Evolution, Architecture, and Metamorphosis

Chapter

13

by
Brian Foote - ***foote@cs.uiuc.edu -*** **(217) 333-3411**
Joseph Yoder - ***yoder@cs.uiuc.edu -*** **(217) 244-4695**
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield
Urbana, IL 61801 USA

Abstract

The dominant force driving software development in the 90's is the need to confront rapid change. Software that cannot adapt as requirements change will perish. This paper presents three patterns that address these forces. SOFTWARE TECTONICS shows how continuous evolution can prevent cataclysmic upheaval. FLEXIBLE FOUNDATIONS catalogs the need to construct systems out of stuff that can evolve along with them. METAMORPHOSIS shows how equipping systems with mechanisms that allow them to dynamically manipulate their environments can help them better integrate into these environments.

Introduction

This paper presents a trio of patterns that had their genesis in an unusual collaboration between the authors. The second author is involved in a project to investigate new approaches to software development for Caterpillar, one of the worlds largest manufacturers of heavy construction equipment, and, not incidentally, a major consumer and developer of software. The first author is involved in research on object-oriented reflection and metalevel architectures, an area that has earned for itself a not altogether undeserved reputation for abstruseness. This collaboration was suggested by Ralph Johnson, who noticed an unusual and interesting connection between what the Caterpillar group was doing to try to get beyond traditional approaches to software design and development, and some of the little-known findings coming out of the reflection community. This paper represents an effort to cast these commonalties as patterns.

The ultimate focus of the paper will be on two patterns that attempt to show how the forces that drive contemporary software development lead to more reflective systems. However, it is difficult to properly comprehend the forces that give rise to these patterns without setting them in the broader contexts of software reuse and evolution. As a result, this paper begins, in the Alexandrian tradition, with a high-level pattern, SOFTWARE TECTONICS, that pertains to evolution and reuse. It casts the need to cope with unrelenting change as one of the principal forces driving the software development process, and shows how this force can be dealt with. The second pattern, FLEXIBLE FOUNDATIONS, attempts to resolve some of the forces unleashed by the first by showing how to construct systems that can cope with change. The third pattern, METAMORPHOSIS, shows how the need for flexibility is omnipresent, and often can only be resolved dynamically. Our hope is that these patterns, taken together, will help the reader to perceive how objects, with their continuous, highly iterative lifecycles, encourage the emergence of the highly flexible and dynamic structural relationships that are characteristic of reflective architectures. We hope to show as well that these patterns are of genuine utility to real developers, and not mere academic curiosities.

It is becoming increasingly clear that software architectures evolve in ways that are distinct from other, more traditional forms of architecture. The pace of evolution in building styles can be measured in decades, or even centuries. The pace of software evolution is increasingly measured in months. The ubiquitousness of change is one of the most striking factors that distinguishes software architecture. The possibility of change is one of the things that gives software its power. The need to confront and accommodate it, is, therefore, an issue that every software designer must address.

Change pervades the lifecycle. Systems stop evolving only when they are no longer used. Indeed, the most volatile and interesting part of system's evolution frequently takes place during what was traditionally called the maintenance phase. Brad Cox [Cox 1986] observed the following about maintenance in 1986:

Software is not at all like wood or steel. Its paint does not chip and it does not rust or rot. Software does not need dusting, waxing, or cleaning. It often does have faults that do need attention, but this is not maintenance, but repair. Repair is fixing something that has been broken by tinkering with it or something that has been broken all along. Conversely, as the environment around software changes, energy must be expended to keep it current. This is not maintenance; holding steady to prevent decline. Evolution is changing to move ahead.

Objects support fine-grained, graceful evolution in a way that no other technology to-date does.

The paper presents examples drawn from the literature, and from our experience with the development and evolution of a significant application in an academic/industrial setting.

Caterpillar, Inc. joined the National Center for Supercomputing Applications at The University of Illinois as an Industrial Partner in December 1989. This partnership has spawned various projects, including an evaluation of supercomputers for analysis, and the investigation of virtual reality as a design tool.

The partnership with NCSA has provided Caterpillar with a glimpse of an approach to software development that is radically different from traditional approaches. This approach involves rapid application development through incremental prototyping and continual evolution.

The most recent Caterpillar project, the *Business Modeling project,* is a pilot project to demonstrate how an appropriate tool might support financial analysis and business decision making more effectively. This project aims to provide managers with a tool for making decisions about such aspects of the business as: financial decision making, market speculation, exchange rates prediction, engineering process modeling, and manufacturing methodologies. It is very important that this tool be flexible, dynamic, and be able to evolve along with business needs. Therefore, it must be constructed in such a way so as to facilitate change. It must also be able to coexist and dynamically cope with a variety of other applications, systems, and services.

The style in which these patterns are presented closely follows that used by Alexander [Alexander et. al 1977] in *A Pattern Language*. (In particular, this is where the diamond separators came from.)

