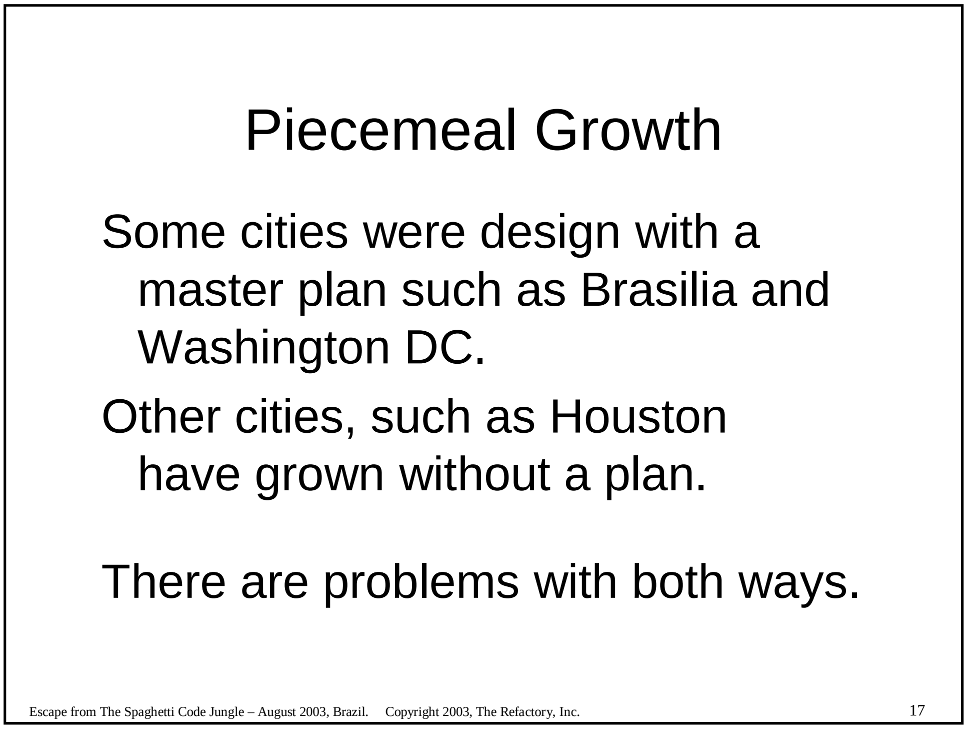
Piecemeal Growth

Iterative Incremental Development

Mir was designed to accommodate maintenance and growth

- **Core** *1986*
- **Kvant 1** *1987*
- **Kvant 2** *1989*
- **Kristall** *1990*
- **Spekter** *1995*
- **Docking** *1995*
- **Priroda** *1996*



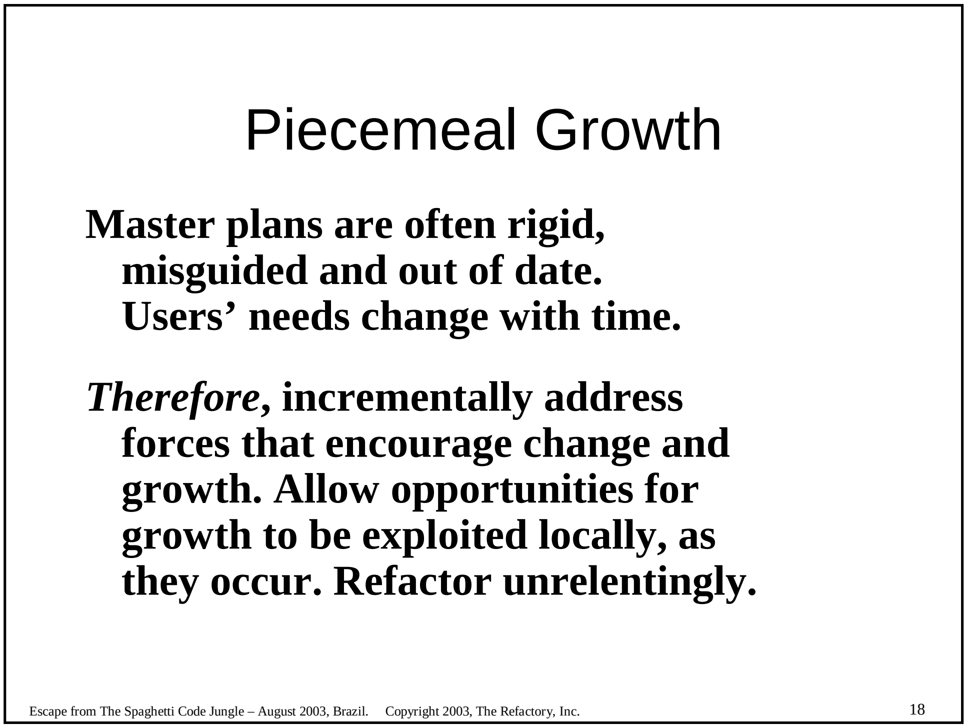


Piecemeal Growth

Some cities were design with a master plan such as Brasilia and Washington DC.

Other cities, such as Houston have grown without a plan.

There are problems with both ways.



Piecemeal Growth

Master plans are often rigid, misguided and out of date. Users’ needs change with time.

