Metadata and Active Object-Models

Brian Foote
Joseph Yoder

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

foote@cs.uiuc.edu (217) 333-3411
yoder@cs.uiuc.edu (217) 244-4695

Wednesday, 29 July 1998

Abstract

A number of forces shape the way in which software evolves. One is a desire to make programs as general as possible. Another is to push configuration decisions out into the data. Yet another is to push them out onto the users. Still another is to defer such decisions until runtime.

The patterns herein explore how complexity migrates from the code to the data as systems mature. As data become more sophisticated, the power that can, in turn, be brought to bear upon them at runtime increases.

This paper presents several patterns from a larger, emerging pattern language: PROPERTY, SMART VARIABLE, SCHEMA, and ACTIVE OBJECT-MODEL. It focuses on PROPERTIES, and observes that three distinct intents underlie what have commonly been called "properties"

Introduction

A number of forces shape the way in which software evolves. One is a desire to make programs as reusable as possible. Another is to push configuration decisions out into the data. Yet another is to push such decisions out onto the users. Still another is to defer such these decisions until runtime.

Data themselves become more universal and reusable when they are accompanied by descriptions of themselves that let other programs make sense of them. They can become even more independent when they are accompanied in their travels by code.

The patterns in our emerging pattern language begin to chronicle how domain specific languages emerge as programs evolve. A program may begin simply, performing but a single task. Later, programmers may broaden its utility by adding options and parameters. When more configuration information is needed, separate configuration files may emerge. As these become more complex, entries in these files may be connected to entities in the program using properties, dynamic variables, and dialogs. Simple values may not suffice. Once properties and dynamic values are present, simple parsers and expression analyzers often are added to the system. This, in turn creates a temptation to add control structures, assignment, and looping facilities to the functional vocabulary provided by the expression analyzer. These can flower into full-blown scripting facilities.

After a while, the domain or business objects come to constitute a program of sorts, which can be dynamically constructed and manipulated by users themselves. During this evolutionary process, descriptions of the data, such as maps of the layouts of data objects, and references to methods or code, are needed to permit these heretofore anonymous capabilities to be accessible during runtime. These descriptions allow these objects to be composed, edited, stored, imported, exported, and (these are programs, after all) debugged.

As this evolutionary process unfolds, and the architecture of a system matures, knowledge about the domain becomes embodied more and more by the *relationships among the objects* that model the domain, and less and less by logic hardwired into the code. Objects in such an ACTIVE OBJECT-MODEL are subject to runtime configuration and manipulation like any other data. Changes to this runtime constellation of objects constitute changes to the model, and to the operations that traverse or interpret it.

Data that describe other data, rather than aspects of the application domain itself, are called *metadata*. Naturally, these layout and code descriptions should be objects too. Hence, metadata have metadata as well.

A successful application inevitably draws a crowd. A host of users on a hosts of hosts will want to use such a program, and the data that go with it. It is important that data produced by one copy of the program be usable by other users at other sites. Such data might reside in a shared or distributed repository such as a database or persistent object base. They might also migrate across a network, via wires, satellites, fibers, radio waves, and even diskettes or tapes.

It is important, too, that these data be accessible not only from copies of the applications that spawned them. Other programs must be able to deal with them as well. When such data are mere "punch card images", or undifferentiated byte streams, this is hard to do. However, when data are escorted by machine readable descriptions of what they mean, they become welcome in a wider range of processing venues.

Our story then, is about how data earn their wings. It chronicles the forces that drive data to become more general. It describes their ascent from digits on punch cards, to lines on data files, and bytes in streams, through structures, and on through their marriage to behaviors, which begot objects. It continues as the need to describe these objects incubates self-descriptions, which themselves are cast as objects, which, in

turn, allow objects to aspire to escape the processes and images in which they were trapped, and roam unencumbered across the network.

The drive to become more general begins modestly. A simple application may acquire command line switches and parameters, to allow its behavior to vary, or permit additional input streams to be specified. As a program becomes yet more general, additional configuration information may be needed. This information may complex, and may even be provided interactively, by end users. Simple, textual interfaces may yield to graphical user interfaces, which themselves may grow more powerful, and, alas more complicated.

As an object-oriented application evolves, the elements of a object-oriented framework emerge. Where raw, undifferentiated, white-box code once was, dynamically pluggable black-box components begin to appear. Internal structure, which was once haphazard, becomes better differentiated, and more refined.

As such a framework evolves, the these elements themselves, together with the protocols and interfaces they expose, come to constitute a domain specific language for the framework's target domain.