SOFTWARE TECTONICS

also known as
EVOLVE OR DIE
EVOLUTION NOT REVOLUTION
GROW SOFTWARE, DON'T BUILD IT
PERPETUAL INCREMENTAL DEVELOPMENT

> *We like it when people always want more! Otherwise, we'd be out of the upgrade business. Sometimes, people ask me what I will do when the compiler is done. Done? No software program that is selling is ever done!*
>
> **Walter Bright,** C++ compiler architect

There are a variety of forces that drive software evolution. That software evolves in response to changing technology, and market forces is beyond dispute. However, the granularity at which software evolves can differ tremendously. A large, mature application may change slowly, if at all, only to be replaced by a more nimble successor. In these cases, the extinct application influences the designs of its successors only indirectly. At the other extreme, consider the notion that programs should be short, disposable artifacts, that can be produced so cheaply that they may be run once and thrown away. These two evolve only to the extent that they influence subsequent programmers.

This pattern considers the broad middle ground, wherein software artifacts themselves are durable enough to be cultivated over a long period of time. We believe that this encompasses the quick, disposable case above, since these programs must rely on an infrastructure of relatively high-level, reusable elements.

❖ ❖ ❖

Different people and organizations have different needs, and requirements change over time.

As software becomes increasingly complex, it can become more difficult to change. This ossification can become an obstacle to the system's evolution, and impede the system's ability to cope with changing requirements. The inability of the system to adapt to the changing needs of its users can cause strain to accumulate. Eventually, something must give.

It is becoming increasingly clear that the way in which software evolves today is at odds with the traditional, front-loaded, coarse-grained way of thinking about the process. Successful programs are no longer built from scratch, atop a simple programming language and a small runtime library. Instead, they draw heavily from existing code, components, frameworks, and applications. It is no longer enough for programmers to merely learn the language and runtime vocabularies underlying their development tools. Today's programmers must comprehend and comply with a variety of interfaces in order to integrate the work of others. These interfaces are frequently *moving targets*.

Successful systems face unrelenting pressure to change. These pressures come from defect repair, hardware evolution, operating system evolution, market competition, increasing user sophistication, etc. It is impossible to predict and cope with these forces in a front-loaded fashion. The system must be able to evolve to address these forces.

Traditional waterfall approaches to software development place the analysis, design, and implementation early in the lifecyle. This is followed by a lengthy maintenance phase. During the maintenance phase, a variety of activities occur. Bugs are repaired, and requests to accommodate new hardware or new features are serviced. After a while, a set of patches and enhancements may be bundled together as a new release. However, this phase is usually characterized by a gradual erosion of program structure. The following passage by Fred Brooks from *The Mythical Man-Month* [Brooks 1975] illustrates this inexorable decline:

All repairs tend to destroy the structure, to increase the entropy and disorder the system. Less and less effort is spend on fixing original design flaws; more and more is spent on fixing flaws introduced by earlier fixes. As time passes, the system becomes less and less well ordered....

Maintenance, it would seem, is like fixing holes in a failing dike. Eventually, it fails, and must be rebuilt. Only then can the lessons learned during it's tenure be exploited.

For nearly a generation, researchers in a number of quarters have promoted an alternative view of the software lifecycle. A number of these views came from the researchers in the object-oriented vanguard of the 1970s. One such view is the notion of *incremental perpetual development* proposed by Carl Hewitt [Hewitt 1977]:

The development of any large system (viewed as a society) having a long and useful life must be viewed as an incremental and evolutionary process. Development begins with specifications, plans, domain dependent knowledge, and scenarios for a large task. Attempts to use this information to create an implementation have the effect of causing revisions: additions, deletions, modifications, specializations, generalizations, etc.

Different people and organizations have different needs. One of the most difficult design challenges facing the software designer is how to balance the potential for generality with the need to confront a wide range of disparate individual concerns. An all too common approach to coping with individual needs is to simply force everyone to adapt to a single way of doing things. However, one size does <u>not</u> fit all. Therefore, it is better to take advantage of the malleability of software to allow it to be tailored to better meet individual needs.

Designing a system to meet the needs of a wide range of individuals or organizations can be an overwhelming task. It is better instead to provide ways by which individuals can customize their systems to address their specific requirements. Such systems might be said to be customizable or tailorable.

When faced with a system that almost, but not quite, meets one's needs, the availability of a customization mechanism can permit that system to be reused. The alternative might be to construct an entirely new system. Making a system tailorable can greatly increase its reuse potential.

Tailorability can operate at several levels. Users can customize their desktops, or provide shortcuts for commands they commonly use. Software architects can tailor existing abstract classes, frameworks, or components so that they precisely meet their needs. The mechanisms for achieving this can take several forms. A system that can be adapted to meet a designers needs via simple parameter manipulations might be thought of as an off-the-rack solution. In traditional systems, when such a perfect fit was not achieved, the designer might have to resort to a cut-and-paste job on the original code. This practice though initially effective, and (alas) still widespread, leads to a proliferation of sloppy, difficult to maintain copies of the original code. The proliferation of such expediently borrowed scraps of code was dubbed *metastisization* in [Foote 1988].

By using objects, designers can tailor existing code without disrupting the integrity of the original code. Instead, customizations can be made using user-specific subclasses via inheritance. This practice, though a great advance over slash-and-burn tactics, is not without its shortcomings. In particular, a good deal of knowledge is needed to use inheritance to wisely subclass existing objects. [Johnson & Foote 1988] called this practice white-box reuse.

