Big Ball of Mud

Brian Foote and Joseph Yoder

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield
Urbana, IL 61801 USA

foote@cs.uiuc.edu (217) 328-3523
yoder@cs.uiuc.edu (217) 244-4695

Saturday, June 26, 1999
Fourth Conference on Patterns Languages of Programs (PLoP '97/EuroPLoP '97)
Monticello, Illinois, September 1997
Technical Report #WUCS-97-34 (PLoP '97/EuroPLoP '97), September 1997
Department of Computer Science, Washington University
Chapter 29
Pattern Languages of Program Design 4
edited by Neil Harrison, Brian Foote, and Hans Rohnert
Addison-Wesley, 2000

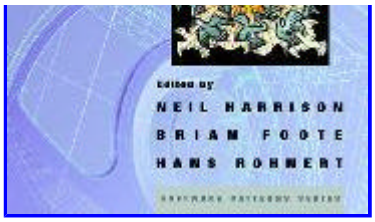
This volume is part of the Addison-Wesley Software Patterns Series.

This paper is also available in the following formats:
[PDF] [Word] [RTF] [PostScript]

Also by **Brian Foote** and **Joseph Yoder**
Architecture, Evolution, and Metamorphosis
The Selfish Class

This paper was recently featured in **Slashdot**

Contents



Abstract

While much attention has been focused on high-level software architectural patterns, what is, in effect, the de-facto standard software architecture is seldom discussed. This paper examines this most frequently deployed of software architectures: the BIG BALL OF MUD. A BIG BALL OF MUD is a casually, even haphazardly, structured system. Its organization, if one can call it that, is dictated more by expediency than design. Yet, its enduring popularity cannot merely be indicative of a general disregard for architecture.

These patterns explore the forces that encourage the emergence of a BIG BALL OF MUD, and the undeniable effectiveness of this approach to software architecture. What are the people who build them doing right? If more high-minded architectural approaches are to compete, we must understand what the forces that lead to a BIG BALL OF MUD are, and examine alternative ways to resolve them.

A number of additional patterns emerge out of the BIG BALL OF MUD. We discuss them in turn. Two principal questions underlie these patterns: Why are so many existing systems architecturally undistinguished, and what can we do to improve them?

Introduction

Over the last several years, a number of authors [Garlan & Shaw 1993] [Shaw 1996] [Buschmann et. al. 1996] [Meszaros 1997] have presented patterns that characterize high-level software architectures, such as PIPELINE and LAYERED ARCHITECTURE. In an ideal world, every system would be an exemplar of one or more such high-level patterns. Yet, this is not so. The architecture that actually predominates in practice has yet to be discussed: the BIG BALL OF MUD.

A BIG BALL OF MUD is haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle. We've all seen them. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. The overall structure of the system may never have been well defined. If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps,

are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems.

Still, this approach endures and thrives. Why is this architecture so popular? Is it as bad as it seems, or might it serve as a way-station on the road to more enduring, elegant artifacts? What forces drive good programmers to build ugly systems? Can we avoid this? Should we? How can we make such systems better?

We present the following seven patterns:

- BIG BALL OF MUD
- THROWAWAY CODE
- PIECEMEAL GROWTH
- KEEP IT WORKING
- SHEARING LAYERS
- SWEEPING IT UNDER THE RUG
- RECONSTRUCTION

Why does a system become a BIG BALL OF MUD? Sometimes, big, ugly systems emerge from THROWAWAY CODE. THROWAWAY CODE is quick-and-dirty code that was intended to be used only once and then discarded. However, such code often takes on a life of its own, despite casual structure and poor or non-existent documentation. It works, so why fix it? When a related problem arises, the quickest way to address it might be to expediently modify this working code, rather than design a proper, general program from the ground up. Over time, a simple throwaway program begets a BIG BALL OF MUD.

Even systems with well-defined architectures are prone to structural erosion. The relentless onslaught of changing requirements that any successful system attracts can gradually undermine its structure. Systems that were once tidy become overgrown as PIECEMEAL GROWTH gradually allows elements of the system to sprawl in an uncontrolled fashion.

If such sprawl continues unabated, the structure of the system can become so badly compromised that it must be abandoned. As with a decaying neighborhood, a downward spiral ensues. Since the system becomes harder and harder to understand, maintenance becomes more expensive, and more difficult. Good programmers refuse to work there. Investors withdraw their capital. And yet, as with neighborhoods, there are ways to avoid, and even reverse, this sort of decline. As with anything else in the universe, counteracting entropic forces requires an investment of energy. Software gentrification is no exception. The way to arrest entropy in software is to refactor it. A sustained commitment to refactoring can keep a system from subsiding into a BIG BALL OF MUD.

A major flood, fire, or war may require that a city be evacuated and rebuilt from the ground up. More often, change takes place a building or block at a time, while the city as a whole continues to function. Once established, a strategy of KEEPING IT WORKING preserves a municipality's vitality as it grows.

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

A simple way to begin to control decline is to cordon off the blighted areas, and put an attractive façade around them. We call this strategy SWEEPING IT UNDER THE RUG. In more advanced cases, there may be no alternative but to tear everything down and start over. When total RECONSTRUCTION becomes necessary, all that is left to salvage is the patterns that underlie the experience.

Some of these patterns might appear at first to be antipatterns [Brown et al. 1998] or straw men, but they are not, at least in the customary sense. Instead, they seek to examine the gap between what we preach and what we practice.

Still, some of them may strike some readers as having a schizoid quality about them. So, for the record, let us put our cards on the table. We are in favor of good architecture.

Our ultimate agenda is to help drain these swamps. Where possible, architectural decline should be prevented, arrested, or reversed. We discuss ways of doing this. In severe cases, architectural abominations may even need to be demolished.

At the same time, we seek not to cast blame upon those who must wallow in these mires. In part, our attitude is to "hate the sin, but love the sinner". But, it goes beyond this. Not every backyard storage shack needs marble columns. There are significant forces that can conspire to compel architecture to take a back seat to functionality, particularly early in the evolution of a software artifact. Opportunities and insights that can allow for architectural progress often are present later rather than earlier in the lifecycle.

A certain amount of controlled chaos is natural during construction, and can be tolerated, as long as you clean up after yourself eventually. Even beyond this though, a complex system may be an accurate reflection of our immature understanding of a complex problem. The class of systems that we can build at all may be larger than the class of systems we can build elegantly, at least at first. A somewhat ramshackle rat's nest might be a state-of-the-art architecture for a poorly understood domain. This should not be the end of the story, though. As we gain more experience in such domains, we should increasingly direct our energies to gleaning more enduring architectural abstractions from them.

The patterns described herein are not intended to stand alone. They are instead set in a context that includes a number of other patterns that we and others have described. In particular, they are set in contrast to the lifecycle patterns, PROTOTYPE PHASE, EXPANSIONARY PHASE, and CONSOLIDATION PHASE, presented in [Foote & Opdyke 1995] and [Coplien 1995], the SOFTWARE TECTONICS pattern in [Foote & Yoder 1996], and the framework development patterns in [Roberts & Johnson 1998].

Indeed, to a substantial extent, much of this chapter describes the disease, while the patterns above describe what we believe can be the cure: a flexible, adaptive, feedback-driven development process in which design and refactoring pervade the lifecycle of each artifact, component, and framework, within and beyond the applications that incubate them.

Forces

A number of forces can conspire to drive even the most architecturally conscientious organizations to produce BIG BALLS OF MUD. These pervasive, *"global" forces* are at work in all the patterns presented. Among these forces:

Time: There may not be enough time to consider the long-term architectural implications of one's design and implementation decisions. Even when systems have been well designed, architectural concerns often must yield to more pragmatic ones as a deadline starts to loom.

One reason that software architectures are so often mediocre is that architecture frequently takes a back seat to more mundane concerns such as cost, time-to-market, and programmer skill. Architecture is often seen as a luxury or a frill, or the indulgent pursuit of lily-gilding compulsives who have no concern for the bottom line. Architecture is often treated with neglect, and even disdain. While such attitudes are unfortunate, they are not hard to understand. Architecture is a long-term concern. The concerns above have to be addressed if a product is not to be stillborn in the marketplace, while the benefits of good architecture are realized later in the lifecycle, as frameworks mature, and reusable black-box components emerge [Foote & Opdyke 1995].

Architecture can be looked upon as a *Risk*, that will consume resources better directed at meeting a fleeting market window, or as an *Opportunity* to lay the groundwork for a commanding advantage down the road.

Indeed, an immature architecture can be an advantage in a growing system because data and functionality can migrate to their natural places in the system unencumbered by artificial architectural constraints. Premature architecture can be more dangerous than none at all, as unproved architectural hypotheses turn into straightjackets that discourage evolution and experimentation.

Cost: Architecture is expensive, especially when a new domain is being explored. Getting the system right seems like a pointless luxury once the system is limping well enough to ship. An investment in architecture usually does not pay off immediately. Indeed, if architectural concerns delay a product's market entry for too long, then long-term concerns may be moot. Who benefits from an investment in architecture, and when is a return on this investment seen? Money spent on a quick-and-dirty project that allows an immediate entry into the market may be better spent than money spent on elaborate, speculative architectural fishing expedition. It's hard to recover the value of your architectural assets if you've long since gone bankrupt.

Programmers with the ability to discern and design quality architectures are reputed to command a premium. These expenses must be weighed against those of allowing an expensive system to slip into premature decline and obsolescence. If you think good architecture is expensive, try bad architecture.

Experience: Even when one has the time and inclination to take architectural concerns into account,

one's experience, or lack thereof, with the domain can limit the degree of architectural sophistication that can be brought to a system, particularly early in its evolution. Some programmers flourish in environments where they can discover and develop new abstractions, while others are more comfortable in more constrained environments (for instance, Smalltalk vs. Visual Basic programmers.) Often, initial versions of a system are vehicles whereby programmers learn what pieces must be brought into play to solve a particular problem. Only after these are identified do the architectural boundaries among parts of the system start to emerge.

Inexperience can take a number of guises. There is absolute, fresh out of school inexperience. A good architect may lack domain experience, or a domain expert who knows the code cold may not have architectural experience.

Employee turnover can wreak havoc on an organization's institutional memory, with the perhaps dubious consolation of bringing fresh blood aboard.

Skill: Programmers differ in their levels of skill, as well as in expertise, predisposition and temperament. Some programmers have a passion for finding good abstractions, while some are skilled at navigating the swamps of complex code left to them by others. Programmers differ tremendously in their degrees of experience with particular domains, and their capacities for adapting to new ones. Programmers differ in their language and tool preferences and experience as well.

Visibility: Buildings are tangible, physical structures. You can look at a building. You can watch it being built. You can walk inside it, and admire and critique its design.

