Big Ball of Mud

Brian Foote
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

Monday, September 15, 1997

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 the most frequently deployed architecture: 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. In order to become so popular, it must be doing something 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 1995] [Garlan & Shaw 1996] [Buschmann et. al. 1996] 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. Their code shows 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 them. 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 six patterns:

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

These are not anti-patterns, at least in the customary sense. They are not straw men. Instead, they seek to examine the gap between what we preach and what we practice. They are 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 1997].

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. Like 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. 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.

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.

A number of forces can conspire to drive even the most architecturally conscientious organizations to produce BIG BALLS OF MUD. These "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 1994].

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.

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.

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.

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.

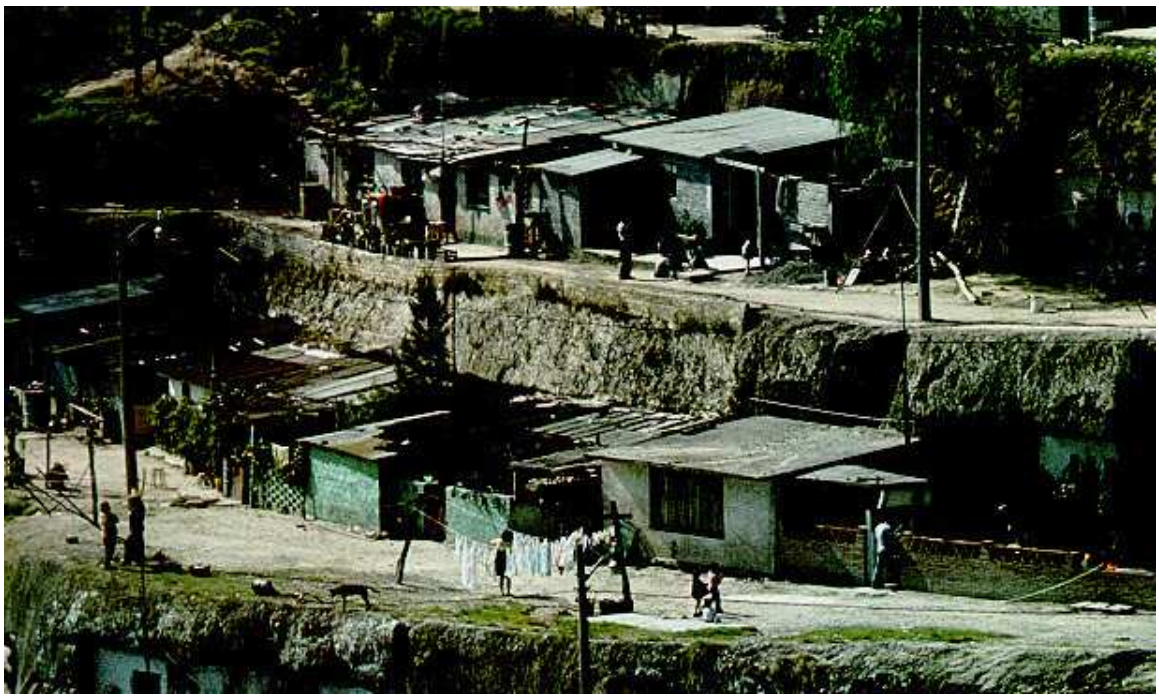
Change: Architecture is a hypothesis about the future that holds that subsequent change will be confined to that part of the design space encompassed by that architecture. Of course, the world has a way of mocking our attempts to make such predictions by tossing us the totally unexpected. A problem we 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.

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.

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.

***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 1993] 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. Every shred of important state data may be global. 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. 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.