An alternative is to specify the protocol for a component that is supplied to an existing framework as a *black-box*. The framework then calls the component back when its services are required. This approach has two benefits: First, the interface between the framework and the component is specified in terms of the component's public protocol. The designer need not know the internals of the existing code to design it. Second, any object that adheres to the framework/component protocol may be substituted for any other, even at runtime.

White-box, inheritance-based relationships can have a static, per-class quality, while black-box, component-based relationships can have a dynamic, per-instance character. We have observed that as a system matures, black-box component-based reuse supplants white-box reuse. Because components are the end product of this evolution, some designers are tempted to attempt to design components directly, and skip the evolutionary process. Components designed in this fashion are seldom reusable. Attempting to short-circuit the evolutionary process by designing components directly most often results in components that resemble first-pass prototypes, <u>not</u> the mature, truly reusable components that emerge from an evolving system.

***<u>Therefore</u>*, give people the ability to *tailor* their systems to meet their individual needs. Build systems that can *adapt* to change as requirements change. Allow systems to change in a series of <u>small</u>, <u>controlled</u> <u>steps</u>, in order to stay the potential upheaval that can result from change deferred.**

One size does not fit all. Allowing a system to be customized or tailored can broaden it's potential applicability and reuse potential. Building it to accommodate evolution can forestall premature obsolescence.

Seismologists have found that if tectonic plates release their energy in a series of small earthquakes, the strain that might have led to a major catastrophe is relieved. So it is with software as well. Systems that are permitted to evolve gracefully in a series of small, controlled stages can stay the seismic upheaval that can result from deferring change.

Not all software will be built to last. Disposable programs have their places. Simple tutorial prototypes, quick-and-dirty macros, and small, one-shot applications often do not, or cannot evolve beyond their initial incarnations. Even here, however, a substantial infrastructure of reusable elements needs to be present to facilitate the productions of such disposable code. This infrastructure, in turn, is most often the result of the sort of evolutionary process described herein.

❖ ❖ ❖

One way to keep a system flexible is to build it out of flexible materials. This is the FLEXIBLE FOUNDATIONS pattern.

FLEXIBLE FOUNDATIONS, in turn, encourages *refactoring*, which allows systems to confront and reverse the entropic pressures that Brooks warned against. The *Fractal Model* describes a set of evolutionary phases that embody this process. The heart of this process is a CONSOLIDATION PHASE, in which the system is refactored to better reflect structural insights that have accrued as it has evolved. [Foote & Opdyke 1994] describes a nascent pattern language, that encompasses this process, and the refactorings that drive it. The reader should refer that paper for information on these, and other, evolutionary and refactoring patterns that help to complete this pattern.

METAMORPHOSIS encourages the construction of systems that retain enough runtime mechanics to allow themselves to be dynamic instruments of their own evolution.

❖ ❖ ❖

The Business Model project is an excellent example of a system that was designed from the onset to cope with change. This project focuses on the use of object-oriented technology, and specifically on the development of object-oriented frameworks, as a key strategy for reusing code and design.

Why is it inevitable that the requirements placed on this tool will evolve, and desirable that the tool be able to cope with this change?

First, Caterpillar is a world-wide enterprise and has many different business units and marketing companies. In the mid-80's Caterpillar made the bold decision to decentralize and let all business units function with a certain degree of autonomy. Nonetheless, these units must share the same tools and databases. Therefore, the business modeling tool must be able to adapt to each business unit's needs, while remaining compatible with shared, company-wide resources. Also, as the business climate changes, the software needs to be able to keep pace.

To meet these requirements, our tool was developed around a object-oriented framework written in Smalltalk [Goldberg & Robson 1983]. Smalltalk allowed us to quickly develop working prototypes, and get immediate feedback from our users. Since VisualWorks is robust enough for production use, these prototypes were able to gracefully evolve into production applications.

Smalltalk is a pure object-oriented language that was chosen because of its extensibility, open-architecture, tailorability, and ease of reuse. The use of Smalltalk has lead to the development of a financial framework where key components have been re-used and integrated into all the financial applications developed for the different business units. The specific needs of the individual business units have been realized by either making "small" changes to this framework, or by adding new modules to the framework.

Systems that cannot cope with change will quickly be left behind by the marketplace. The same is true, of course, of corporations as well.

FLEXIBLE FOUNDATIONS

also known as
OPEN ARCHITECTURES
OPEN IMPLEMENTATIONS
GETTING UNDER THE HOOD
OBJECT-ORIENTED OBJECT-ORIENTED SYSTEMS
COEVOLUTION

Building software with FLEXIBLE FOUNDATIONS helps to resolve the need for continual, incremental evolution described by the SOFTWARE TECTONICS pattern.

❖ ❖ ❖

As systems confront changing requirements, they must change as well. Tools, languages, and frameworks which cannot change along with these systems will eventually become impediments to their evolution. Excessively rigid systems can be obstacles to their own evolution. It is not appropriate to expose the same face to every client.

Systems, tools, and languages that cannot themselves evolve can eventually become obstacles to the evolution of the systems that use them.

A good way to open the architecture up is to selectively expose the internal architecture of the system so that the elements of this architecture can serve as basis for changes and extensions. Note that the focus is on the substructure of the system, not on the source code itself. It is the architecture which is being opened up, not the entire implementation.

The views of a system that are exposed in this fashion are distinct from the primary public protocol through which the system is normally used. These views can be thought of as ways of getting under the hood, when necessary.