A program's user interface presents the public face of a program, much as a building's exterior manifests its architecture. However, unlike buildings, only the people who build a program see how it looks inside.

Programs are made of bits. The manner in which we present these bits greatly affects our sense of how they are put together. Some designers prefer to see systems depicted using modeling languages or PowerPoint pictures. Others prefer prose descriptions. Still others prefer to see code. The fashion in which we present our architectures affects our perceptions of whether they are good or bad, clear or muddled, and elegant or muddy.

Indeed, one of the reasons that architecture is neglected is that much of it is "under the hood", where nobody can see it. If the system works, and it can be shipped, who cares what it looks like on the inside?

Complexity: One reason for a muddled architecture is that software often reflects the inherent complexity of the application domain. This is what Brooks called "essential complexity" [Brooks 1995]. In other words, the software is ugly because the problem is ugly, or at least not well understood. Frequently, the organization of the system reflects the sprawl and history of the organization that built it (as per CONWAY'S LAW [Coplien 1995]) and the compromises that were made along the way. Renegotiating these relationships is often difficult once the basic boundaries among system elements are drawn. These relationships can take on the immutable character of "site" boundaries that Brand [Brand 1994] observed in real cities. Big problems can arises when the needs of the applications force unrestrained communication across these boundaries. The system becomes a tangled mess, and what little structure is there can erode further.

Change: Architecture is a hypothesis about the future that holds that subsequent change will be confined to that part of the design space encompassed by that architecture. Of course, the world has a way of mocking our attempts to make such predictions by tossing us the totally unexpected. A problem we might have been told was definitely ruled out of consideration for all time may turn out to be dear to the heart of a new client we never thought we'd have. Such changes may cut directly across the grain of fundamental architectural decisions made in the light of the certainty that these new contingencies could never arise. The "right" thing to do might be to redesign the system. The more likely result is that the architecture of the system will be expediently perturbed to address the new requirements, with only passing regard for the effect of these radical changes on the structure of the system.

Scale: Managing a large project is a qualitatively different problem from managing a small one, just as leading a division of infantry into battle is different from commanding a small special forces team. Obviously, "divide and conquer" is, in general, an insufficient answer to the problems posed by scale. Alan Kay, during an invited talk at OOPSLA '86 observed that "good ideas don't always scale." That observation prompted Henry Lieberman to inquire "so what do we do, just scale the bad ones?"

BIG BALL OF MUD

alias
SHANTYTOWN
SPAGHETTI CODE

Shantytowns are squalid, sprawling slums. Everyone seems to agree they are a bad idea, but forces conspire to promote their emergence anyway. What is it that they are doing right?

Shantytowns are usually built from common, inexpensive materials and simple tools. Shantytowns can be built using relatively unskilled labor. Even though the labor force is "unskilled" in the customary

sense, the construction and maintenance of this sort of housing can be quite labor intensive. There is little specialization. Each housing unit is constructed and maintained primarily by its inhabitants, and each inhabitant must be a jack of all the necessary trades. There is little concern for infrastructure, since infrastructure requires coordination and capital, and specialized resources, equipment, and skills. There is little overall planning or regulation of growth. Shantytowns emerge where there is a need for housing, a surplus of unskilled labor, and a dearth of capital investment. Shantytowns fulfill an immediate, local need for housing by bringing available resources to bear on the problem. Loftier architectural goals are a luxury that has to wait.

Maintaining a shantytown is labor-intensive and requires a broad range of skills. One must be able to improvise repairs with the materials on-hand, and master tasks from roof repair to ad hoc sanitation. However, there is little of the sort of skilled specialization that one sees in a mature economy.

All too many of our software systems are, architecturally, little more than shantytowns. Investment in tools and infrastructure is too often inadequate. Tools are usually primitive, and infrastructure such as libraries and frameworks, is undercapitalized. Individual portions of the system grow unchecked, and the lack of infrastructure and architecture allows problems in one part of the system to erode and pollute adjacent portions. Deadlines loom like monsoons, and architectural elegance seems unattainable.

❖ ❖ ❖

As a system nears completion, its actual users may begin to work with it for the first time. This experience may inspire changes to data formats and the user interface that undermine architectural decisions that had been thought to be settled. Also, as Brooks [Brooks 1995] has noted, because software is so flexible, it is often asked to bear the burden of architectural compromises late in the development cycle of hardware/software deliverables precisely because of its flexibility.

This phenomenon is not unique to software. Stewart Brand [Brand 1994] has observed that the period just prior to a building's initial occupancy can be a stressful period for both architects and their clients. The money is running out, and the finishing touches are being put on just those parts of the space that will interact the most with its occupants. During this period, it can become evident that certain wish-list items are not going to make it, and that exotic experiments are not going to work. Compromise becomes the "order of the day".

The time and money to chase perfection are seldom available, nor should they be. To survive, we must do what it takes to get our software working and out the door on time. Indeed, if a team completes a project with time to spare, today's managers are likely to take that as a sign to provide less time and money or fewer people the next time around.

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

Cost: Architecture is a long-term investment. It is easy for the people who are paying the bills to dismiss it, unless there is some tangible immediate benefit, such a tax write-off, or unless surplus money and time happens to be available. Such is seldom the case. More often, the customer needs something working by tomorrow. Often, the people who control and manage the development process simply do not regard architecture as a pressing concern. If programmers know that workmanship is invisible, and managers don't want to pay for it anyway, a vicious circle is born.

Skill: Ralph Johnson is fond of observing that is inevitable that "on average, average organizations will have average people". One reason for the popularity and success of BIG BALL OF MUD approaches might be that this appoach doesn't require a hyperproductive virtuoso architect at every keyboard.

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

It may seem to a programmer that whether to don hip boots and wade into a swamp is a major quality-of-life matter, but programmer comfort is but one concern to a manager, which can conflict with many others. Architecture and code quality may strike management as frills that have only an indirect impact on their bottom lines.

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

The case made here resembles Gabriel's "Worse is Better" arguments [Gabriel 1991] in a number of respects. 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. There are those who might construe this as a sort of blackboard approach [Buschmann 1996], but it more closely resembles a grab bag of undifferentiated state. Where state information is compartmentalized, it may be passed promiscuously about though Byzantine back channels that circumvent the system's original structure.

Variable and function names might be uninformative, or even misleading. Functions themselves may make extensive use of global variables, as well as long lists of poorly defined parameters. The function themselves are lengthy and convoluted, and perform several unrelated tasks. Code is duplicated. The flow of control is hard to understand, and difficult to follow. The programmer's intent is next to impossible to discern. The code is simply unreadable, and borders on indecipherable. The code exhibits the unmistakable signs of patch after patch at the hands of multiple maintainers, each of whom barely understood the consequences of what he or she was doing. Did we mention documentation? What documentation?

BIG BALL OF MUD might be thought of as an anti-pattern, since our intention is to show how passivity in the face of forces that undermine architecture can lead to a quagmire. However, its undeniable popularity leads to the inexorable conclusion that it is a pattern in its own right. It is certainly a pervasive, recurring solution to the problem of producing a working system in the context of software development. It would seem to be the path of least resistance when one confronts the sorts of forces discussed above. Only by understanding the logic of its appeal can we channel or counteract the forces that lead to a BIG BALL OF MUD.

One thing that isn't the answer is rigid, totalitarian, top-down design. Some analysts, designers, and architects have an exaggerated sense of their ability to get things right up-front, before moving into implementation. This approach leads to inefficient resources utilization, analysis paralysis, and design straightjackets and cul-de-sacs.

Kent Beck has observed that the way to build software is to: Make it work. Make it right. Make it fast [Beck 1997]. "Make it work" means that we should focus on functionality up-front, and get something running. "Make it right" means that we should concern ourselves with how to structure the system only after we've figured out the pieces we need to solve the problem in the first place. "Make it fast" means that we should be concerned about optimizing performance only after we've learned how to solve the problem, and after we've discerned an architecture to elegantly encompass this functionality. Once all this has been done, one can consider how to make it cheap.

When it comes to software architecture, form follows function. Here we mean "follows" not in the traditional sense of dictating function. Instead, we mean that the distinct identities of the system's architectural elements often don't start to emerge until after the code is working.

Domain experience is an essential ingredient in any framework design effort. It is hard to try to follow a front-loaded, top-down design process under the best of circumstances. Without knowing the architectural demands of the domain, such an attempt is premature, if not foolhardy. Often, the only way to get domain experience early in the lifecycle is to hire someone who has worked in a domain before from someone else.

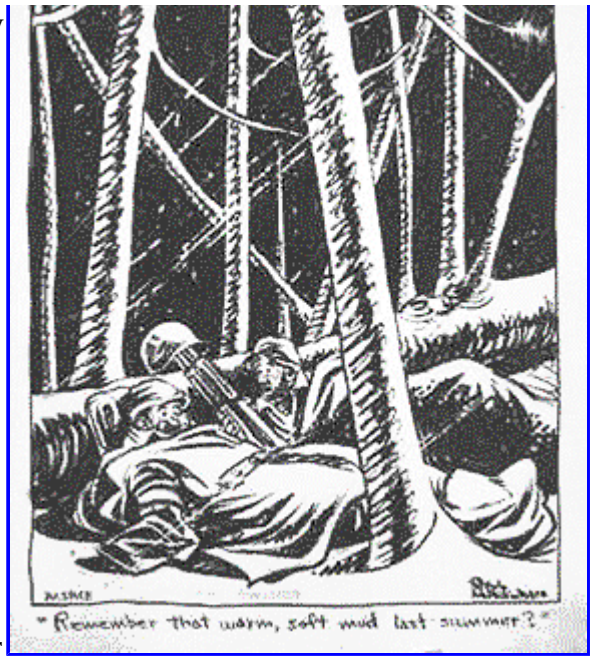
The quality of one's tools can influence a system's architecture. If a system's architectural goals are inadequately communicated among members of a team, they will be harder to take into account as the system is designed and constructed.

Finally, engineers will differ in their levels of skill and commitment to architecture. Sadly, architecture has been undervalued for so long that many engineers regard life with a BIG BALL OF MUD as normal. Indeed some engineers are particularly skilled at learning to navigate these quagmires, and guiding others through them. Over time, this symbiosis between architecture and skills can change the character of the organization itself, as swamp guides become more valuable than architects. 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 1998a] 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. In a land devoid of landmarks, such guides may become indispensable.

The incentives that drive the evolution of such systems can, at times, operate perversely. Just as it is easier to be verbose than concise, it is easier to build complex systems than it is to build simple ones. Skilled programmers may be able to create complexity more quickly than their peers, and more quickly than they can document and explain it. Like an army outrunning its logistics train, complexity increases until it reaches the point where such programmers can no longer reliably cope with it.