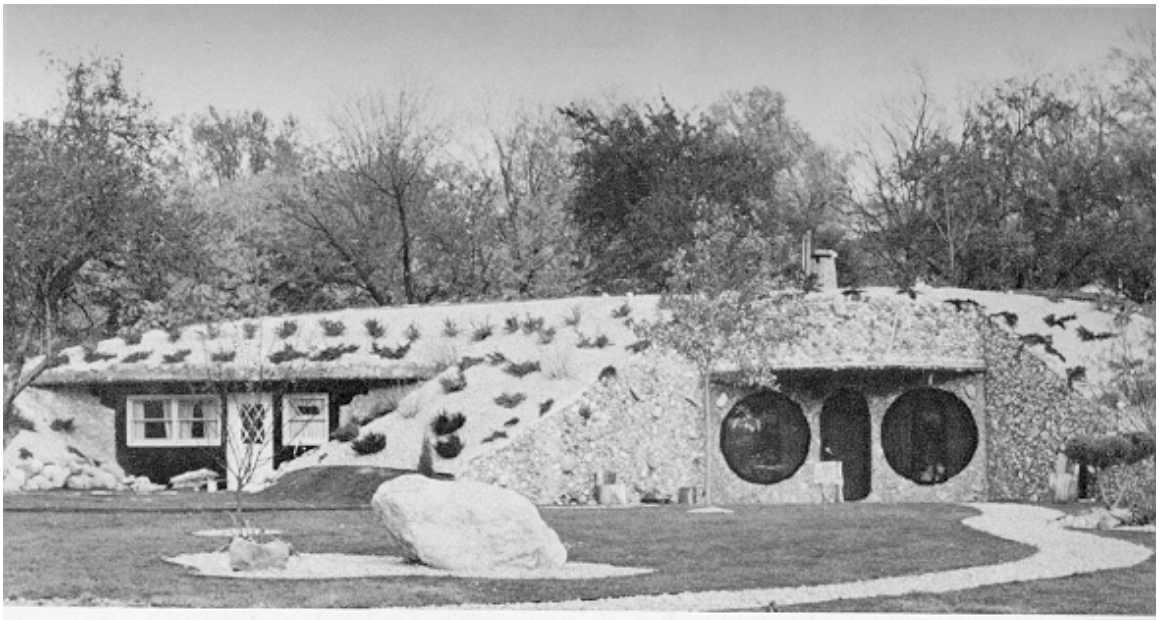
When it comes to software architecture, form follows function. The distinct identities of the system's architectural elements often don't start to emerge until after the code is working.

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 1997] 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.

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", and the observation that this was, perhaps, the dominant architecture currently deployed, during a meeting of the University of Illinois Patterns Discussion Group several years ago. We have been using the term ever since.

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. One must take care that this gradual process of repair doesn't erode its structure, or 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.

[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.

THROWAWAY CODE

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

❖ ❖ ❖

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.

❖ ❖ ❖

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 is, in effect, a distributed web-based application that runs on four different machines on two continents. Conference information is maintained on a machine in St. Louis, while registration records are kept on machines in Illinois and Germany. The system can generate web-based reports of registration activity, and now even instantaneously maintains 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 differ 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 `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.

PIECEMEAL GROWTH

alias
URBAN SPRAWL
ITERATIVE-INCREMENTAL DEVELOPMENT

The Russian Mir Space Station Complex was designed for reconfiguration and modular growth. The Core module was launched in 1986, and the Kvant and Kvant-2 modules joined the complex in 1987 and 1989. The Kristall module was added in 1990. The Spektr and shuttle Docking modules were added in 1995, the latter surely a development not anticipated in 1986. The station's final module, Priroda, 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/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.

❖ ❖ ❖

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

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 kitche*n 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.

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.

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

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 extendt 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. 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. Broad advances on all fronts are avoided. Instead, change is broken down into small, manageable chunks.

❖ ❖ ❖

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 1994]. 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

> 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 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, an so on.

❖ ❖ ❖

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

There may be times where taking a system down for a major overhaul can be justified, but usually, doing so is fraught with peril. 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.

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

There may be times where taking a system down for a major overhaul can be justified, but usually, doing so is fraught with peril. One 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.

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 is the past decade has to radically changed my own practice, 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.

❖ ❖ ❖

Always beginning with a working system helps to encourage PIECEMEAL GROWTH.

SWEEPING IT UNDER THE RUG

alias
POTEMPKIN VILLAGE
HOUSECLEANING
PRETTY FACE
QUARENTINE
HIDING IT UNDER THE BED

Reactor 4 Chernobyl NPS. Covered with Sarcophagus since accident in 1986.
0.96.07.02.17 DEC 1995
CHERNOBYL UKRAINE D
© Greenpeace/Shirley

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.