For instance, primary tasks such as variable definition or assignment, in the case of programming languages, or window definition, in the case of window systems, are usually considered base-level, rather than metalevel operations. However, facilities that allowed either a language or window system to be queried to determine how much memory it was using usually would be considered metalevel facilities.

Of course, a goal of those charged with shepherding the evolution of such a system should be to accommodate as wide a range of requirements as possible using the system's public interface. Modifications made via the reflective interfaces of a system may, if they are deemed to be of general interest, incorporated into the public interface. Some will be sufficiently exotic, mundane, or specific as to be not deemed worthy of general exposure.

The reflection community has garnered a not altogether undeserved reputation for abstruseness, due in large part to its penchant for producing dense prose, and to a certain self-important zeal it has displayed in coining exotic new terminology to attempt to exalt otherwise mundane self-referential architectural insights. *(The preceding is an attempt at constructing a self-referential sentence of sorts.)* What we hope to convey here is that a reflective object-oriented system is simply one which:

- is constructed from parts and tools that are also built from objects,
- and that has access to the objects that comprise these parts and tools.

To do this, it is useful to examine some of the traditional criteria for reflection. One premise of Smith's early reflection research [Smith 1983] was that a computational system is <u>about</u> something. For instance, an airline reservation system is about passengers, airplanes, arrivals, and departures, and an accounting system is about financial transactions of various sorts. A reflective system is one that has <u>itself</u> as its subject mater. The distinction between the model and the medium dissolves.

Maes [Maes 1987] identified three steps that one must take to make a computation system reflective:

I. Build a *self-representation* of the system.

II. Provide a means by which this self-representation may be <u>manipulated</u>.

III. Make sure such manipulations really do <u>immediately</u> <u>affect</u> the underlying system.

The third requirement enforces the so-called causal connection requirement. A system's self-representation is said to be causally connected to the system itself when any operation performed on this representation is indistinguishable in its effect from one performed directly on the system itself. In other words, the self-representation should either be the system, or should be implemented in such a way so as to make it impossible to tell that it isn't. Systems in which the self-representation actually comprises the implementation of the underlying system are said to be *procedurally reflective* systems. Systems in which this mechanism is less direct are called *declaratively reflective*.

***Therefore*, give tools, languages, or frameworks the ability to manipulate themselves. To do this, build these elements out of first-class objects.**

Another way to think of this is to consider that self-manipulation might be a good test of whether a system's design has the power and flexibility to permit graceful evolution. If the underpinnings of a system are built from well-designed first-class objects, they get this power more-or-less for free.

When a system and the substrates from which it is built, including its languages and tools, are all built from objects, variants of these substrates can coevolve with the system. When the architecture of these elements is open, their potential for reuse is greatly increased. This can help the programmer avoid duplication in cases where the underlying system elements almost, but not quite, meet his or her requirements [Kiczales 1994].

Providing flexibility of this sort in systems built of traditional, compiled code can be difficult. Where only such tools are available, the cost of providing flexible foundations may be prohibitive. Fortunately, modern object-oriented languages and environments provide viable alternatives to these antiquated approaches.