This is akin to a phenonmenon dubbed the *PeterPrinciple of Programming* by authors on the Wiki-Wiki web [Cunninghan 1999a]. Complexity increases rapidly until the it reaches a level of complexity just beyond that with which programmers can comfortably cope. At this point, complexity and our abilities to contain it reach an uneasy equilibrium. The blitzkrieg bogs down into a siege. We built the most complicated system that can possible work [Cunningham 1999b].

Such code can become a personal fiefdom, since the author care barely understand it anymore, and no one

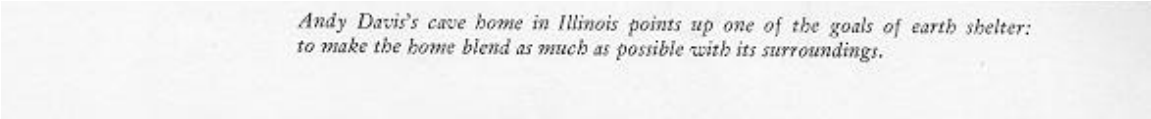


else can come close. Once simple repairs become all day affairs, as the code turns to mud. It becomes increasingly difficult for management to tell how long such repairs ought to take. Simple objectives turn into trench warfare. Everyone becomes resigned to a turgid pace. Some even come to prefer it, hiding in their cozy foxholes, and making their two line-per-day repairs.

It is interesting to ask whether some of the differences in productivity seen between hyper-productive organizations and typical shops are due not to differences in talent, but differences in terrain. Mud is hard to march through. The hacker in the trenches must engage complexity in hand-to-hand combat every day. Sometimes, complexity wins.

Status in the programmer's primate pecking order is often earned through ritual displays of cleverness, rather than through workman-like displays of simplicity and clarity. That which a culture glorifies will flourish.

Yet, a case can be made that the casual, undifferentiated structure of a BIG BALL OF MUD is one of its secret advantages, since forces acting between two parts of the system can be directly addressed without having to worry about undermining the system's grander architectural aspirations. These aspirations are modest ones at best in the typical BIG BALL OF MUD. Indeed, a casual approach to architecture is emblematic of the early phases of a system's evolution, as programmers, architects and users learn their way around the domain [Foote & Opdyke 1995]. During the PROTOTYPE and EXPANSIONARY PHASES of a systems evolution, expedient, white-box inheritance-based code borrowing, and a relaxed approach to encapsulation are common. Later, as experience with the system accrues, the grain of the architectural domain becomes discernable, and more durable black-box components begin to emerge. In other words, it's okay if the system looks at first like a BIG BALL OF MUD, at least until you know better.



Andy Davis's cave home in Illinois points up one of the goals of earth shelter: to make the home blend as much as possible with its surroundings.

❖ ❖ ❖

Brian Marick first suggested the name "BIG BALL OF MUD" as a name for these sort of architectures, and the observation that this was, perhaps, the dominant architecture currently deployed, during a meeting of the University of Illinois Software Architecture Group several years ago. We have been using the term ever since. The term itself, in turn, appears to have arisen during the '70s as a characterization of Lisp.

BIG BALL OF MUD architectures often emerge from throw-away prototypes, or THROWAWAY CODE, because the prototype is kept, or the disposable code is never disposed of. (One might call these "little balls of mud".)

They also can emerge as gradual maintenance and PIECEMEAL GROWTH impinges upon the structure of a mature system. Once a system is working, a good way to encourage its growth is to KEEP IT WORKING. When the SHEARING LAYERS that emerge as change drives the system's evolution run against the existing grain of the system, its structure can be undermined, and the result can be a BIG BALL OF MUD.

The PROTOTYPE PHASE and EXPANSION PHASE patterns in [Foote & Opdyke 1995] both emphasize that a period of exploration and experimentation is often beneficial before making enduring architectural commitments.

However, these activities, which can undermine a system's structure should be interspersed with CONSOLIDATION PHASES [Foote & Opdyke 1995], during which opportunities to refactor the system to enhance its structure are exploited. Proponents of Extreme Programming [Beck 2000] also emphasize continuous coding and refactoring.

[Brand 1994] observes that buildings with large spaces punctuated with regular columns had the paradoxical effect of encouraging the innovative reuse of space precisely because they *constrained* the design space. Grandiose flights of architectural fancy weren't possible, which reduced the number of design alternatives that could be put on the table. Sometimes FREEDOM FROM CHOICE [Foote 1988] is what we really want.

One of mud's most effective enemies is sunshine. Subjecting convoluted code to scrutiny can set the stage for its refactoring, repair, and rehabilitation. Code reviews are one mechanism one can use to expose code to daylight.

Another is the Extreme Programming practice of pair programming [Beck 2000]. A pure pair programming approach requires that every line of code written be added to the system with two programmers present. One types, or "drives", while the other "rides shotgun" and looks on. In contrast to traditional solitary software production practices, pair programming subjects code to immediate scrutiny, and provides a means by which knowledge about the system is rapidly disseminated.

Indeed, reviews and pair programming provide programmers with something their work would not otherwise have: an audience. Sunlight, it is said is a powerful disinfectant. Pair-practices add an

element of performance to programming. An immediate audience of one's peers provides immediate incentives to programmers to keep their code clear and comprehensible, as well as functional.

An additional benefit of pairing is that accumulated wisdom and best practices can be rapidly disseminated throughout an organization through successive pairings. This is, incidentally, the same benefit that sexual reproduction brought to the genome.

By contrast, if no one ever looks at code, everyone is free to think they are better than average at producing it. Programmers will, instead, respond to those relatively perverse incentives that do exist. Line of code metrics, design documents, and other indirect measurements of progress and quality can become central concerns.

There are three ways to deal with BIG BALLS OF MUD. The first is to keep the system healthy. Conscientiously alternating periods of EXPANSION with periods of CONSOLIDATION, refactoring and repair can maintain, and even enhance a system's structure as it evolves. The second is to throw the system away and start over. The RECONSTRUCTION pattern explores this drastic, but frequently necessary alternative. The third is to simply surrender to entropy, and wallow in the mire.

Since the time of Roman architect Marcus Vitruvius, [Vitruvius 20 B.C.] architects have focused on his trinity of desirables: *Firmitas* (**strength**), *Utilitas* (**utility**), and *Venustas* (**beauty**). A BIG BALL OF MUD usually represents a triumph of utility over aesthetics, because workmanship is sacrificed for functionality. Structure and durability can be sacrificed as well, because an incomprehensible program defies attempts at maintenance. The frenzied, feature-driven "bloatware" phenomenon seen in many large consumer software products can be seen as evidence of designers having allowed purely utilitarian concerns to dominate software design.

THROWAWAY CODE

alias
QUICK HACK
KLEENEX CODE
DISPOSABLE CODE
SCRIPTING
KILLER DEMO
PERMANENT PROTOTYPE
BOOMTOWN

❖ ❖ ❖

A homeowner might erect a temporary storage shed or car port, with every intention of quickly tearing it down and replacing it with something more permanent. Such structures have a way of enduring indefinitely. The money expected to replace them might not become available. Or, once the new structure is constructed, the temptation to continue to use the old one for "a while" might be hard to resist.

Likewise, 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.

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

Time, or a lack thereof, is frequently the decisive force that drives programmers to write THROWAWAY CODE. Taking the time to write a proper, well thought out, well documented program might take more time that is available to solve a problem, or more time that the problem merits. Often, the programmer will make a frantic dash to construct a minimally functional program, while all the while promising him or herself that a better factored, more elegant version will follow thereafter. They may know full well that building a reusable system will make it easier to solve similar problems in the future, and that a more polished architecture would result in a system that was easier to maintain and extend.

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.

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

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.

Programmers are usually not domain experts, especially at first. Use cases or CRC cards [Beck & Cunningham 1989] can help them to discover domain objects. However, nothing beats building a prototype to help a team learn its way around a domain.

When you build a prototype, there is always the risk that someone will say "that's good enough, ship it". One way to minimize the risk of a prototype being put into production is to write the prototype in using a language or tool that you couldn't possible use for a production version of your product.

Proponents of Extreme Programming [Beck 2000] often construct quick, disposable prototypes called "spike solutions". Prototypes help us learn our way around the problem space, but should never be mistaken for good designs [Johnson & Foote 1988].

Not every program need be a palace. A simple throwaway program is like a tent city or a mining boomtown, and often has no need for fifty year solutions to its problems, given that it will give way to a ghost town in five.

The real problem with THROWAWAY CODE comes when it isn't thrown away.

❖ ❖ ❖

The production of THROWAWAY CODE is a nearly universal practice. Any software developer, at any skill or experience level, can be expected to have had at least occasional first-hand experience with this approach to software development. For example, in the patterns community, two examples of quick-and-dirty code that have endured are the PLoP online registration code, and the Wiki-Wiki Web pages.

The EuroPLoP/PLoP/UP online registration code was, in effect, a distributed web-based application that ran on four different machines on two continents. Conference information was maintained on a machine in St. Louis, while registration records were kept on machines in Illinois and Germany. The system could generate web-based reports of registration activity, and now even instantaneously maintaineed an online attendees list. It began life in 1995 as a quick-and-dirty collection of HTML, scavenged C demonstration code, and csh scripts. It was undertaken largely as an experiment in web-based form processing prior to PLoP ‘95, and, like so many things on the Web, succeeded considerably beyond the expectations of its authors. Today, it is still essentially the same collection of HTML, scavenged C demonstration code, and csh scripts. As such, it showcases how quick-and-dirty code can, when successful, take on a life of its own.

The original C code and scripts probably contained fewer than three dozen original lines of code. Many lines were cut-and-paste jobs that differed only in the specific text they generate, or fields that they check.

Here’s an example of one of the scripts that generates the attendance report:

```
echo "<H2>Registrations: <B>" `ls | wc  -l` "</B></H2>"
echo "<CODE>"
echo "Authors: <B>" `grep 'Author = Yes' * | wc  -l` "</B>"
echo "<BR>"
echo "Non-Authors: <B>" `grep 'Author = No' * | wc  -l` "</B>"
```

```
echo "<BR><BR>"
```

This script is slow and inefficient, particularly as the number of registrations increases, but not least among its virtues is the fact that it *works*. Were the number of attendees to exceed more than around one hundred, this script would start to perform so badly as to be unusable. However, since hundreds of attendees would exceed the physical capacity of the conference site, we knew the number of registrations would have been limited long before the performance of this script became a significant problem. So while this approach is, in general, a lousy way to address this problem, it is perfectly satisfactory within the confines of the particular purpose for which the script has ever actually been used. Such practical constraints are typical of THROWAWAY CODE, and are more often than not undocumented. For that matter, everything about THROWAWAY CODE is more often than not undocumented. When documentation exists, it is frequently not current, and often not accurate.