***Therefore*, incrementally address forces that encourage change and growth. Allow opportunities for growth to be exploited locally, as they occur. Refactor unrelentingly.**

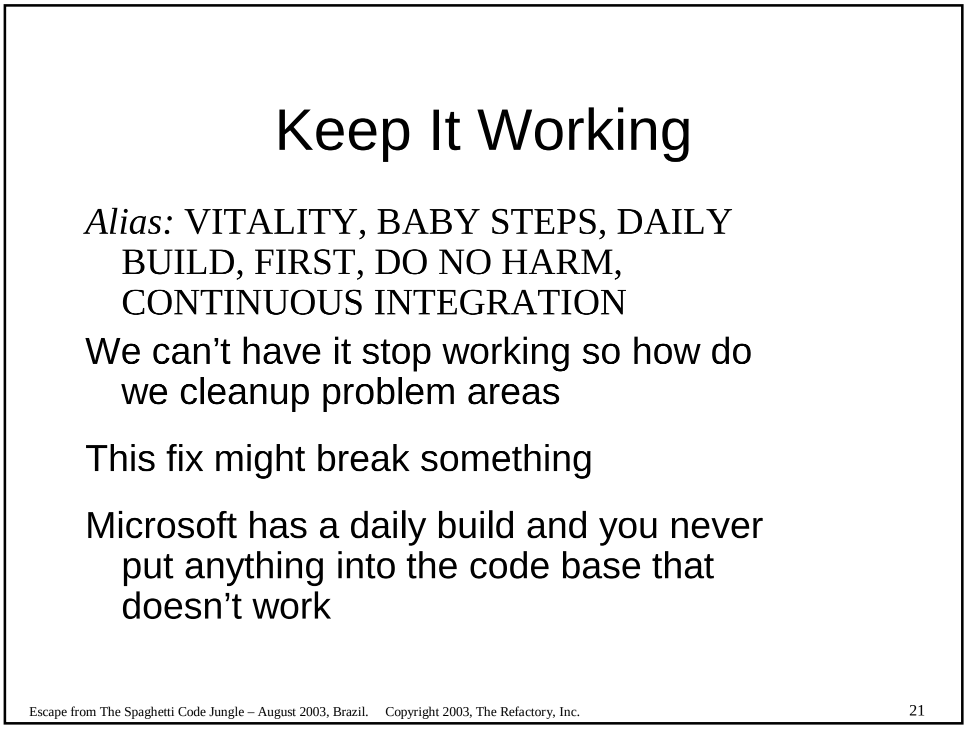
Piecemeal Growth Forces

Change: The fundamental problem with top-down design is that real world requirement are inevitably moving targets. You can't simply aspire to solve the problem at hand once and for all, because, by the time you're done, the problem will have changed out from underneath you.

Aesthetics: The goal of up-front design is to be able to discern and specify the significant architectural elements of a system before ground is broken for it.

Piecemeal Growth

When designers are faced with a choice between building something elegant from the ground up, or undermining the architecture of the existing system to quickly address a problem, architecture usually loses. Indeed, this is a natural phase in a system’s evolution [Foote & Opdyke 1995]. This might be thought of as *messy kitchen* phase, during which pieces of the system are scattered across the counter, awaiting an eventual cleanup. The danger is that the clean up is never done.



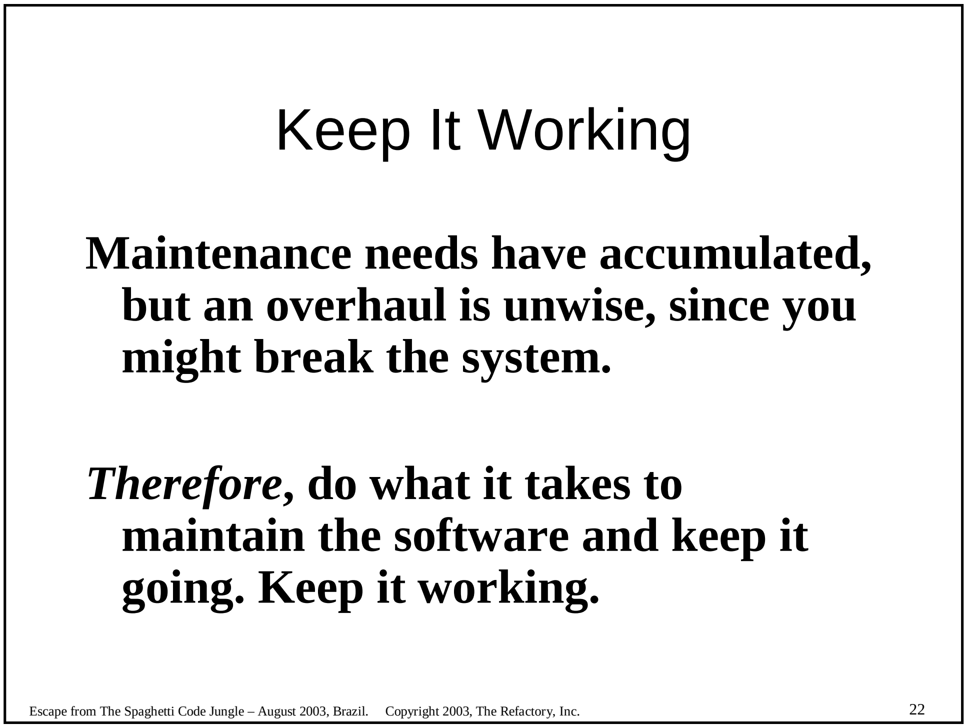
Keep It Working

Alias: VITALITY, BABY STEPS, DAILY BUILD, FIRST, DO NO HARM, CONTINUOUS INTEGRATION

We can’t have it stop working so how do we cleanup problem areas

This fix might break something

Microsoft has a daily build and you never put anything into the code base that doesn’t work



Keep It Working

Maintenance needs have accumulated, but an overhaul is unwise, since you might break the system.