Often, something else happens as well. The configuration user interface and tools grow more powerful too, so as to expose more and more flexibility and power to the users. At first, simple parameters are exposed. Later, expressions and simple logical rules may be proffered. Finally, control structures might emerge, and the full power of this emerging language is exposed to the user. Users may be offered existing behaviors, or new behaviors might be added using scripts which might be interpreted, or even compiled at runtime. Editors emerge that allow users to directly manipulate the objects that constitute their "programs".

This story might have a familiar ring to those readers who have followed the research done over the years into reflection and metalevel architectures. Of course, the reflection literature has earned it's recondite reputation the hard way (that is, through unrepentant abstruseness). Our tale might be seen as an attempt to render their *Finnigan's Wake* as, if not a *Mother Goose Tale*, at least a trip *Through the Looking Glass*.

The patterns in this paper are part of a larger pattern language that we are writing. We currently envision a language that will include the following patterns.

The patterns included in this PLoP '98 version of this work are shown in **bold**:

The patterns in this collection can be broken down into the following categories:

1. DATA
2. METADATA

Patterns that arise from *pushing decisions out onto the user*:

3. PARAMETERIZATION
4. CONFIGURATION
5. EXPRESSIONS
6. SCRIPTS
7. DIALOGS
8. TABLES

Patterns that arise as a *domain specific languages* emerges:

9. PROPERTY
10. SMART VARIABLES
11. SCHEMA / DESCRIPTOR
12. ACTIVE OBJECT-MODEL
13. SPECS

14. MESSAGE ROUTING
15. CONTEXT
16. NAMESPACES
17. EDITOR
18. VISUAL BUILDER
19. DYNAMIC VALIDATION
20. HISTORY
21. VALUE HOLDER / SMART VALUES

Patterns that become relevant as data become "*self aware*" (or more ***reflective***)

22. METACLASS
23. IDEMPOTENCE
24. SYNTHETIC CODE
25. CODE AS DATA
26. CAUSAL CONNECTION
27. BOOSTRAPPING

Global Forces

A variety of forces impinge upon evolving systems. Some of them pervade the patterns below, and are enumerated here to avoid duplication:

Portability: When an artifact works with a variety of applications, on a variety of platforms, it is more likely to be reused.

Efficiency: Highly dynamic systems can be inimical to efficiency. However, efficiency is often a false idol. For instance, the cost of referencing an object in a remote database may be several orders of magnitude more expensive than accessing a local object, and such overhead may overwhelm secondary concerns, such as the cost of accessors vs. direct variable references.

Complexity: Complex data structures and code are hard to debug and comprehend. Alas, many programmers are better at creating complexity than simplicity.

Dynamism: Interactive programming environments, visual builders and debuggers, and distributed applications all benefit from a more dynamic approach to software system architecture.

Dynamism can be dangerous, though. More dynamic systems can be harder to debug, maintain, and understand. One wouldn't let a child learn to ride a bicycle on a busy highway.

Resources: Dynamic strategies can be costly in terms of space, processing time, secondary storage, etc.

Safety: Dynamic strategies allow users to circumvent and undermine compile-time safeguards.

Flexibility: A program should be versatile, and usable in a variety of contexts. This, in turn enhances:

Reusability: A versatile, flexible application, or, for that matter, a code-level artifact, should be as reusable as possible. The reuse of such code avoids duplicated effort, eases the learning and comprehension burden of new programmers, and makes maintenance easier, since multiple, redundant copies of essentially the same code need not be maintained.

Adaptability: It is essential that an artifact be flexible enough so as to confront and address changing requirements. We distinguish several "shades" of adaptability.

Maintainability: It is important that an artifact be maintainable enough to as to confront and address changing requirements. Code that can't be worked on will lapse into stagnation.

Tailorability: One size does not fit all. Often, an artifact will not fit the needs of a particular user "off the rack", but can be tailored to do so when certain "alterations" can be made.

Customizability: Just an artifact can be tailored to a particular user or users, it can be customized to adapt it better for a particular task. This may seem at first to be a lot like tailorability, but we find that distinguishing between forces for change than emanate from individual users and those that arise from taking on different tasks useful.

Pushing Complexity into the Data: When complexity is pushed into the data, it can be coped with dynamically, at runtime. Configuration information can travel with the data, rather than being locked up in explicit code.

Pushing Configuration Decisions out onto the User: As a framework evolves, more and more configuration decisions are pushed out onto the user. Users become programmers of sorts. The trick, of course, is not to force them to be general purpose programmers. They don't have the training for this, and would fear that their social lives would be ruined. And, real programmers would be out of jobs.