The Wiki-Web code at www.c2.com also started as a CGI experiment undertaken by Ward Cunningham also succeeded beyond the author's expectations. The name "wiki" is one of Ward's personal jokes, having been taken from a Hawaiian word for "quick" that the author had seen on an airport van on a vacation in Hawaii. Ward has subsequently used the name for a number of quick-and-dirty projects. The Wiki Web is unusual in that any visitor may change anything that anyone else has written indiscriminately. This would seem like a recipe for vandalism, but in practice, it has worked out well. In light of the system's success, the author has subsequently undertaken additional work to polish it up, but the same quick-and-dirty Perl CGI core remains at the heart of the system.

Both systems might be thought of as being on the verge of graduating from little balls of mud to BIG BALLS OF MUD. The registration system's C code *metastasized* from one of the NCSA HTTPD server demos, and still contains zombie code that testifies to this heritage. At each step, KEEPING IT WORKING is a premiere consideration in deciding whether to extend or enhance the system. Both systems might be good candidates for RECONSTRUCTION, were the resources, interest, and audience present to justify such an undertaking. In the mean time, these systems, which are still sufficiently well suited to the particular tasks for which they were built, remain in service. Keeping them on the air takes far less energy than rewriting them. They continue to evolve, in a PIECEMEAL fashion, a little at a time.

You can ameloriate the architectural erosion that can be caused by quick-and-dirty code by isolating it from other parts of your system, in its own objects, packages, or modules. To the extent that such code can be quarantined, its ability to affect the integrity of healthy parts of a system is reduced.

Once it becomes evident that a purportedly disposable artifact is going to be around for a while, one can turn one's attention to improving its structure, either through an iterative process of PIECEMEAL GROWTH, or via a fresh draft, as discussed in the RECONSTRUCTION pattern.

From *boomtown* to *ghost town*:
The mining town of **Rhyolite**, in Death Valley, was briefly the third largest city in Nevada.
Then the ore ran out.

PIECEMEAL GROWTH

alias
URBAN SPRAWL
ITERATIVE-INCREMENTAL DEVELOPMENT

The Russian Mir ("Peace") Space Station Complex was designed for reconfiguration and modular growth. The Core module was launched in 1986, and the Kvant ("Quantum") and Kvant-2 modules joined the complex in 1987 and 1989. The Kristall ("Crystal") module was added in 1990. The Spektr ("Spectrum") and shuttle Docking modules were added in 1995, the latter surely a development not anticipated in 1986. The station's final module, Priroda ("Nature"), was launched in 1996. The common core and independent maneuvering capabilities of several of the modules have allowed the complex to be rearranged several times as it has grown.

Urban planning has an uneven history of success. For instance, Washington D.C. was laid out according to a master plan designed by the French architect L'Enfant. The capitals of Brazil (Brasilia) and Nigeria (Abuja) started as paper cities as well. Other cities, such as Houston, have grown without any overarching plan to guide

them. Each approach has its problems. For instance, the radial street plans in L'Enftant's master plan become awkward past a certain distance from the center. The lack of any plan at all, on the other hand, leads to a patchwork of residential, commercial, and industrial areas that is dictated by the capricious interaction of local forces such as land ownership, capital, and zoning. Since concerns such as recreation, shopping close to homes, and noise and pollution away from homes are not brought directly into the mix, they are not adequately addressed.

Most cities are more like Houston than Abuja. They may begin as settlements, subdivisions, docks, or railway stops. Maybe people were drawn by gold, or lumber, access to transportation, or empty land. As time goes on, certain settlements achieve a critical mass, and a positive feedback cycle ensues. The city's success draws tradesmen, merchants, doctors, and clergymen. The growing population is able to support infrastructure, governmental institutions, and police protection. These, in turn, draw more people. Different sections of town develop distinct identities. With few exceptions, (Salt Lake City comes to mind) the founders of these settlements never stopped to think that they were founding major cities. Their ambitions were usually more modest, and immediate.

❖ ❖ ❖

It has become fashionable over the last several years to take pot shots at the "traditional" waterfall process model. It may seem to the reader that attacking it is tantamount to flogging a dead horse. However, if it be a dead horse, it is a tenacious one. While the approach itself is seen by many as having been long since discredited, it has spawned a legacy of rigid, top-down, front-loaded processes and methodologies that endure, in various guises, to this day. We can do worse that examine the forces that led to its original development.

In the days before waterfall development, programming pioneers employed a simple, casual, relatively undisciplined "code-and-fix" approach to software development. Given the primitive nature of the problems of the day, this approach was frequently effective. However, the result of this lack of

discipline was, all too often, a BIG BALL OF MUD.

The waterfall approach arose in response to this muddy morass. While the code-and-fix approach might have been suitable for small jobs, it did not scale well. As software became more complex, it would not do to simply gather a room full of programmers together and tell them to go forth and code. Larger projects demanded better planning and coordination. Why, it was asked, can't software be engineered like cars and bridges, with a careful analysis of the problem, and a detailed up-front design prior to implementation? Indeed, an examination of software development costs showed that problems were many times more expensive to fix during maintenance than during design. Surely it was best to mobilize resources and talent up-front, so as to avoid maintenance expenses down the road. It's surely wiser to route the plumbing correctly now, before the walls are up, than to tear holes in them later. Measure twice, cut once.

One of the reasons that the waterfall approach was able to flourish a generation ago was that computers and business requirements changed at a more leisurely pace. Hardware was very expensive, often dwarfing the salaries of the programmers hired to tend it. User interfaces were primitive by today's standards. You could have any user interface you wanted, as long as it was an alphanumeric "green screen". Another reason for the popularity of the waterfall approach was that it exhibited a comfortable similarity to practices in more mature engineering and manufacturing disciplines.

Today's designers are confronted with a broad onslaught of changing requirements. It arises in part from the rapid growth of technology itself, and partially from rapid changes in the business climate (some of which is driven by technology). Customers are used to more sophisticated software these days, and demand more choice and flexibility. Products that were once built from the ground up by in-house programmers must now be integrated with third-party code and applications. User interfaces are complex, both externally and internally. Indeed, we often dedicate an entire tier of our system to their care and feeding. Change threatens to outpace our ability to cope with it.

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

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. You can't simply do what the customer wants, for quite often, they don't know what they want. You can't simply plan, you have to plan to be able to adapt. If you can't fully anticipate what is going to happen, you must be prepared to be nimble.

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. A superior design, given this mindset, is one that elegantly and completely specifies the system's structure before a single line of code has been written. Mismatches between these blueprints and reality are considered aberrations, and are treated as mistakes on the part of the designer. A better design would have anticipated these oversights. In the presence of volatile requirements, aspirations towards such design perfection are as vain as the desire for a hole-in-one on every hole.

To avoid such embarrassment, the designer may attempt to cover him or herself by specifying a more complicated, and more general solution to certain problems, secure in the knowledge that others will bear the burden of constructing these artifacts. When such predictions about where complexity is needed are correct, they can indeed be a source of power and satisfaction. This is part of their allure of

Venustas. However, sometime the anticipated contingencies never arise, and the designer and implementers wind up having wasted effort solving a problem that no one has ever actually had. Other times, not only is the anticipated problem never encountered, its solution introduces complexity in a part of the system that turns out to need to evolve in another direction. In such cases, speculative complexity can be an unnecessary obstacle to subsequent adaptation. It is ironic that the impulse towards elegance can be an unintended source of complexity and clutter instead.

In its most virulent form, the desire to anticipate and head off change can lead to "analysis paralysis", as the thickening web of imagined contingencies grows to the point where the design space seems irreconcilably constrained.

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

Successful software attracts a wider audience, which can, in turn, place a broader range of requirements on it. These new requirements can run against the grain of the original design. Nonetheless, they can frequently be addressed, but at the cost of cutting across the grain of existing architectural assumptions. [Foote 1988] called this architectural erosion *midlife generality loss*.

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. With real kitchens, the board of health will eventually intervene. With software, alas, there is seldom any corresponding agency to police such squalor. Uncontrolled growth can ultimately be a malignant force. The result of neglecting to contain it can be a BIG BALL OF MUD.

In *How Buildings Learn*, Brand [Brand 1994] observed that what he called *High Road* architecture often resulted in buildings that were expensive and difficult to change, while vernacular, *Low Road* buildings like bungalows and warehouses were, paradoxically, much more adaptable. Brand noted that *Function melts form*, and low road buildings are more amenable to such change. Similarly, with software, you may be reluctant to desecrate another programmer's cathedral. Expedient changes to a low road system that exhibits no discernable architectural pretensions to begin with are easier to rationalize.

In the Oregon Experiment [Brand 1994][Alexander 1988] Alexander noted:

> *Large-lump development is based on the idea of **replacement**. Piecemeal Growth is based on the idea of **repair**. … Large-lump development is based on the fallacy that it is possible to build perfect buildings. Piecemeal growth is based on the healthier and more realistic view that mistakes are inevitable. … Unless money is available for repairing these mistakes, every building, once built, is condemned to be, to some extent unworkable. … Piecemeal growth is based on the assumption that adaptation between buildings and their users is necessarily a slow and continuous business which cannot, under any circumstances, be achieve in a single leap.*

Alexander has noted that our mortgage and capital expenditure policies make large sums of money available up front, but do nothing to provide resources for maintenance, improvement, and evolution

[Brand 1994][Alexander 1988]. In the software world, we deploy our most skilled, experienced people early in the lifecycle. Later on, maintenance is relegated to junior staff, and resources can be scarce. The so-called maintenance phase is the part of the lifecycle in which the price of the fiction of master planning is really paid. It is maintenance programmers who are called upon to bear the burden of coping with the ever widening divergence between fixed designs and a continuously changing world. If the hypothesis that architectural insight emerges late in the lifecycle is correct, then this practice should be reconsidered.

Brand went on to observe *Maintenance **is** learning.* He distinguishes three levels of learning in the context of systems. This first is habit, where a system dutifully serves its function within the parameters for which it was designed. The second level comes into play when the system must adapt to change. Here, it usually must be modified, and its capacity to sustain such modification determines it's degree of adaptability. The third level is the most interesting: *learning to learn*. With buildings, adding a raised floor is an example. Having had to sustain a major upheaval, the system adapts so that subsequent adaptations will be much less painful.

PIECEMEAL GROWTH can be undertaken in an opportunistic fashion, starting with the existing, living, breathing system, and working outward, a step at a time, in such a way as to not undermine the system's viability. You enhance the program as you use it. Broad advances on all fronts are avoided. Instead, change is broken down into small, manageable chunks.