***Therefore*, do what it takes to maintain the software and keep it going. Keep it working.**

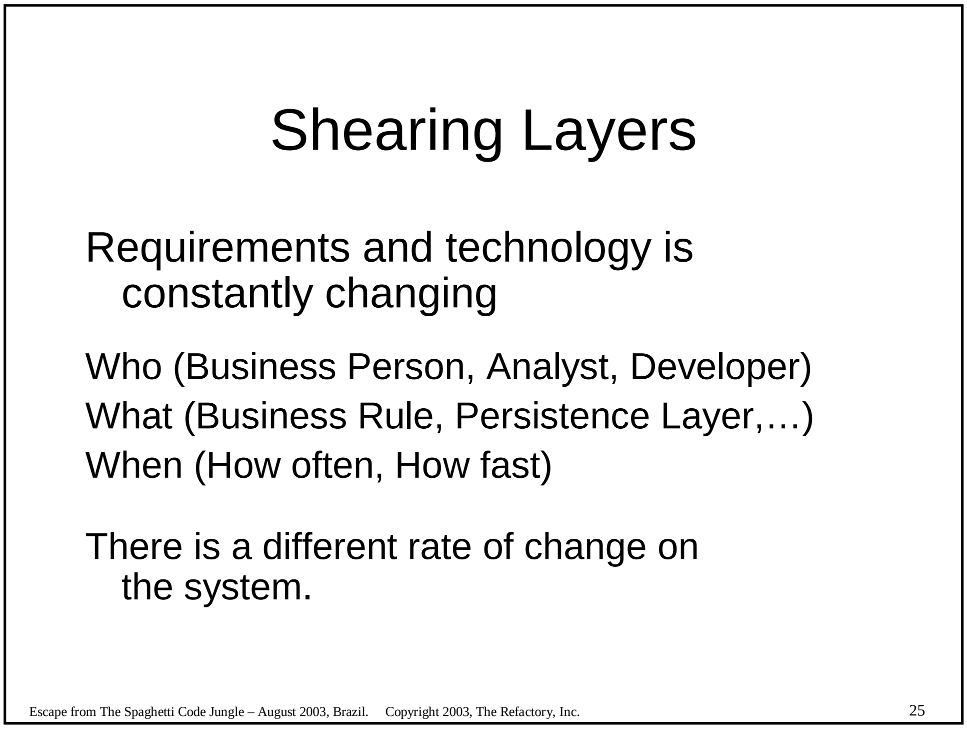
Keep It Working Forces

Workmanship: Architects who live in the house they are building have an obvious incentive to insure that things are done properly, since they will directly reap the consequences when they do not.

Dependability: These days, people rely on our software artifacts for their very livelihoods, and even, at time, for their very safety.

Keep It Working

Another vital factor in ensuring a system's continued vitality is a commitment to rigorous testing [Marick 1995][Bach 1994]. It's hard to keep a system working if you don't have a way of making sure it work. Testing is one of pillars of Extreme Programming. XP practices call for the development of unit tests before a single line of code is written.

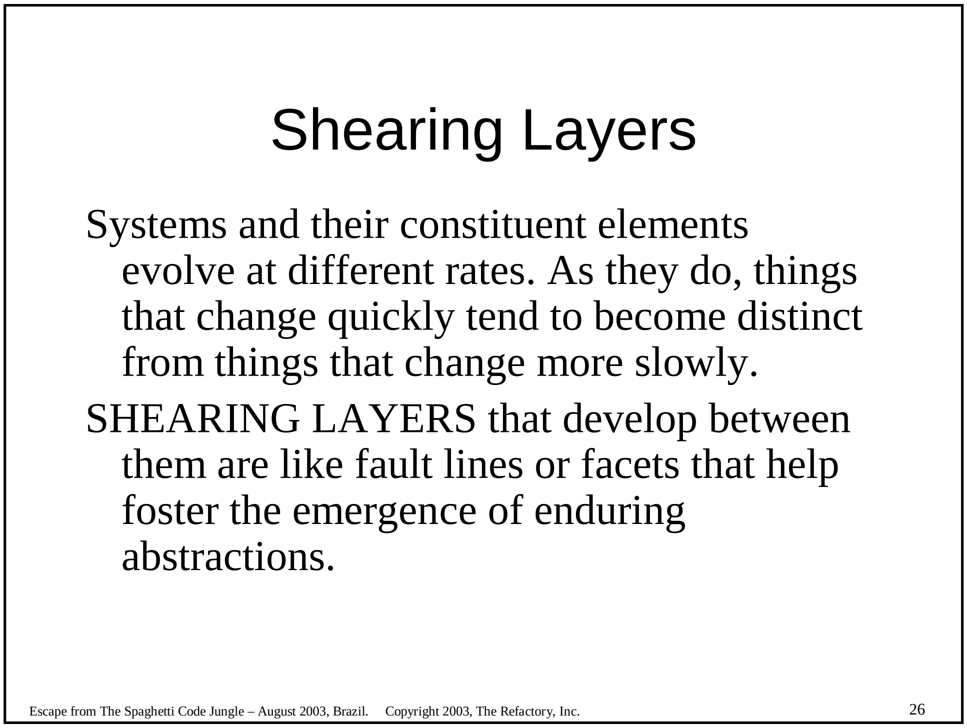


Shearing Layers

Requirements and technology is constantly changing

Who (Business Person, Analyst, Developer)
What (Business Rule, Persistence Layer,…)
When (How often, How fast)

There is a different rate of change on the system.



Shearing Layers

Systems and their constituent elements evolve at different rates. As they do, things that change quickly tend to become distinct from things that change more slowly.

SHEARING LAYERS that develop between them are like fault lines or facets that help foster the emergence of enduring abstractions.

Shearing Layers

Different artifacts change at different rates.

***Therefore*, factor your system so that artifacts that change at similar rates are together.**

Shearing Layers Forces

Adaptability and *Stability* are forces that are in constant tension. On one hand, systems must be able to confront novelty without blinking. On the other, they should not squander their patrimony on spur of the moment misadventures.