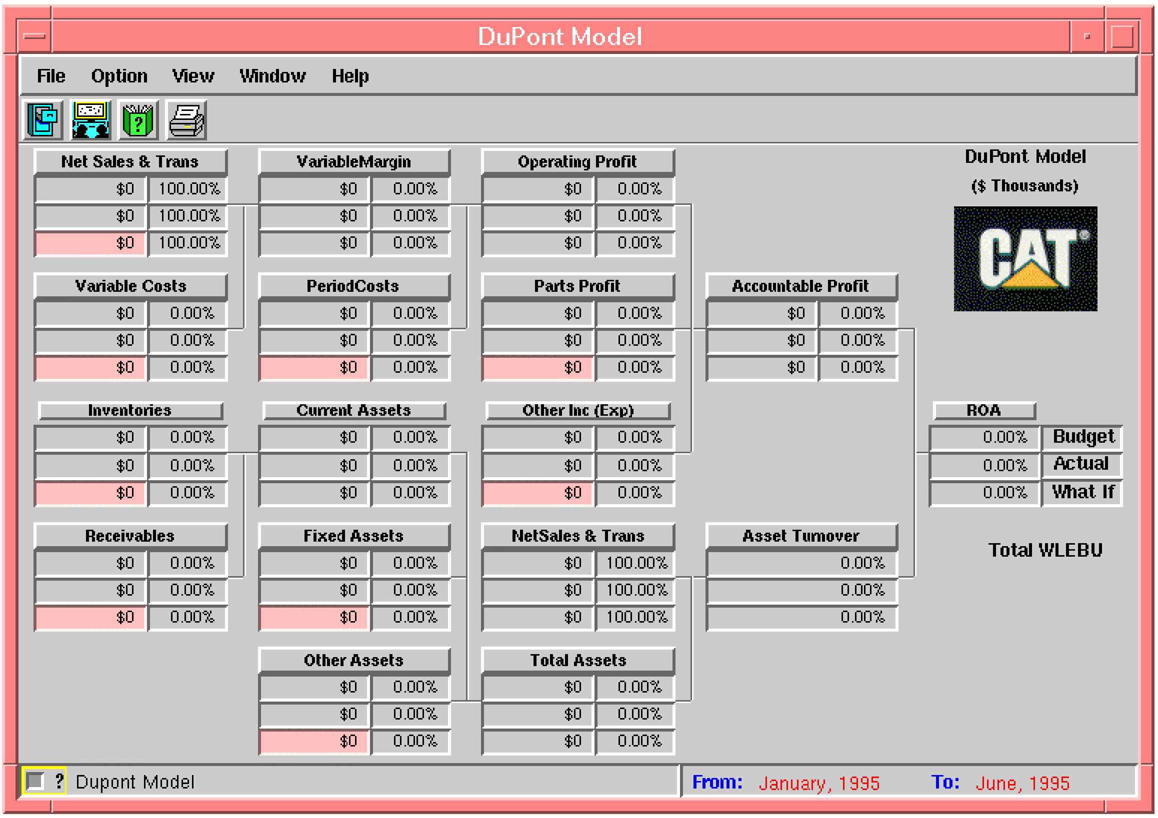
❖ ❖ ❖

VisualWorks is a powerful object-oriented development environment for graphical, client-server applications. It includes an application framework and visual interface builders to help design graphical user interfaces (GUI's).

Visual interface builders such as those provided by Smalltalk vendors today, let you quickly outline an interface as long as it is composed of basic widgets such as buttons, text-fields, menus, and scrolling fields. They generate empty methods that you can fill in later to invoke the desired behaviors. One can also develop more complicated interfaces by grouping these widgets together and ading in the appropriate behaviors, constraints, and tie-ins to databases.

Visual languages like this are an important feature that allows software developers to be more productive in not only the early development/prototype stage, but also in the production stage of code writing. The primary development of the financial aspect of the "Business Model" has been directed toward building or extending visual languages for the quick development of financial applications. Most of the code is automatically generated by the drawing of the interface and the data-flow of the program.

The **DuPont Model** [Johnson & Kaplan, 1987] (illustrated below) is a graphical model of a view of Profit/Loss statements for businesses. It provides a quick way for managers and accountants to view their return on assets.



As can be seen by the DuPont model example, their is a common interface widget that is used many times. By adding a "DuPont widget" to the visual builder with methods for the automatic generation of related code, the developer can quickly tailor different DuPont models to meet the needs of different users.

When building a DuPont Model as in the Caterpillar example, there are formulas and database queries that are associated with each graphical box on the screen. Normally this would require a lot of back end coding. The developer would first use the Visual Works Interface Builder to draw in all of the text fields and buttons and then add in the associated behaviors such as defining all of the queries/formulas along with all of the associated constraints. If this only needed to be done once, then it probably would be advantagous to develop the software this way. However, every business unit was looking at developing a DuPont model with minor tweaks which included different boxes, constraints, formulas, and database queries. This along with the fact that the data structure and layout will be somewhat different for each business unit, prompted us to extend the Visual Builder framework.

We were able to extend the Visual Builder framework to allow for the re-use of code and easy extension of the DuPont model by creating another interface widget like the DuPont box which has all of the fields and the associated button, the developer can then draw the DuPont boxes quickly on the screen and use the property editor along with the builder's automatic define method to automatically generate the formulas, the default query methods, and the needed constraints.

Many programming languages have been developed to provide developers with many built-in functions/capabilities/tools that allow for the quick and accurate development of software projects. Most often these languages are developed with specific needs in mind. If the software being developed maps easily into the domain of the language being used, then the developer can easily develop the desired software. However, most large development projects do not easily map directly into the limited domain provided by the programming language. VisualWorks is a language that allows the developer to modify the language to map to different domains.

Since VisualWorks allows one to get their hands on the insides of the visual builder, it opens up the system for easily adding a new interface widget. We usually think of the frameworks in VisualWorks as being the *View* and *ApplicationModel* subclasses. But the visual interface builder itself is comprised of a set of frameworks that can be extended. These frameworks provide the reflective interfaces that make it possible for the developer to easily extend the Visual Builder. Extending the visual interface builder is often the quickest way to make flexible and powerful software that maps to the domain.

The Smalltalk-80 system is constructed from a set of objects which are themselves subject to modification. The language, framework, and tools with which the system is built all reside in the default system image, and hence may be changed. These changes are usually done using abstract classes available within the Smalltalk image thus allowing for a lot of reuse.

METAMORPHOSIS

also known as
CHANGING THE RULES
CHANGING THE PERFORMANCE
DYNAMIC SCHEMA
DYNAMIC OBJECT INCORPORATION
DYNAMIC LANGUAGES
METAINFORMATION
LATE BINDING
REMOVING YOUR OWN APPENDIX

METAMORPHOSIS helps to resolve the forces that arise in evolving systems by providing a means by which a system's behavior can be augmented without changing its primary interface. These forces are described in the SOFTWARE TECTONICS pattern. Providing the means by which a system may dynamically extend itself also resolves the gradual evolution criteria of that pattern.

❖ ❖ ❖

It is difficult for statically compiled applications to manipulate objects that were unknown when an application was compiled. Sometimes it is necessary to augment or change a running system.

A *mutable* system is one in which the behavior of existing parts of the system can be changed. This is in contrast to an *extensible* system, which allows new elements to be added to a system, but does not allow the modification of existing parts.