One of the most striking things about PIECEMEAL GROWTH is the role played by *Feedback*. Herbert Simon [Simon 1969] has observed that few of the adaptive systems that have been forged by evolution or shaped by man depend on prediction as their main means of coping with the future. He notes that two complementary mechanisms, homeostasis, and retrospective feedback, are often far more effective. Homeostasis insulates the system from short-range fluctuations in its environment, while feedback mechanisms respond to long-term discrepancies between a system's actual and desired behavior, and adjust it accordingly. Alexander [Alexander 1964] has written extensively the roles that homeostasis and feedback play in adaptation as well.

If you can adapt quickly to change, predicting it becomes far less crucial. Hindsight, as Brand observes [Brand 1994] is better than foresight. Such rapid adaptation is the basis of one of the mantras of Extreme Programming [Beck 2000]: *You're not going to need it*.

Proponents of XP (as it is called) say to pretend you are not a smart as you think you are, and wait until this clever idea of yours is actually required before you take the time to bring it into being. In the cases where you were right, hey, you saw it coming, and you know what to do. In the cases where you were wrong, you won't have wasted any effort solving a problem you've never had when the design heads in an unanticipated direction instead.

Extreme Programming relies heavily on feedback to keep requirements in sync with code, by emphasizing short (three week) iterations, and extensive, continuous consultation with users regarding design and development priorities throughout the development process. Extreme Programmers do not engage in extensive up-front planning. Instead, they produce working code as quickly as possible, and steer these prototypes towards what the users are looking for based on feedback.

Feedback also plays a role in determining coding assignments. Coders who miss a deadline are assigned a different task during the next iteration, regardless of how close they may have been to completing the task. This form of feedback resembles the stern justice meted out by the jungle to the

fruit of uncompetitive pairings.

Extreme Programming also emphasizes testing as an integral part of the development process. Tests are developed, ideally, before the code itself. Code is continuously tested as it is developed.

There is a "back-to-the-future" quality to Extreme Programming. In many respects, it resembles the blind *Code and Fix* approach. The thing that distinguishes it is the central role played by feedback in driving the system's evolution. This evolution is abetted, in turn, by modern object-oriented languages and powerful refactoring tools.

Proponents of extreme programming portray it as placing minimal emphasis on planning and up-front design. They rely instead on feedback and continuous integration. We believe that a certain amount of up-front planning and design is not only important, but inevitable. No one really goes into any project blindly. The groundwork must be laid, the infrastructure must be decided upon, tools must be selected, and a general direction must be set. A focus on a shared architectural vision and strategy should be established early.

Unbridled, change can undermine structure. Orderly change can enhance it. Change can engender malignant sprawl, or healthy, orderly growth.

A broad consensus that objects emerge from an *iterative incremental* evolutionary process has formed in the object-oriented community over the last decade. See for instance [Booch 1994]. The SOFTWARE TECTONICS pattern [Foote & Yoder 1996] examines how systems can incrementally cope with change.

The biggest risk associated with PIECEMEAL GROWTH is that it will gradually erode the overall structure of the system, and inexorably turn it into a BIG BALL OF MUD. A strategy of KEEPING IT WORKING goes hand in hand with PIECEMEAL GROWTH. Both patterns emphasize acute, local concerns at the expense of chronic, architectural ones.

To counteract these forces, a permanent commitment to CONSOLIDATION and refactoring must be made. It is through such a process that local and global forces are reconciled over time. This lifecyle perspective has been dubbed the *fractal model* [Foote & Opdyke 1995]. To quote Alexander [Brand 1994][Alexander 1988]:

> *An organic process of growth and repair must create a gradual sequence of changes, and these changes must be distributed evenly across all levels of scale. [In developing a college campus] there must be as much attention to the repair of details—rooms, wings of buildings, windows, paths—as to the creation of brand new buildings. Only then can the environment be balanced both as a whole, and in its parts, at every moment in its history.*

KEEP IT WORKING

alias
VITALITY
BABY STEPS
DAILY BUILD
FIRST, DO NO HARM

```
Probably the greatest factor that keeps us moving forward is that we use the
system all the time, and we keep trying to do new things with it. It is this
"living-with" which drives us to root out failures, to clean up inconsistencies,
and which inspires our occasional innovation.
```

Daniel H. H. Ingalls [Ingalls 1983]

Once a city establishes its infrastructure, it is imperative that it be kept working. For example, if the sewers break, and aren't quickly repaired, the consequences can escalate from merely unpleasant to genuinely life threatening. People come to expect that they can rely on their public utilities being available 24 hours per day. They (rightfully) expect to be able to demand that an outage be treated as an emergency.

❖ ❖ ❖

Software can be like this. Often a business becomes dependent upon the data driving it. Businesses have become critically dependent on their software and computing infrastructures. There are numerous mission critical systems that must be on-the-air twenty-four hours a day/seven days per week. If these systems go down, inventories can not be checked, employees can not be paid, aircraft cannot be routed, and so on.

There may be times where taking a system down for a major overhaul can be justified, but usually, doing so is fraught with peril. However, once the system is brought back up, it is difficult to tell which from among a large collection of modifications might have caused a new problem. Every change is suspect. This is why deferring such integration is a recipe for misery. Capers Jones [Jones 1999] reported that the chance that a significant change might contain a new error--a phenomenon he ominously referred to as a *Bad Fix Injection*-- was about 7% in the United States. This may strike some readers as a low figure. Still, it's easy to see that compounding this possibility can lead to a situation where multiple upgrades are increasing likely to break a system.

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

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. The idea of the architect-builder is a central theme of Alexander's work. Who better to resolve the forces impinging upon each design issue as it arises as the person who is going to have to live with these decisions? The architect-builder will be the direct beneficiary of his or her own workmanship and care. Mistakes and shortcuts will merely foul his or her own nest.

Dependability: These days, people rely on our software artifacts for their very livelihoods, and even, at time, for their very safety. It is imperative that ill-advise changes to elements of a system do not drag the entire system down. Modern software systems are intricate, elaborate webs of interdependent

elements. When an essential element is broken, everyone who depends on it will be affected. Deadlines can be missed, and tempers can flare. This problem is particularly acute in BIG BALLS OF MUD, since a single failure can bring the entire system down like a house of cards.

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

When you are living in the system you're building, you have an acute incentive not to break anything. A plumbing outage will be a direct inconvenience, and hence you have a powerful reason to keep it brief. You are, at times, working with live wires, and must exhibit particular care. A major benefit of working with a live system is that feedback is direct, and nearly immediate.

One of the strengths of this strategy is that modifications that break the system are rejected immediately. There are always a large number of paths forward from any point in a system's evolution, and most of them lead nowhere. By immediately selecting only those that do not undermine the system's viability, obvious dead-ends are avoided.

Of course, this sort of reactive approach, that of kicking the nearest, meanest woolf from your door, is not necessarily globally optimal. Yet, by eliminating obvious wrong turns, only more insidiously incorrect paths remain. While these are always harder to identify and correct, they are, fortunately less numerous than those cases where the best immediate choice is also the best overall choice as well.

It may seem that this approach only accommodates minor modifications. This is not necessarily so. Large new subsystems might be constructed off to the side, perhaps by separate teams, and integrated with the running system in such a way as to minimize distruption.

Design space might be thought of as a vast, dark, largely unexplored forest. Useful potential paths through it might be thought of as encompassing working programs. The space off to the sides of these paths is much larger realm of non-working programs. From any given point, a few small steps in most directions take you from a working to a non-working program. From time to time, there are forks in the path, indicating a choice among working alternatives. In unexplored territory, the prudent strategy is never to stray too far from the path. Now, if one has a map, a shortcut through the trekless thicket that might save miles may be evident. Of course, pioneers, by definition, don't have maps. By taking small steps in any direction, they know that it is never more than a few steps back to a working system.

> Some years ago, Harlan Mills proposed that any software system should be grown by incremental development. That is, the system first be made to run, even though it does nothing useful except call the proper set of dummy subprograms. Then, bit by bit, it is fleshed out, with the subprograms in turn being developed into actions or calls to empty stubs in the level below.
>
> …
>
> Nothing in the past decade has so radically changed my own practice, and its effectiveness.
>
> …
>
> One always has, at every stage, in the process, a working system. I find that teams can *grow* much more complex entities in four months than they can *build.*
>
> -- *From "No Silver Bullet"* [Brooks 1995]

Microsoft mandates that a DAILY BUILD of each product be performed at the end of each working day. Nortel adheres to the slightly less demanding requirement that a working build be generated at the end of each week [Brooks 1995][Cusumano & Shelby 1995]. Indeed, this approach, and keeping the last working version around, are nearly universal practices among successful maintenance programmers.

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 works. 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.

❖ ❖ ❖

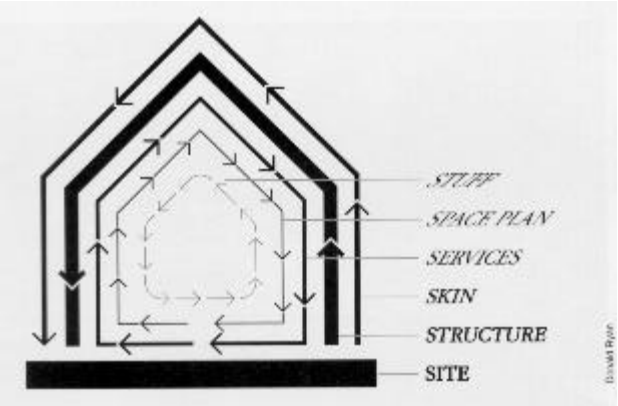
Always beginning with a working system helps to encourage PIECEMEAL GROWTH. Refactoring is the primary means by which programmers maintain order from inside the systems in which they are working. The goal of refactoring is to leave a system working as well after a refactoring as it was before the refactoring. Aggressive unit and integration testing can help to guarantee that this goal is met.

SHEARING LAYERS


```
Hummingbirds and flowers are quick, redwood trees are slow, and whole redwood
  forests are even slower. Most interaction is within the same pace level--
hummingbirds and flowers pay attention to each other, oblivious to redwoods, who
                          are oblivious to them.
```

R. V. O'Neill, *A Hierarchical Concept of Ecosystems*

The notion of SHEARING LAYERS is one of the centerpieces of Brand's *How Buildings Learn* [Brand 1994]. Brand, in turn synthesized his ideas from a variety of sources, including British designer Frank Duffy, and ecologist R. V. O'Neill.



Brand quotes Duffy as saying: "Our basic argument is that there isn't any such thing as a building. A building properly conceived is several layers of longevity of built components".