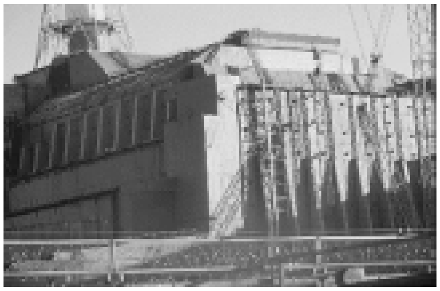
Shearing Layers
Accepting Changes

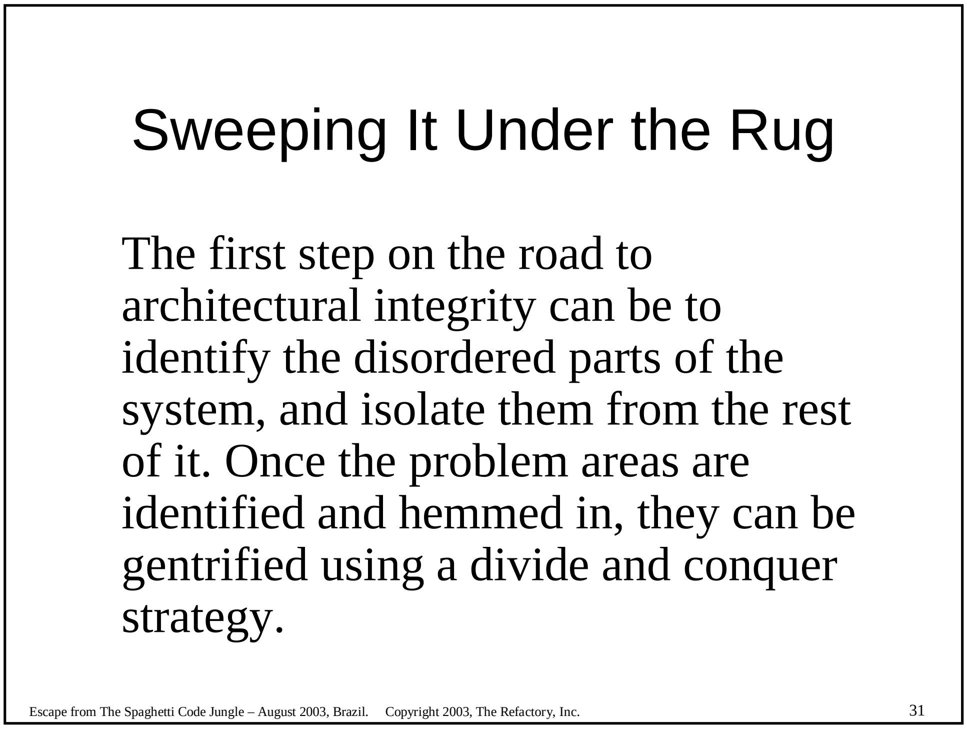
Part of the impetus behind using METADATA [Foote & Yoder 1998] is the observation that pushing complexity and power into the data pushes that same power (and complexity) out of the realm of the programmer and into the realm of users themselves. Metadata are often used to model static facilities such as classes and schemas, in order to allow them to change dynamically.

Sweeping It Under the Rug

Alias: Pretty Face, Façade, Housecleaning

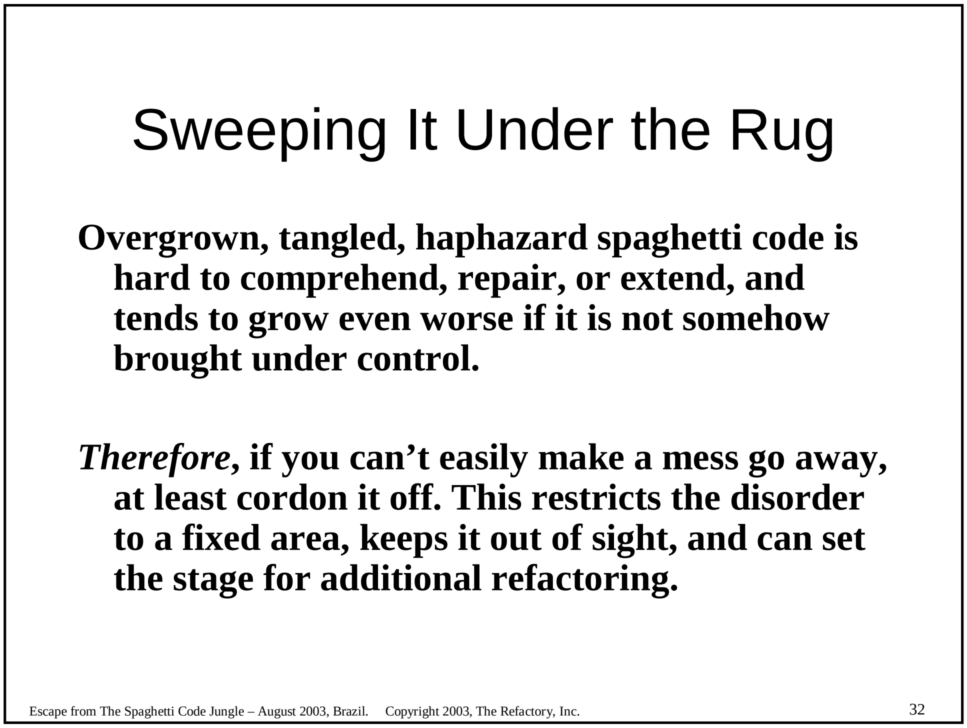
“You may not know how to get rid of a problem, but at least you can cordon it off…”





Sweeping It Under the Rug

The first step on the road to architectural integrity can be to identify the disordered parts of the system, and isolate them from the rest of it. Once the problem areas are identified and hemmed in, they can be gentrified using a divide and conquer strategy.



Sweeping It Under the Rug

Overgrown, tangled, haphazard spaghetti code is hard to comprehend, repair, or extend, and tends to grow even worse if it is not somehow brought under control.