One example of a mutable system is one where extensions to existing tools can be incorporated in the menus for those tools. One does not create a new tool. Instead one adds capabilities to the existing tool. In order to do this, the tool's menus must be mutable.

A mutable language allows the behavior of existing constructs to be changed. Debugging tools can make use of such facilities. Because of the potential for circularity, programmers must exercise extreme care when they change the way existing language constructs behave. Typical programming environments implement debugging facilities in an ad-hoc fashion beneath the language level. Hence, programmers cannot access and augment these facilities.

An extensible system allows users to add features. A mutable system allows users to change existing features.

Consider an analogy drawn from the theater world. There are two ways to change what happens on-stage during a play. The most direct way is to change the script. When that is not desirable, there is an alternative: you can change the performers, or the nature of the performance. Most readers will, I trust, concede that a performance of MacBeth might take on a different character with someone like Jerry Lewis rather than Sir Laurence Olivier in the title role, even though the script is unchanged. Similarly, productions of MacBeth in Kabuki-style have a decidedly different character than do traditional productions.

What has this to do with software? If one wants to change the way a program works, one could change its code or data directly. However, sometimes this is not desirable, or even possible. For, instance, assumptions about a subsystems primary interface may pervade the system. When this is the case, one can intervene indirectly, by changing the way that some underlying element of the system on which the application depends works.

One layer that underlies every system is the machinery associated with the programming language in which it is written. Were one interested in changing all the formatted I/O in a C program, one can either change every call to printf in one's program, or provide a new printf. However, other linguistic facilities are more difficult to modify. Languages that do provide full, runtime access to such mechanisms are said to be reflective.

This sort of relationship between an application and its substrates is not limited to the language layer. Facilities such as operating system calls, window systems, network services, mathematics libraries, or even other user written layers can all have the sort of relationship with an application such that these kinds of interventions are possible. When this is the case, and when this is the easiest place to get at these facilities, changing the performance rather than changing the script can be an effective way of solving otherwise intractable design problems.

Not only must systems provide a way for programmers to get under the hood, they must provide user-serviceable parts as well. What's more, they should ensure that these parts retain this serviceability in the field. For instance, languages without significant metalevel architectures such as C++ trap objects in binary object files. Once C++ programs are compiled, information pertaining to layout, object and class identity and the like is usually completely discarded. Some more modern systems have found it useful to retain some of this information in vendor-specific browsing and debugging structures. The growing movement to standardize runtime type information (RTTI) in the C++ community is evidence of a genuine need for metainformation. Some of these proposals go so far as to all but establish what are in effect first-class class objects in C++. This movement is driven not by linguistic purists, but by the requirements of real programmers in the field. There is frustration that often programmers must construct mechanisms themselves to reestablish information that the compiler knew in the first place, but threw away. A good sign that a linguistic facility is necessary is that a number of people go to a great deal of trouble to independently invent it. This would seem to be the case with metainformation. One can make the case that the extraordinary success of Visual Basic is due in part to that fact that the language is more reflective than C++.

Languages such as Smalltalk-80 and CLOS have what is in some ways the opposite problem. These languages provide fairly complete metalevel architectures, but usually imprison objects in snapshots or images.

For truly *autonomous objects* to break the umbilicals that tie them to single processes and images, the traditional division of responsibilities among system components must be refactored.

Autonomous objects must have access to global namespace services, so that they can find the other objects to which they are tied. Autonomous objects that interact with object-bases would benefit from the knowledge that truly first-class objects can glean about their own layouts.

For autonomous objects to function on platforms other than their home platforms, code must be bound to them at runtime. Smalltalk, Self, and Java provide code portability by defining code in terms of byte codes for a virtual machine. Dynamic translation of the sort found in Smalltalk-80 [Deutsch &

Schiffman 1984] and Self [Chambers et. al 1989] might be factored into the operating system, or provided as a runtime service, so that native code can be made available to any object on demand.

A vital factor in realizing autonomous objects is genuine first-classness. They will require a system-wide object model with a fully realized metalevel architecture, and not one based exclusively on v-tables.

A good sign that a programming language feature is needed is when a lot of people go to a great deal of effort to build it themselves atop or beside existing languages. There is abundant evidence that first-class, dynamic, metalevel objects are such a feature.

Most programming systems that support graphical user interfaces now support mapped, dynamic data structures called *resources*. These are usually cast at about the level of C structs. However, since they often are created and manipulated by using resource editing tools, they must usually employ their own conventions for manipulating what is, in effect, metainformation. Some realizations add unique symbolic objects that resemble Lisp **atoms**, and powerful, dynamic evaluators to allow runtime resource expressions to be processed. Often, elaborate schemes must be devised to set up runtime correspondences between names for routines in the resource namespace, and the same routines as they were known to the compiler and linker. The irony here is that all the facilities that have to be created in an ad-hoc, implementation specific fashion by the architects of these systems are essentially duplicating things that the original programming system knew how to do as well. In fact, the information that the programmer must redundantly recreate for these systems is often information that the compiler knew in the first place, but compiled away.

For similar reasons, many applications are adding, simple interpreted macro languages to their applications, while building these applications in a more powerful object-oriented language that by virtue of the system's architecture cannot be reused, and is hence out of reach.