❖ ❖ ❖

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.

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.

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.

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.

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 1994]. [Foote & Yoder 1997] 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

also known as
TOTAL REWRITE
DEMOLITION
THROWAWAY THE FIRST ONE
START OVER

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

fcs_demolition_26sec.mov

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?

❖ ❖ ❖

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 had, given a steady infusion of

resources, buildings can last indefinitely. 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.

***Therefore*, throw it away it 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.

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.

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 these that guide the new implementation. With luck, these architectural insights can be embodied in 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.

❖ ❖ ❖

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 1997] 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]. RECONSTRUCTION provides an opportunity for this misplaced hubris to exercise itself, so one must keep a wary eye open for it.

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.

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 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.

Acknowledgements

A number of people have striven to help us avoid turning this paper into an unintentional example of its central theme. We are grateful to the members of the University of Illinois Patterns Group: John Brant, Ian Chai, Ralph Johnson, Lewis Muir, Dragos Manolescu, Brian Marick, Eiji Nabika, and Don Roberts. We'd like to also thank our tireless shepherd, Bobby Woolf, who trudged through the muck of several earlier versions of this paper. Finally, we'd like to acknowledge, in advance, our readers at our PLoP '97 Conference Writer's Workshop.

References

[Alexander 1979]
Christopher Alexander
The Timeless Way of Building
Oxford University Press, Oxford, UK, 1979
http://www.oup-usa.org/

[Alexander et. al 1977]
C Alexander, S. Ishikawa, and M. Silverstein
A Pattern Language
Oxford University Press, Oxford, UK, 1977
http://www.oup-usa.org/

[Alexander 1988]
Christopher Alexander
The Oregon Experiment
Oxford University Press, Oxford, UK, 1988
http://www.oup-usa.org/

[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

[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

[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
http://heg-school.awl.com/cseng/swpatterns/

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

[Foote 1998]
Brian Foote
Designing to Facilitate Change
with Object-Oriented Frameworks
Masters Thesis, 1988
University of Illinois at Urbana-Champaign
http://www-sal.cs.uiuc.edu/~foote

[Foote & Opdyke 1995]
Brian Foote and William F. Opdyke
Life-cycle and Refactoring Patterns
that Support Evolution and Reuse
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
http://heg-school.awl.com/cseng/swpatterns/

[Foote & Yoder 1996]
Brian Foote and Joseph Yoder
Architecture, Evolution, and Metamorphosis
Second Conference on Pattern
Languages of Programs (PLoP '95)
Monticello, Illinois, September 1995
Pattern Languages of Program Design 2
edited by John Vlissides, James O. Coplein,
and Norman L. Kerth.
Addison-Wesley, 1996
http://heg-school.awl.com/cseng/swpatterns/

[Foote & Yoder 1996]
Brian Foote and Joseph Yoder
The Selfish Class
Third Conference on Pattern
Languages of Programs (PLoP '96)
Monticello, Illinois, September 1996
Pattern Languages of Program Design 3
edited by Robert Martin, Dirk Riehle,
and Frank Buschmann
Addison-Wesley, 1997
http://heg-school.awl.com/cseng/swpatterns/

[Gabriel 1993]
Richard P. Gabriel
Lisp: Good News Bad News and How to Win Big
http://www.cs.washington.edu/homes/ctkwok/worseisbetter/worseisbetter.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, Reading, MA, 1995
http://www.awl.com/cp/Gamma.html

[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 E. Johnson and Brian Foote
Designing Reusable Classes
Journal of Object-Oriented Programming
Volume 1, Number 2, June/July 1988
pages 22-35
http://laputa.isdn.uiuc.edu/drc.html

[Roberts & Johnson 1996]
Don Roberts and Ralph E. Johnson
Evolve Frameworks into Domain-Specific
Languages
Third Conference on Pattern
Languages of Programs (PLoP '96)
Monticello, Illinois, September 1996
Pattern Languages of Program Design 3
edited by Robert Martin, Dirk Riehle,
and Frank Buschmann
Addison-Wesley, 1997
http://heg-school.awl.com/cseng/swpatterns/