Therefore**, if you can’t easily make a mess go away, at least cordon it off. This restricts the disorder to a fixed area, keeps it out of sight, and can set the stage for additional refactoring.**

Sweeping It Under the Rug “Additional Forces”

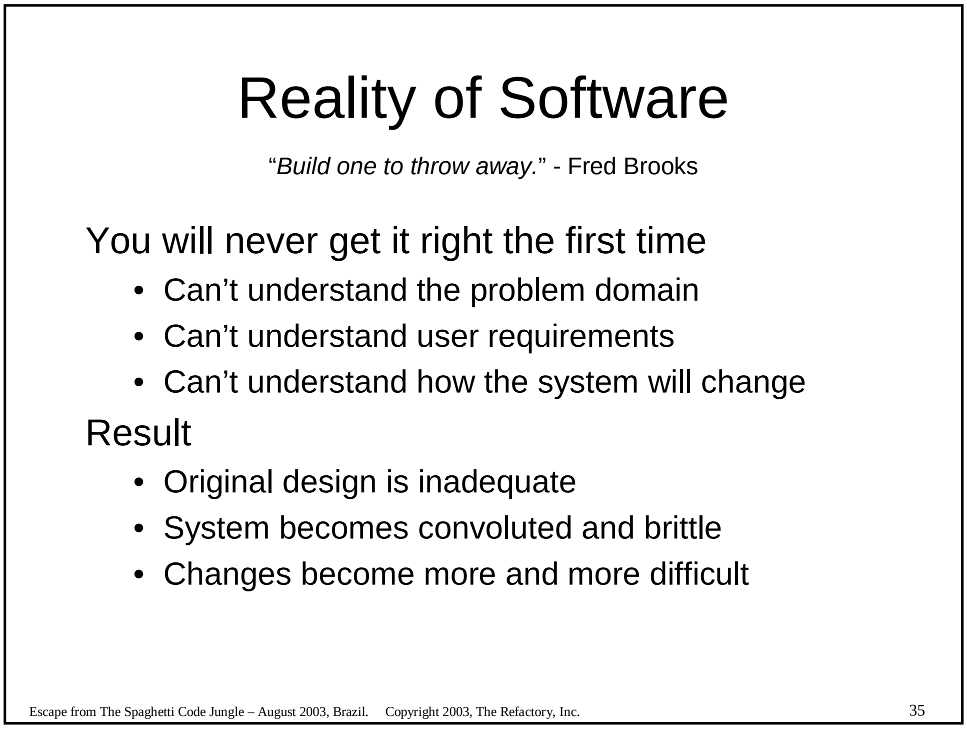
Comprehensibility: It should go without saying that comprehensible, attractive, well engineered code will be easier to maintain and extend than complicated, convoluted code.

Morale: Indeed, the price of life with a BIG BALL OF MUD goes beyond the bottom line. Life in the muddy trenches can be a dispiriting fate. Making even minor modifications can lead to maintenance marathons.

Sweeping It Under the Rug

Overgrown, tangled, haphazard spaghetti code is hard to comprehend, repair, or extend, and tends to grow even worse if it is not somehow brought under control.

***Therefore*, if you can’t easily make a mess go away, at least cordon it off. This restricts the disorder to a fixed area, keeps it out of sight, and can set the stage for additional refactoring.**



Reality of Software

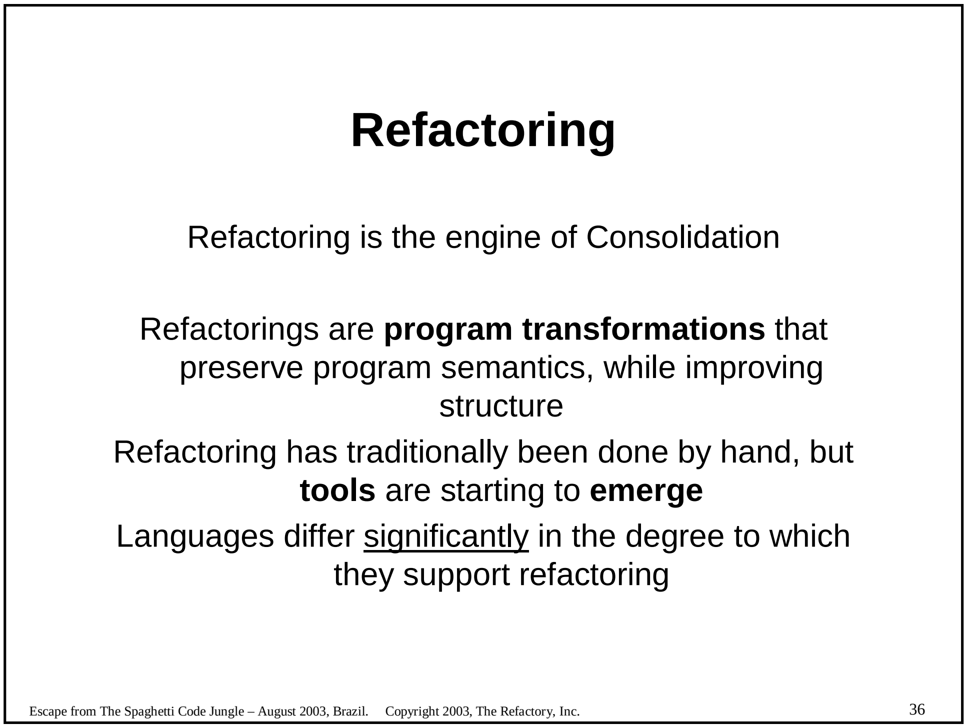
“*Build one to throw away.*” - Fred Brooks

You will never get it right the first time

- Can’t understand the problem domain
- Can’t understand user requirements
- Can’t understand how the system will change

Result

- Original design is inadequate
- System becomes convoluted and brittle
- Changes become more and more difficult



Refactoring

Refactoring is the engine of Consolidation

Refactorings are **program transformations** that preserve program semantics, while improving structure