Consider the difficulty one encounters in trying to construct a query for an object in an object-oriented database for an object one has not encountered before using C++. The only way to address this issue is to once again construct a dynamic language, with its own metalevel data structures, atop of C++.

The potential for Balkanization can be seen most acutely in current efforts to define object brokering services and object models. In many respects, these seem to be architectural end-runs around the linguistic community. This evasiveness is justified, given the degree to which mainstream language designers have avoided the issues these efforts are trying to address. In the end, it will be objects themselves, and not languages that will be the central focus of system design efforts.

***Therefore*, provide mechanisms so that the behavior of an object or system can be augmented, without changing fundamental interface or behavior. Systems that allow dynamic access to compilation facilities, or that allow late binding of the namespaces in which objects reside, can allow foreign objects to be incorporated into running applications at runtime.**

When both applications and their substrates are built from objects, they can evolve together as requirements evolve, or as specific users present specific needs. When the runtime mechanics of a system are thus accessible, that system can integrate better into a changing community of applications and services than a system that is set in concrete.

Metamorphosis is a powerful technique. However, many conventional systems will place a premium on support for these facilities, if they provide them at all. When they are not provided, runtime support for them will have to be provided by the user. Therefore, this technique should not be employed cavalierly. When objects can be redesigned so that their layouts are knowable in advance, such a redesign should be considered, and weighed against any loss of potential generality. In those cases where applications simply cannot be given prior knowledge of certain kinds of objects, metamorphosis can be the only viable solution.

❖ ❖ ❖

In order to stay competitive, Caterpillar has noticed that they must be able to quickly evolve to new ways of doing business. They need ways to be able to make new decisions quickly and change the way they do business according to those decisions. In order to do this they need to be able to dynamically choose their variables and business logic and then query from their data sources accordingly.

VisualWorks by ParcPlace has provided a framework for creating static SQL database queries. The framework allows for the developer to graphically create SQL queries that map to Oracle and Sysbase Databases. These queries then gets converted into a Smalltalk method that can be called upon when desired. Smalltalk objects can also be passed into the generated methods and conversions and comparisons are supported by the framework. This framework can also query the database for the current data model the developer is interested in and then create objects to map to the desired tables within the database. It is also easy to extend the framework to add undeveloped database functions or extend the mapping to other Database vendors.

Basically what happens is that the generated methods are parsed and SQL code is generated that includes the joins, projections, select-where, group-by, and order-by clauses. The generated SQL code is then packed up and shipped across the network via SQLNET. The returned database values are converted into objects that describe the attributes for each table within the database. These returned values can then be displayed, evaluated, or processed dynamically just like any Smalltalk object.

The problem arises when one wants to dynamically change or create SQL queries during run time. Since the SQL framework supplied by ParcPlace only provides ways to pre-defined static queries we built a framework of SQL Query objects that allows for the creation of dynamic SQL queries through the use of Smalltalk expressions. For example, the experienced Smalltalk Programmer might desire to be able to write code very similar that portrayed in **example1**.

```
example1

        "Perform a natural join on Models and ProductFamily
        projecting model number and family description.
        Then return the values."

   | models productFamily tmpQuery|
   models := TableQuery tableFor: #Models.
   productFamily := TableQuery tableFor: #ProductFamilies.
   tmpQuery := models naturalJoin: productFamily.
   tmpQuery := tmpQuery orderBy: (models fieldFor: 'modelNumber')).
   tmpQuery := tmpQuery project:(models fieldFor: 'modelNumber'),
               (productFamily fieldFor: 'familyDescription').
   ^tmpQuery values
```

Also, it might be nice to take a query that is formed as above and then "wrap" some new constraints on the query such as select only those in the above query for the current month. Adding new constraints to the queries such as this might only be realized during run-time, thus making the static creation of queries insufficient.

Our solution to allowing for the dynamic creation of SQL objects was to define GroupQuery, OrderQuery, ProjectQuery, SelectionQuery, and TableQuery classes which are all subclasses of the QueryObject abstract class. We also created QueryExpression objects that allow for the developer to build query expressions as in the example above.

The Query objects know how to respond to the appropriate message to build the queries and wrap constraints to themselves during run-time. The design pattern that fits here is the interpreter pattern from Design Patterns [Gamma et. al, 1995].

We were able to reuse all of the code from VisualWorks original framework of parsing a method into SQL and submitting the SQL across the net and then creating objects representing the desired values returned from the database.

Our dynamic SQL framework has allowed for late binding of constraints to the SQL objects by allowing the developer to build a parser for developing queries and "wrap" additional constraints to SQL objects as the application runs.

This example demonstrates the principles of reuse, extensions, and modifications. Reuse by the simple approach of blindly reusing the framework for generating the SQL code and using SQLNET to get the desired results and populate objects with them. Extensions are based upon the addition of the "Query Expression" objects for allowing the developer to write queries in Smalltalk like expressions and to also allow for the ease of extending these objects dynamically by wrapping additional constraints before the SQL code is generated. Modifications can be done to the existing framework by adding in behaviors for additional desired SQL functionality or the framework can also be extended by adding database drivers not supported in the default image provided by ParcPlace.

Smalltalk allows dynamic translation of the sort done in our SQL Object example. Our example parses a field of possible queries and simply wraps additional constraints, that the end user can create, around SQL Objects that are passed around. The dynamic translation is done through the simple parsing of strings that build the SQL Objects with the desired constraints and wraps them around the original SQL Object.