Brand distilled Duffy's proposed layers into these six: Site, Structure, Skin, Services, Space Plan, and Stuff. Site is geographical setting. Structure is the load bearing elements, such as the foundation and skeleton. Skin is the exterior surface, such as siding and windows. Services are the circulatory and nervous systems of a building, such as its heating plant, wiring, and plumbing. The Space Plan includes walls, flooring, and ceilings. Stuff includes lamps, chairs, appliances, bulletin boards, and paintings.

These layers change at different rates. Site, they say, is eternal. Structure may last from 30 to 300 years. Skin lasts for around 20 years, as it responds to the elements, and to the whims of fashion. Services succumb to wear and technical obsolescence more quickly, in 7 to 15 years. Commercial Space Plans may turn over every 3 years. Stuff, is, of course, subject to unrelenting flux [Brand 1994].

❖ ❖ ❖

Software systems cannot stand still. Software is often called upon to bear the brunt of changing requirements, because, being as that it is made of bits, it can change.

Different artifacts change at different rates.

Adaptability: A system that can cope readily with a wide range of requirements, will, all other things being equal, have an advantage over one that cannot. Such a system can allow unexpected requirements to be met with little or no reengineering, and allow its more skilled customers to rapidly address novel challenges.

Stability: Systems succeed by doing what they were designed to do as well as they can do it. They earn their niches, by bettering their competition along one or more dimensions such as cost, quality, features, and performance. See [Foote & Roberts 1998] for a discussion of the occasionally fickle nature of such completion. Once they have found their niche, for whatever reason, it is essential that short term concerns not be allowed to wash away the elements of the system that account for their mastery of their niche. Such victories are inevitably hard won, and fruits of such victories should not be squandered. Those parts of the system that do what the system does well must be protected from fads, whims, and other such spasms of poor judgement.

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.

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

Most interactions in a system tend to be within layers, or between adjacent layers. Individual layers

tend to be about things that change at similar rates. Things that change at different rates diverge. Differential rates of change encourage layers to emerge. Brand notes as well that occupational specialties emerge along with these layers. The rate at which things change shapes our organizations as well. For instance, decorators and painters concern themselves with interiors, while architects dwell on site and skin. We expect to see things that evolve at different rates emerge as distinct concerns. This is "separate that which changes from that which doesn't" [Roberts & Johnson 1998] writ large.

Can we identify such layers in software?

Well, at the bottom, there are data. Things that change most quickly migrate into the data, since this is the aspect of software that is most amenable to change. Data, in turn, interact with users themselves, who produce and consume them.

Code changes more slowly than data, and is the realm of programmers, analysts and designers. In object-oriented languages, things that will change quickly are cast as black-box polymorphic components. Elements that will change less often may employ white-box inheritance.

The abstract classes and components that constitute an object-oriented framework change more slowly than the applications that are built from them. Indeed, their role is to distill what is common, and enduring, from among the applications that seeded the framework.

As frameworks evolve, certain abstractions make their ways from individual applications into the frameworks and libraries that constitute the system's infrastructure [Foote 1988]. Not all elements will make this journey. Not all should. Those that do are among the most valuable legacies of the projects that spawn them. Objects help shearing layers to emerge, because they provide places where more fine-grained chunks of code and behavior that belong together can coalesce.

The Smalltalk programming language is built from a set of objects that have proven themselves to be of particular value to programmers. Languages change more slowly than frameworks. They are the purview of scholars and standards committees. One of the traditional functions of such bodies is to ensure that languages evolve at a suitably deliberate pace.

Artifacts that evolve quickly provide a system with dynamism and flexibility. They allow a system to be fast on its feet in the face of change.

Slowly evolving objects are bulwarks against change. They embody the wisdom that the system has accrued in its prior interactions with its environment. Like tenure, tradition, big corporations, and conservative politics, they maintain what has worked. They worked once, so they are kept around. They had a good idea once, so maybe they are a better than even bet to have another one.

Wide acceptance and deployment causes resistance to change. If changing something will break a lot of code, there is considerable incentive not to change it. For example, schema reorganization in large enterprise databases can be an expensive and time-consuming process. Database designers and administrators learn to resist change for this reason. Separate job descriptions, and separate hardware, together with distinct tiers, help to make these tiers distinct.

The phenomenon whereby distinct concerns emerge as distinct layers and tiers can be seen as well with graphical user interfaces.

Part of the impetus behind using METADATA [Foote & Yoder 1998b] 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. The effect is analogous to that seen with modular office furniture, which allows office workers to easily, quickly, and cheaply move partitions without having to enlist architects and contractors in the effort.

Over time, our frameworks, abstract classes, and components come to embody what we've learned about the structure of the domains for which they are built. More enduring insights gravitate towards the primary structural elements of these systems. Things which find themselves in flux are spun out into the data, where users can interact with them. Software evolution becomes like a centrifuge spun by change. The layers that result, over time, can come to a much truer accommodation with the forces that shaped them than any top-down agenda could have devised.

> Things that are good have a certain kind of structure. You can't get that structure except dynamically. Period. In nature you've got continuous very-small-feedback-loop adaptation going on, which is why things get to be harmonious. That's why they have the qualities we value. If it wasn't for the time dimension, it wouldn't happen. Yet here we are playing the major role creating the world, and we haven't figured this out. That is a very serious matter.
>
> **Christopher Alexander** -- [Brand 1994]

❖ ❖ ❖

This pattern has much in common with the HOT SPOTS pattern discussed in [Roberts & Johnson 1998]. Indeed, separating things that change from those that do not is what drives the emergence of SHEARING LAYERS. These layers are the result of such differential rates of change, while HOT SPOTS might be thought of as the rupture zones in the fault lines along which slippage between layers occurs. This tectonic slippage is suggestive as well of the SOFTWARE TECTONICS pattern [Foote

& Yoder 1996], which recommends fine-grained iteration as a means of avoiding catastrophic upheaval. METADATA and ACTIVE OBJECT-MODELS [Foote & Yoder 1998b] allow systems to adapt more quickly to changing requirements by pushing power into the data, and out onto users.

SWEEPING IT UNDER THE RUG

alias
POTEMKIN VILLAGE
HOUSECLEANING
PRETTY FACE
QUARANTINE
HIDING IT UNDER THE BED
REHABILITATION

One of the most spectacular examples of *sweeping a problem under the rug* is the concrete sarcophagus that Soviet engineers constructed to put a 10,000 year lid on the infamous reactor number four at Chernobyl, in what is now Ukraine.

If you can't make a mess go away, at least you can hide it. Urban renewal can begin by painting murals over graffiti and putting fences around abandoned property. Children often learn that a single heap in the closet is better than a scattered mess in the middle of the floor.

❖ ❖ ❖

There are reasons, other than aesthetic concerns, professional pride, and guilt for trying to clean up messy code. A deadline may be nearing, and a colleague may want to call a chunk of your code, if you could only come up with an interface through which it could be called. If you don't come up with an easy to understand interface, they'll just use someone else's (perhaps inferior) code. You might be cowering during a code-review, as your peers trudge through a particularly undistinguished example of your work. You know that there are good ideas buried in there, but that if you don't start to make them more evident, they may be lost.

There is a limit to how much chaos an individual can tolerate before being overwhelmed. At first glance, a BIG BALL OF MUD can inspire terror and despair in the hearts of those who would try to tame it. 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.

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.

Comprehensibility: It should go without saying that comprehensible, attractive, well-engineered code will be easier to maintain and extend than complicated, convoluted code. However, it takes *Time* and money to overhaul sloppy code. Still, the *Cost* of allowing it to fester and continue to decline should not be underestimated.

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. Programmers become timid, afraid that tugging at a loose thread may have unpredictable consequences. After a while, the myriad threads that couple every part of the system to every other come to tie the programmer down as surely as Gulliver among the Lilliputians [Swift 1726]. Talent may desert the project in the face of such bondage.

It should go without saying that comprehensible, attractive, well-engineered code will be easier to maintain and extend than complicated, convoluted code. However, it takes time and money to overhaul sloppy code. Still, the cost of allowing it to fester and continue to decline should not be underestimated.

***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.**

By getting the dirt into a single pile beneath the carpet, you at least know where it is, and can move it around. You've still got a pile of dirt on your hands, but it is localized, and your guests can't see it. As the engineers who entombed reactor number four at Chernobly demonstrated, sometimes you've got to get a lid on a problem before you can get serious about cleaning things up. Once the problem area is contained, you can decontaminate at a more leisurely pace.

To begin to get a handle on spaghetti code, find those sections of it that seem less tightly coupled, and start to draw architectural boundaries there. Separate the global information into distinct data structures, and enforce communication between these enclaves using well-defined interfaces. Such steps can be the first ones on the road to re-establishing the system's conceptual

integrity, and discerning nascent architectural landmarks.

Putting a fresh interface around a run down region of the system can be the first step on the way architectural rehabilitation. This is a long row to hoe, however. Distilling meaningful abstractions from a BIG BALL OF MUD is a difficult and demand task. It requires skill, insight, and persistence. At times, RECONSTRUCTION may seem like the less painful course. Still, it is not like unscrambling an egg. As with rehabilitation in the real world, restoring a system to architectural health requires resources, as well as a sustained commitment on the part of the people who live there.

The UIMX user interface builder for Unix and Motif, and the various Smalltalk GUI builders both provide a means for programmers to cordon off complexity in this fashion.

❖ ❖ ❖

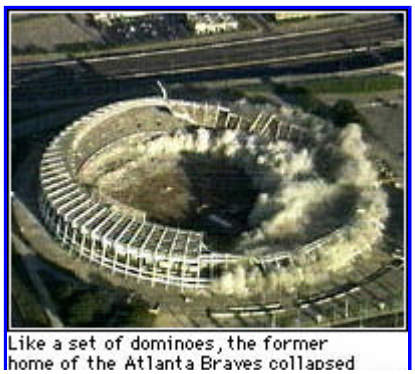
One frequently constructs a FAÇADE [Gamma et. al. 1995] to put a congenial "pretty face" on the unpleasantness that is SWEPT UNDER THE RUG. Once these messy chunks of code have been quarantined, you can expose their functionality using INTENTION REVEALING SELECTORS [Beck 1997].

This can be the first step on the road to CONSOLIDATION too, since one can begin to hem in unregulated growth than may have occurred during PROTOTYPING or EXPANSION [Foote & Opdyke 1995]. [Foote & Yoder 1998a] explores how, ironically, inscrutable code can persist because it is difficult to comprehend.

This paper also examines how complexity can be hidden using suitable defaults (WORKS OUT OF THE BOX and PROGRAMMING-BY-DIFFERRENCE), and interfaces that gradually reveal additional capabilities as the client grows more sophisticated.