Refactoring has traditionally been done by hand, but **tools** are starting to **emerge**

Languages differ significantly in the degree to which they support refactoring

Definition of Refactoring

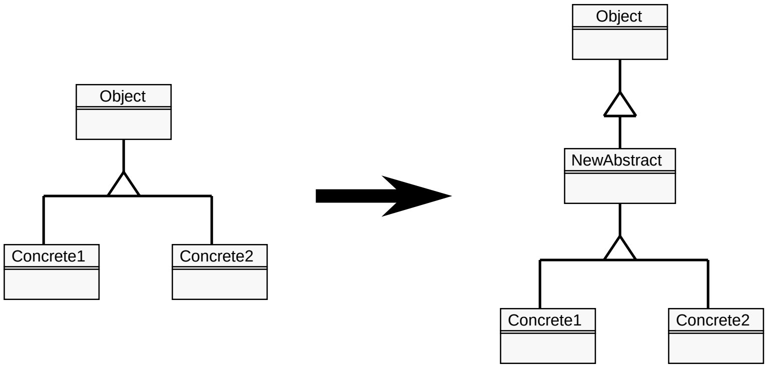
Interface design and functional factoring constitute the key intellectual content of software and are far more difficult to create or re-create than code. - Peter Deutsch

vt. - The process of redesigning the abstractions in a program

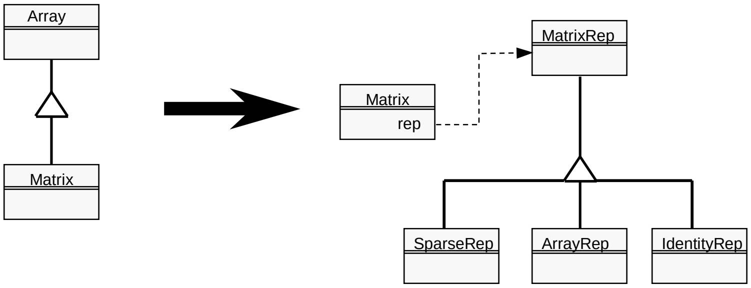
n. - A behavior-preserving source-to-source program transformation

A Simple Refactoring

Create Empty Class



A Complex Refactoring



Where Refactorings Come From

Application Maintenance
Application Extension
Application Development
Framework Development

“Refactoring often applies Design Patterns”

Can help clean up Big Balls of Mud!

Design Patterns

Design Patterns

Reconstruction

(Total Rewrite, Demolition)

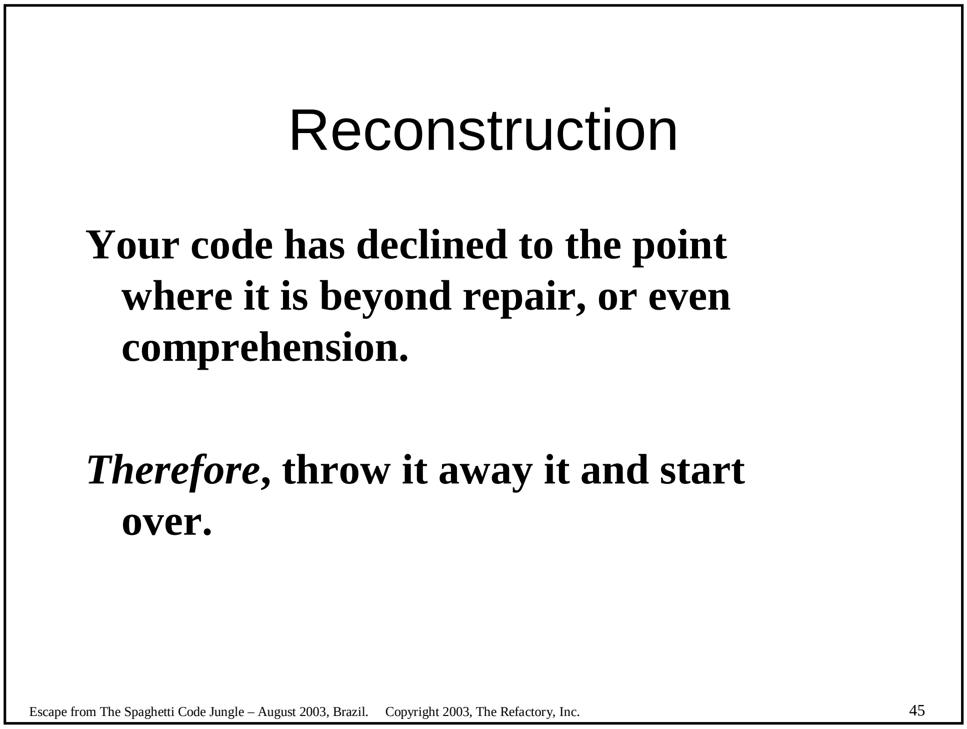
Atlanta’s Fulton County Stadium was built in 1966 and demolished in 1997.

Two single purpose stadia, with skyboxes, are replacing it

Like a set of dominoes, the former home of the Atlanta Braves collapsed Saturday (Courtesy WXIA)