Conclusion

It is no longer possible to avoid the fact that to be successful, software has to be able to rapidly adapt to changing conditions and requirements. To cope with this fact, software researchers and developers must find new ways to confront the need for continual evolution. Software can no longer be designed up-front, and left to drift as the marketplace passes it by. Nor can it be set in concrete, unable to adapt to the differing needs of different people or organizations. Instead, software must be designed so that it can change along with the requirements that drive its evolution.

One way to do this is to build software out of objects. Objects can help an evolving system cope with change, by confining variants to subclasses using inheritance, and by promoting the emergence of abstract classes and frameworks. Because objects can be refactored, the emergence of new components and better, more reusable frameworks is promoted too. When a system is built from elements that are themselves objects, such as languages or tools, these objects can evolve along with the applications that use them, rather than presenting obstacles to such evolution.

Today, no application is an island. Today's applications live in a world where they must integrate with a variety of other objects, frameworks, services, and databases. Systems that can dynamically access the mechanisms with which they interact with the world can more effectively adapt to the environments in which they are embedded than those that cannot.

We were gratified to discover that the cross-pollination of Caterpillar's effort to cultivate new approaches to software development with the heretofore Laputian world of reflection has lead to what we think are genuinely useful, practical, and valuable ideas about how to build programs. In patterns, we think we've found the ideal medium for capturing and disseminating these ideas.

Acknowledgments

This collaboration was initiated by Ralph Johnson, who first noted the connection between what the second author's group was doing, and the often arcane claims of the reflection community. Professor Johnson also provided invaluable insights and observations as the paper progressed.

We are grateful as well to the members of the University of Illinois Smalltalk Group: John Brant, Michael Chung, and Donald Roberts, and Professor Johnson's Patterns seminar: Eric Scouten, Ron Absher, John McIntosh, Charles Herring, and Mark Kendrat, who soldiered through a particularly rough earlier draft of this paper, and provided a variety of commentary and advice, much of which was genuinely useful.

Desmond D'Souza shepherded our next, still unruly, draft through the PLoP '95 program committee.

Finally, we'd like to express our gratitude, in advance, to the PLoP '95 writers workshops participants, who, with their candid but constructive criticism, helped us shape and polish the current incarnations of these patterns.

References

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

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

[Brooks 1975]
Frederick P. Brooks
The Mythical Man-Month:
Essays on Software Engineering
Addison-Wesley, Reading MA, 1975

[Chambers et. al. 1989]
Craig Chambers, David Ungar, Elgin Lee
An Efficient Implementation of SELF
a Dynamically-Typed Object-Oriented
Language Based on Prototypes
OOPSLA '89 Proceedings
New Orleans, LA
October 1-6 1989, pages 49-70

[Cox 1986]
Brad Cox
Object-Oriented Programming:
An Evolutionary Approach
Addison-Wesley, 1986

[Deutsch & Schiffman 1984]
L. Peter Deutsch and Allan M. Schiffman
Efficient Implementation of the
Smalltalk-80 System
Proceedings of the Tenth Annual ACM Symposium
on Principles of Programming Languages,
1983, pages 297-302

[Foote 1988]
Brian Foote
Designing to Facilitate Change with
Object-Oriented Frameworks
Masters Thesis, 1988
University of Illinois at Urbana-Champaign

[Foote 1993a]
Brian Foote
A Fractal Model of the Lifecyle
of Reusable Objects
(abstract)

OOPSLA '93 Workshop on Process
Standards and Iteration
Washington, DC
Jim Coplein, Russel Winder, and
Susan Hutz, organizers

[Foote & Opdyke 1994]
Brian Foote and William F. Opdyke
Lifecycle 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

[Goldberg & Robson 1983]
Adele Goldberg and David Robson
Smalltalk-80:
The Language and its Implementation
Addison-Wesley, Reading, MA, 1983

[Hewitt 1977]
Carl Hewitt
Smalltalk-80:
Control Structures as Patterns
of Message Passing
Artificial Intelligence,
Vol. 8, pp. 323-363, 1977

[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

[Johnson & Kaplan 1986]
Ralph E. Johnson and Simon Kaplan
Towards Reusable Software
Designs and Implementations
Proceedings of the Workshop on
Future Directions in Computer
Architecture and Software
May 5-7, 1986, Seabrook Island, Charlston, SC

[Johnson & Kaplan 1987]
H. Thomas Johnson and Robert S. Kaplan
Relevance Lost:
The Rise and Fall of Management Accounting
Harvard Business School Press
Boston, MA, 1987

[Kiczales 1994
Gregor Kiczales
Why are Black Boxes so Hard to Reuse?
(Towards a New Model of Abstraction in
the Engineering of Software)
Invited Talk: OOPSLA '94
Portland, OR
ACM SIGPLAN Notices
Volume 29, Number 10, October 1994

[Maes 1987]
Pattie Maes
Computational Reflection
Artificial Intelligence Laboratory
Vrije Universiteit Brussel
Technical Report 87-2

[Smith 1983]
Brian Cantwell Smith
Reflection and Semantics in Lisp
Proceedings of the 1984 ACM
Principles of Programming Languages Conference
pages 23-35