RECONSTRUCTION

alias
TOTAL REWRITE
DEMOLITION
THROWAWAY THE FIRST ONE
START OVER



Like a set of dominoes, the former home of the Atlanta Braves collapsed

Atlanta's Fulton County Stadium was built in 1966 to serve as the home of baseball's Atlanta Braves, and football's Atlanta Falcons. In August of 1997, the stadium was demolished. Two factors contributed to its relatively rapid obsolescence. One was that the architecture of the original stadium was incapable of accommodating the addition of the "sky-box" suites that the spreadsheets of '90s sporting economics demanded. No conceivable retrofit could accommodate this requirement. Addressing it meant starting over, from the ground up. The second was that the stadium's attempt to provide a cheap, general solution to the problem of providing a forum for both baseball and football audiences compromised the needs of both. In only thirty-one years, the balance among these forces had shifted decidedly. The facility is being replaced by two new single-purpose stadia.

Might there be lessons for us about unexpected requirements and designing general components here?

❖ ❖ ❖

Plan to Throw One Away (You Will Anyway) -- Brooks

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. However, less emphasis is given to value of the experience the team might have salvaged from their initial, unsuccessful draft. Could this first draft have been the unsung hero of this tale?

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

Obsolescence: Of course, one reason to abandon a system is that it is in fact technically or economically obsolete. These are distinct situations. A system that is no longer state-of-the-art may still sell well, while a technically superior system may be overwhelmed by a more popular competitor for non-technical reasons.

In the realm of concrete and steel, blight is the symptom, and a withdrawal of capital is the cause. Of course, once this process begins, it can feed on itself. On the other hand, given a steady infusion of resources, buildings can last indefinitely. It's not merely entropy, but an unwillingness to counteract it, that allows buildings to decline. In Europe, neighborhoods have flourished for hundreds of years. They have avoided the boom/bust cycles that characterize some New World cities.

Change: Even though software is a highly malleable medium, like Fulton County Stadium, new demands can, at times, cut across a system's architectural assumptions in such a ways as to make accommodating them next to impossible. In such cases, a total rewrite might be the only answer.

Cost: Writing-off a system can be traumatic, both to those who have worked on it, and to those who have paid for it. Software is often treated as an asset by accountants, and can be an expensive asset at that. Rewriting a system, of course, does not discard its conceptual design, or its staff's experience. If it is truly the case that the value of these assets is in the design experience they embody, then accounting practices must recognize this.

Organization: Rebuilding a system from scratch is a high-profile undertaking, that will demand

considerable time and resources, which, in turn, will make high-level management support essential.

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

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.

One reason to start over might be that the previous system was written by people who are long gone. Doing a rewrite provides new personnel with a way to reestablish contact between the architecture and the implementation. Sometimes the only way to understand a system it is to write it yourself. Doing a fresh draft is a way to overcome neglect. Issues are revisited. A fresh draft adds vigor. You draw back to leap. The quagmire vanishes. The swamp is drained.

Another motivation for building a new system might be that you feel that you've got the experience you need to do the job properly. One way to have gotten this experience is to have participated at some level in the unsuccessful development of a previous version of the system.

Of course, the new system is not designed in a vacuum. Brook's famous tar pit is excavated, and the fossils are examined, to see what they can tell the living. It is essential that a thorough post-mortem review be done of the old system, to see what it did well, and why it failed. Bad code can bog down a good design. A good design can isolate and contain bad code.

When a system becomes a BIG BALL OF MUD, its relative incomprehensibility may hasten its demise, by making it difficult for it to adapt. It can persist, since it resists change, but cannot evolve, for the same reason. Instead, its inscrutability, even when it is to its s hort-term benefit, sows the seeds of its ultimate demise.

If this makes muddiness a frequently terminal condition, is this really a bad thing? Or is it a blessing that these sclerotic systems yield the stage to more agile successors? Certainly, the departure of these ramshackle relics can be a cause for celebration as well as sadness.

Discarding a system dispenses with its implementation, and leaves only its conceptual design behind. Only the patterns that underlie the system remain, grinning like a Cheshire cat. It is their spirits that help to shape the next implementation. With luck, these architectural insights will be reincarnated as genuine reusable artifacts in the new system, such as abstract classes and frameworks. It is by finding these architectural nuggets that the promise of objects and reuse can finally be fulfilled.

There are alternatives to throwning your system away and starting over. One is to embark on a regimen of incremental refactoring, to glean architectural elements and discernable abstractions from the mire. Indeed, you can begin by looking for coarse fissures along which to separate parts of the system, as was suggested in SWEEPING IT UNDER THE RUG. Of course, refactoring is more

effective as a prophylactic measure that as a last-restort therapy. As with any edifice, it is a judgement call, whether to rehab or restort for the wrecking ball.

Another alternative is to reassess whether new components and frameworks have come along that can replace all or part of the system. When you can reuse and retrofit other existing components, you can spare yourself the time and expense involved in rebuilding, repairing, and maintaining the one you have.

The United States Commerce Department defines *durable goods* as those that are designed to last for three years or more. This category traditionally applied to goods such as furniture, appliances, automobiles, and business machines. Ironically, as computer equipment is depreciating ever more quickly, it is increasingly our software artifacts, and not our hardware, that fulfill this criterion. Firmitas has come to the realm of bits and bytes.

Apple's Lisa Toolkit, and its successor, the Macintosh Toolbox, constitute one of the more intriguing examples of RECONSTRUCTION in the history of personal computing.

An architect's most useful tools are an eraser at the drafting board, and a wrecking bar at the site
-- Frank Lloyd Wright

❖ ❖ ❖

The SOFTWARE TECTONICS pattern discussed in [Foote & Yoder 1996] observes that if incremental change is deferred indefinitely, major upheaval may be the only alternative. [Foote & Yoder 1998a] explores the WINNING TEAM phenomenon, whereby otherwise superior technical solutions are overwhelmed by non-technical exigencies.

Brooks has eloquently observed that the most dangerous system an architect will ever design is his or her second system [Brooks 1995]. This is the notorious second-system effect. RECONSTRUCTION provides an opportunity for this misplaced hubris to exercise itself, so one must keep a wary eye open for it. Still, there are times when the best and only way to make a system better is to throw it away and start over. Indeed, one can do worse than to heed Brook's classic admonition that you should "plan to throw one away, you will anyway".

Mir reenters the atmosphere over Fiji on 22 March, 2001

Conclusion

In the end, software architecture is about how we distill experience into wisdom, and disseminate it. We think the patterns herein stand alongside other work regarding software architecture and evolution that we cited as we went along. Still, we do not consider these patterns to be anti-patterns. There are good reasons that good programmers build BIG BALLS OF MUD. It may well be that the economics of the software world are such that the market moves so fast that long term architectural ambitions are foolhardy, and that expedient, slash-and-burn, disposable programming is, in fact, a state-of-the-art strategy. The success of these approaches, in any case, is undeniable, and seals their pattern-hood.

People build BIG BALLS OF MUD because they work. In many domains, they are the only things that have been shown to work. Indeed, they work where loftier approaches have yet to demonstrate that they can compete.

It is not our purpose to condemn BIG BALLS OF MUD. Casual architecture is natural during the early stages of a system's evolution. The reader must surely suspect, however, that our hope is that we can aspire to do better. By recognizing the forces and pressures that lead to architectural malaise, and how and when they might be confronted, we hope to set the stage for the emergence of truly durable artifacts that can put architects in dominant positions for years to come. The key is to ensure that the system, its programmers, and, indeed the entire organization, *learn* about the domain, and the architectural opportunities looming within it, as the system grows and matures.

Periods of moderate disorder are a part of the ebb and flow of software evolution. As a master chef tolerates a messy kitchen, developers must not be afraid to get a little mud on their shoes as they explore new territory for the first time. Architectural insight is not the product of master plans, but of hard won experience. The software architects of yesteryear had little choice other than to apply the lessons they learned in successive drafts of their systems, since RECONSTRUCTION was often the only practical means they had of supplanting a mediocre system with a better one. Objects, frameworks, components, and refactoring tools provide us with another alternative. Objects present a medium for expressing our architectural ideas at a level between coarse-grained applications and components and low level code. Refactoring tools and techniques finally give us the means to cultivate these artifacts as they evolve, and capture these insights.

The onion-domed *Church of the Intercession of the Virgin on the Moat* in Moscow is one of Russia's most famous landmarks. It was built by Tsar Ivan IV just outside of the Kremlin walls in 1552 to commemorate Russia's victory over the Tatars at Kazan. The church is better known by it's nickname, St. Basil's. Ivan too is better known by his nickname "Ivan the Terrible". Legend has it that once the cathedral was completed, Ivan, ever true to his reputation, had the architects blinded, so that they could never build anything more beautiful. Alas, the state of software architecture today is such that few of us need fear for our eyesight.

Acknowledgments

A lot of people have striven to help us avoid turning this paper into an unintentional example of its central theme.

We are grateful first of all to the members of the University of Illinois Software Architecture Group, John Brant, Ian Chai, Ralph Johnson, Lewis Muir, Dragos Manolescu, Brian Marick, Eiji Nabika, John (Zhijiang) Han, Kevin Scheufele, Tim Ryan, Girish Maiya, Weerasak Wittawaskul, Alejandra Garrido, Peter Hatch, and Don Roberts, who commented on several drafts of this work over the last three years.

We'd like to also thank our tireless shepherd, Bobby Woolf, who trudged through the muck of several earlier versions of this paper.

Naturally, we'd like to acknowledge the members of our PLoP '97 Conference Writer's Workshop, Norm Kerth, Hans Rohnert, Clark Evans, Shai Ben-Yehuda, Lorraine Boyd, Alejandra Garrido, Dragos Manolescu, Gerard Meszaros, Kyle Brown, Ralph Johnson, and Klaus Renzel.

Lorrie Boyd provided some particularly poignant observations on scale, and the human cost of projects that fail.

UIUC Architecture professor Bill Rose provided some keen insights on the durability of housing stock, and history of the estrangement of architects from builders.

Thanks to Brad Appleton, Michael Beedle, Russ Hurlbut, and the rest of the people in the Chicago Patterns Group for their time, suggestions, and ruminations on reuse and reincarnation.

Thanks to Steve Berczuk and the members of the Boston Area Patterns Group for their review.