Reconstruction

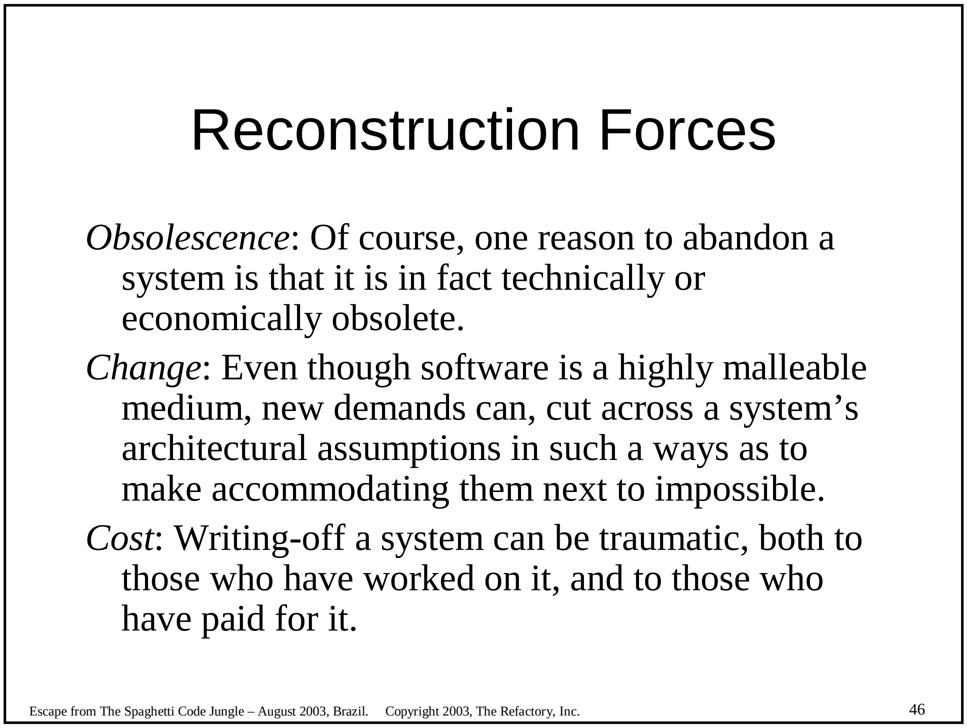
Extreme Programming [Beck 2000] had its genesis in the Chrysler Comprehensive Compensation project (C3). It began with a cry for help from a foundering project, and a decision to discard a year and a half's worth of work. The process they put in place after they started anew laid the foundation for XP, and the author's credit these approaches for the subsequent success of the C3 effort.



Reconstruction

Your code has declined to the point where it is beyond repair, or even comprehension.

***Therefore*, throw it away it and start over.**

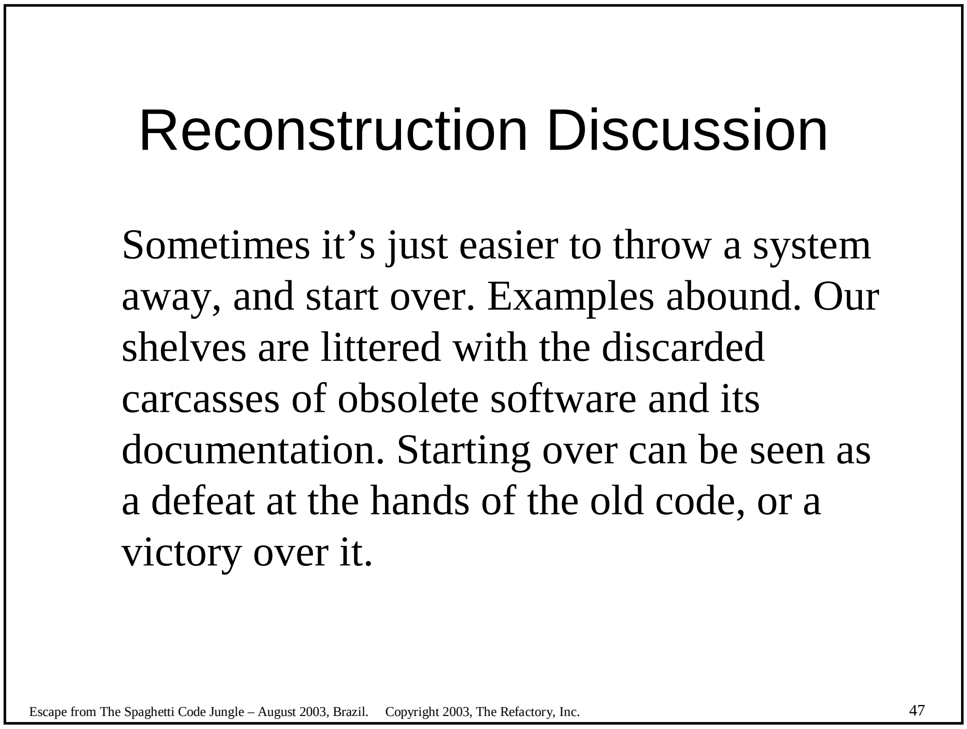


Reconstruction Forces

Obsolescence: Of course, one reason to abandon a system is that it is in fact technically or economically obsolete.

Change: Even though software is a highly malleable medium, new demands can, cut across a system’s architectural assumptions in such a ways as to make accommodating them next to impossible.

Cost: Writing-off a system can be traumatic, both to those who have worked on it, and to those who have paid for it.



Reconstruction Discussion

Sometimes it’s just easier to throw a system away, and start over. Examples abound. Our shelves are littered with the discarded carcasses of obsolete software and its documentation. Starting over can be seen as a defeat at the hands of the old code, or a victory over it.

Silver Buckshot

There are no silver bullets
…...Fred Brooks

- Objects
- Frameworks
- Patterns
- Architecture
- Process/Organization
- Tools

UIUC Patterns Group
Software Architecture Group
Ralph Johnson’s Group

- Objects
- Reuse
- Frameworks
- Adaptive Architecture
- Components
- Refactoring
- Evolution
- Patterns

Our Perspective

Objects, Patterns, Frameworks, and Refactoring really do work, and can lead to the production of better, more durable, more reusable code

To achieve this requires a commitment to tools, architecture, and software evolution, and to people with superior technical skills and domain insight

Adaptive Object-Models

Separates what changes from what doesn't.

Architectures that can dynamically adapt to new user requirements by storing descriptive (metadata) information about the business rules that are interpreted at runtime.

Sometimes called a "reflective architecture" or a "meta-architecture ".

Highly Flexible – Business people (non-programmers) can change it too.

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!

Bird on Patterns

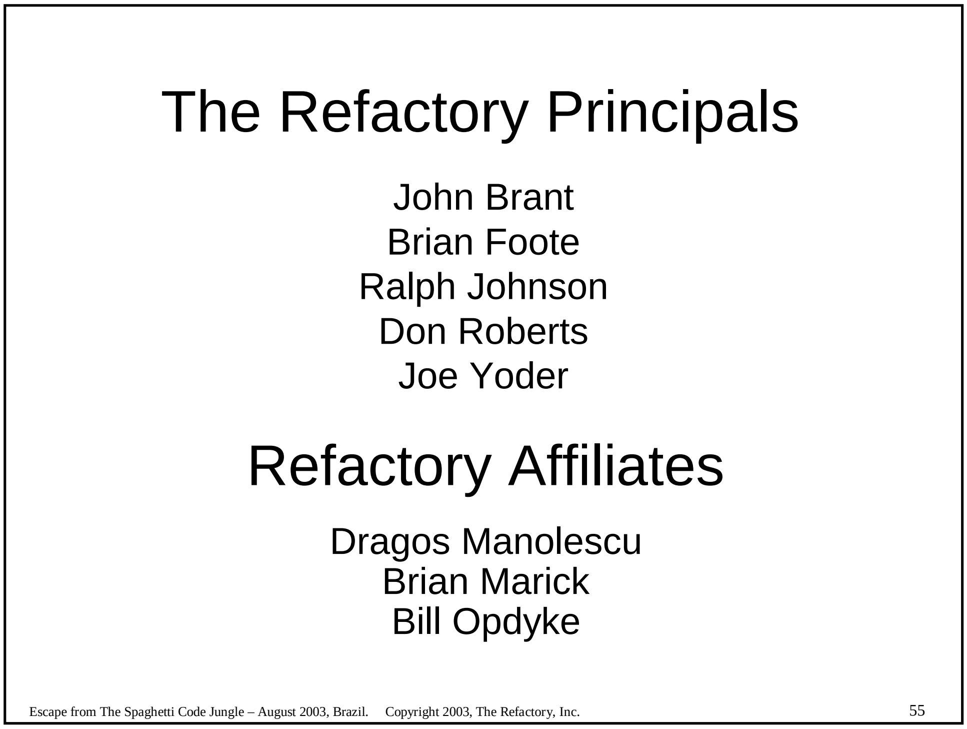
Learn the patterns and then forget ‘em

-- Charlie Parker

http://www.hillside.net

The Refactory, Inc.

The Refactory principles are experienced in software development, especially in object-oriented technology. We've been studying and developing software since 1973. Our current focus has been object-oriented technology, software architecture, and patterns. We have developed frameworks using Smalltalk, C++, and Java, have helped design several applications, and mentored many new Smalltalk, Java, and C++, C# developers. Highly experienced with Frameworks, Software Evolution, Refactoring, Objects, Flexible and Adaptable Systems (Adaptive Object-Models), Testing, Workflow Systems, and Agile Software Development including methods like eXtreme Programming (XP).

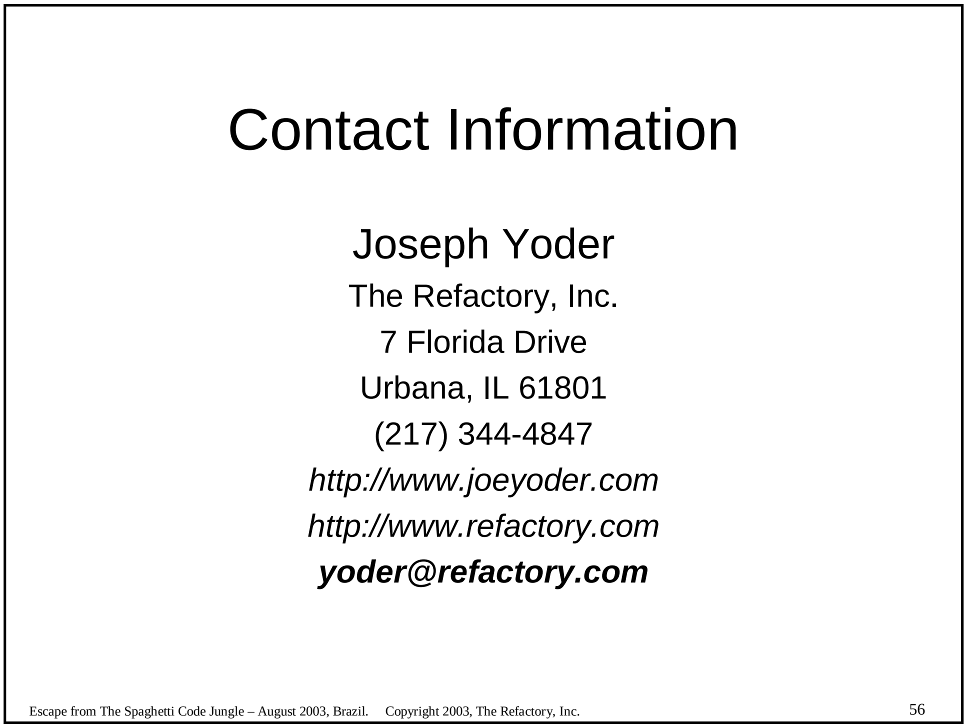


The Refactory Principals

John Brant
Brian Foote
Ralph Johnson
Don Roberts
Joe Yoder

Refactory Affiliates

Dragos Manolescu
Brian Marick
Bill Opdyke



Contact Information

Joseph Yoder
The Refactory, Inc.
7 Florida Drive
Urbana, IL 61801
(217) 344-4847
http://www.joeyoder.com
http://www.refactory.com
yoder@refactory.com