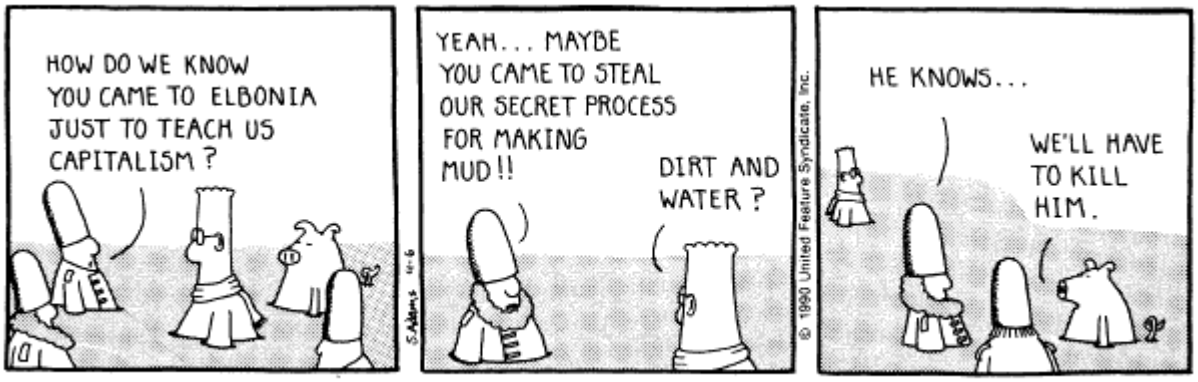
Thanks too to Joshua Kerievsky and the Design Patterns Study Group of New York City for their comments.

We'd like to express our gratitude as well to Paolo Cantoni, Chris Olufson, Sid Wright, John Liu, Martin Cohen, John Potter, Richard Helm, and James Noble of the Sydney Patterns Group, who workshopped this paper during the late winter, er, summer of early 1998.

John Vlissides, Neil Harrison, Hans Rohnert, James Coplien, and Ralph Johnson provided some particularly candid, incisive and useful criticism of some of the later drafts of the paper.

A number of readers have observed, over the years, that BIG BALL OF MUD has a certain dystopian,

Dilbert-esque quality to it. We are grateful to United Features Syndicate, Inc. for not having, as of yet, asked us to remove the following cartoon from the web-based version of BIG BALL OF MUD.



References

[Alexander 1964]
Christopher Alexander
Notes on the Synthesis of Form
Harvard University Press, Cambridge, MA, 1964

[Alexander 1979]
Christopher Alexander
The Timeless Way of Building
Oxford University Press, Oxford, UK, 1979

[Alexander et. al 1977]
C. Alexander, S. Ishikawa, and M. Silverstein
A Pattern Language
Oxford University Press, Oxford, UK, 1977

[Alexander 1988]
Christopher Alexander
The Oregon Experiment
Oxford University Press, Oxford, UK, 1988

[Bach 1997]
James Bach, Softwae Testing Labs
Good Enough Software: Beyond the Buzzword
IEEE Computer, August 1997

[Beck 1997]
Kent Beck
Smalltalk Best Practice Patterns
Prentice Hall, Upper Saddle River, NJ, 1997

[Beck & Cunningham 1989]
Kent Beck and Ward Cunningham
A Laboratory for Teaching Object-Oriented Thinking
OOPSLA '89 Proceedings
New Orleans, LA
October 1-6 1989, pages 1-6

[Beck 2000]
Kent Beck
Embracing Change: Extreme Programming Explained
Cambridge University Press, 2000

[Booch 1994]
Grady Booch
Object-Oriented Analysis and Design with Applications
Benjamin/Cummings, Redwood City, CA, 1994

[Brand 1994]
Stewart Brand
How Buildings Learn: What Happens After They're Built
Viking Press, 1994

[Brooks 1995]
Frederick P. Brooks, Jr.
The Mythical Man-Month (Anniversary Edition)
Addison-Wesley, Boston, MA, 1995

[Brown et al. 1998]
William J. Brown, Raphael C. Malveau,
Hays W. "Skip" McCormick III, and Thomas J. Mobray
Antipatterns: Refactoring, Software Architectures, and Projects in Crisis
Wiley Computer Publishing, John Wiley & Sons, Inc., 1998

[Buschmann et al. 1996]
Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stahl
Pattern-Oriented Software Architecture: A System of Patterns
John Wiley and Sons, 1996

[Coplien 1995]
James O. Coplien
A Generative Development-Process Pattern Language
First Conference on Pattern Languages of Programs (PLoP '94)
Monticello, Illinois, August 1994
Pattern Languages of Program Design
edited by James O. Coplien and Douglas C. Schmidt
Addison-Wesley, 1995

[Cunningham 1999a]
Ward Cunningham
Peter Principle of Programming
Portland Pattern Repository
13 August 1999
http://www.c2.com/cgi/wiki?PeterPrincipleProgramming

[Cunningham 1999b]
Ward Cunningham
The Most Complicated Thing that Could Possible Work
Portland Pattern Repository
13 August 1999
http://www.c2.com/cgi/wiki?TheMostComplexWhichCanBeMadeToWork

[Cusumano & Shelby 1995]
Michael A. Cusumano and Richard W. Shelby
Microsoft Secrets
The Free Press, New York, NY, 1995

[Foote 1988]
Brian Foote (Advisor: Ralph Johnson)

Designing to Facilitate Change with Object-Oriented Frameworks
Masters Thesis, 1988
Dept. of Computer Science,
University of Illinois at Urbana-Champaign

[Foote & Opdyke 1995]
Brian Foote and William F. Opdyke
Lifecycle and Refactoring Patterns that Support Evolution and Reuse
First Conference on Patterns Languages of Programs (PLoP '94)
Monticello, Illinois, August 1994
Pattern Languages of Program Design
edited by James O. Coplien and Douglas C. Schmidt
Addison-Wesley, 1995

This volume is part of the Addison-Wesley Software Patterns Series.

[Foote & Yoder 1996]
Brian Foote and Joseph W. Yoder
Evolution, Architecture, and Metamorphosis
Second Conference on Patterns Languages of Programs (PLoP '95)
Monticello, Illinois, September 1995
Pattern Languages of Program Design 2
edited by John M. Vlissides, James O. Coplien, and Norman L. Kerth
Addison-Wesley, 1996

This volume is part of the Addison-Wesley Software Patterns Series.

[Foote & Roberts 1998]
Brian Foote and Don Roberts
Lingua Franca
Fifth Conference on Patterns Languages of Programs (PLoP '98)
Monticello, Illinois, August 1998
Technical Report #WUCS-98-25 (PLoP '98/EuroPLoP '98), September 1998
Department of Computer Science, Washington University

[Foote & Yoder 1996]
Brian Foote and Joseph W. Yoder
Evolution, Architecture, and Metamorphosis
Second Conference on Patterns Languages of Programs (PLoP '95)
Monticello, Illinois, September 1995
Pattern Languages of Program Design 2
edited by John M. Vlissides, James O. Coplien, and Norman L. Kerth
Addison-Wesley, 1996

This volume is part of the Addison-Wesley Software Patterns Series.

[Foote & Yoder 1998a]
Brian Foote and Joseph W. Yoder
The Selfish Class
Third Conference on Patterns Languages of Programs (PLoP '96)
Monticello, Illinois, September 1996
Technical Report #WUCS-97-07, September 1996
Department of Computer Science, Washington University
Pattern Languages of Program Design 3
edited by Robert Martin, Dirk Riehle, and Frank Buschmann
Addison-Wesley, 1998
http://www.laputan.org

This volume is part of the Addison-Wesley Software Patterns Series. Brian also wrote an introduction for this volume.

[Foote & Yoder 1998b]
Brian Foote and Joseph W. Yoder
Metadata
Fifth Conference on Patterns Languages of Programs (PLoP '98)
Monticello, Illinois, August 1998
Technical Report #WUCS-98-25 (PLoP '98/EuroPLoP '98), September 1998
Department of Computer Science, Washington University

[Fowler 1999]
Martin Fowler
Refactoring: Improving the Design of Existing Code
Addison Wesley Longman, 1999

[Gabriel 1991]
Richard P. Gabriel
Lisp: Good News Bad News and How to Win Big
http://www.laputan.org/gabriel/worse-is-better.html

[Gabriel 1996]
Richard P. Gabriel
Patterns of Software: Tales from the Software Community
Oxford University Press, Oxford, UK, 1996
http://www.oup-usa.org/

[Gamma et al. 1995]
Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides
Design Patterns: Elements of Reusable Object-Oriented Software
Addison-Wesley Longman, Reading, MA, 1995

[Garlan & Shaw 1993]
David Garlan and Mary Shaw
An Introduction to Software Architecture
V. Ambriola and G. Totora, editors
Advances in Software Engineering and Knowledge Engineering, Vol 2.
Singapore: World Scientific Publishing, 1993, pp. 1-39

[Ingalls 1983]
Daniel H. H. Ingalls
The Evolution of the Smalltalk Virtual Machine
Smalltalk-80: Bits of History, Words of Advice
edited by Glenn Krasner
Addison-Wesley, 1983

[Johnson & Foote 1988]
Ralph Johnson and Brian Foote
Designing Reusable Classes
Journal of Object-Oriented Programming

Volume 1, Number 2, June/July 1988

[Marick 1995]
Brian Marick
The Craft of Software Testing
Prentice-Hall, Upper Saddle River, NJ, 1995

[Meszaros 1997]
Gerard Meszaros
Archi-Patterns: A Process Pattern Language for Defining Architectures
Fourth Conference on Pattern Languages of Programs (PLoP '97)
Monticello, Illinois, September 1997

[Roberts & Johnson 1998]
Don Roberts and Ralph E. Johnson
Evolve Frameworks into Domain-Specific Languages
Third Conference on Patterns Languages of Programs (PLoP '96)
Monticello, Illinois, September 1996
Technical Report #WUCS-97-07, September 1996
Department of Computer Science, Washington University
Pattern Languages of Program Design 3
edited by Robert Martin, Dirk Riehle, and Frank Buschmann
Addison-Wesley, 1998

[Shaw 1996]
Mary Shaw
Some Patterns for Software Architectures
Second Conference on Patterns Languages of Programs (PLoP '95)
Monticello, Illinois, September 1995
Pattern Languages of Program Design 2
edited by John M. Vlissides, James O. Coplien, and Norman L. Kerth
Addison-Wesley, 1996

[Simon 1969]
Herbert A. Simon
The Sciences of the Artificial
MIT Press, Cambridge, MA, 1969

[Swift 1726]
Johnathan Swift
Travels Into Several Remote Nations Of The World.
In four parts. By Lemuel Gulliver, First a Surgeon, and then a Captain of several Sh
B. Motte, London, 1726.

[Vitruvius 20 B.C.]
Marcus Vitruvius Pollio (60 B.C-20 B.C.)
De Architectura
translated by Joseph Gwilt
Priestley and Weale, London, 1826

Brian Foote foote@laputan.org
Last Modified: *27 August 2001*