Autonomy/Mobility: Once behavior and data, together with their descriptions, are liberated from application code, they can travel independently of these applications, and be used in a wider range of programs, on a wider range of platforms.

Comprehensibility: Metadata helps to document its associated data. Indeed, data files with metadata in them were often referred to as "self-documenting" data files during the '70s. Of course, the opposite can be true as well.

PROPERTY

also known as
ATTRIBUTES
ANNOTATIONS
DYNAMIC ATTRIBUTES
DYNAMIC VARIABLES
VARIABLE STATE
DYNAMIC SLOTS
PROPERTY LIST

❖ ❖ ❖

How do you allow individual objects to augment their state at runtime?

Image a system in which objects that track the assembly of products in a manufacturing shop are themselves routed through this system. The original designs for these objects might have focused on concerns such as part numbers and inventory information. New requirements might dictate that certain objects have a manufacturing routing slip attached to them as they move through the system. The original system made no provisions for such attachments. Once way to address this problem might be to add a new field for these routing slip attachments. However, there are several problems associated with this approach. One is that only a handful of instances will ever need such attachments, while the overhead cost for this field will be paid by every product object in the system. Another is that there may be a variety of these attachments. For instance, some products might have timestamp annotations made as they pass certain stations. We could add fields for all such annotations, but the costs and complexity would escalate rapidly. What we really want is a way to add a new variable to any object on-the-fly.

***Therefore*, provide runtime mechanisms for accessing, altering, adding, and removing properties or attributes at runtime.**

An implementation of the PROPERTY pattern will involve the following *participants*:

`Indicators`

These are the key or name values with which properties will be looked up. The name is taken from the original Lisp 1.5 implementation of property lists.

`Descriptors`

Objects that describe the attributes of a property. They may include display names, type information, the indicator objects, constraints, default values, and references to accessor functions.

`List`

Properties are usually stored in a random access data structure, such as a `Linked List`, `Dictionary` or `Hashtable`.

`Owner`

This dictionary is owned by the object that possesses the properties. Usually each *instance* of an object has its own property dictionary. However, an external data structure that maps instances or instance/indicator pairs might also be used.

`Client`

Clients, when transparent implementations of the PROPERTY pattern are used, can be unaware they are using PROPERTIES. More often, properties will be referenced using a different syntax than for normal variables. Also, clients must take particular care to cope with the consequences of a property's absence, since, most objects won't be carrying them.

`Value`

In dynamically typed languages, an object of any type will usually be permitted as the value of a property. Where type checking is present, downcasting from types like `Object` is usually used. Some implementations use `String` values as property values.

The following minimal set of operations on properties will usually be supplied in some form by object that have properties. These operations are generic, but are presented here using a Java-like syntax:

```
void addProperty(Indicator name,
      Descriptor descriptor, Object value);
void removeProperty(Indicator name);
boolean hasProperty(Indicator name);
void setProperty(Indicator name, Object value);
Object getProperty(Indicator name);
```

The `hasProperty()` will either be explictly or implicitly present. When it is not explictly present, a distinguished value such as `Property.ABSENT` might be returned by `getProperty()` and `setProperty()` to indicate the absence of a property, or an `Exception` might be generated.

Some implementations don't provide an explicit `addProperty()` operation, and allow the first call to `setProperty()` to create a new property instead. This is often the case when property `Attributes` are not present.

Similarly, the `removeProperty()` operation can be dispensed with by providing for removal of a property when a designated value is assigned to it, such as `Property.REMOVE`. This value, naturally, must be one that need never be the value of a `Property`.

One or more of the following additional operations might be present in some form as well:

```
Descriptor getDescriptor(Indicator name);
Descriptor[] getDescriptors();
Object[] propertyList();
```

The role, if any of the `Descriptor` objects, will vary depending upon the language and implementation strategy used. In dynamically typed languages such as CLOS, Smalltalk, or Self, they may not be present at all. In languages such as C++, Java, and C, minimal type information is might be used to indicate how different property value should be downcast. It is also used by tools such as editors, visual builders and debuggers.

Sometimes it is difficult to trace a pattern back to its origin. This is not the case with PROPERTIES. We can be quite definite as to where this idea first arose. Properties first appeared in MIT's early Lisp systems, and were described in the landmark *Lisp 1.5 Programmers Manual* [McCarthy et al. 1962].

Every atomic symbol in Lisp 1.5 had a property list. The first time a symbol was encountered, a property list was created for it. In Lisp 1.5, property lists began with a special sentinel value (-1). The rest of the list contained the properties themselves, as indicator/value pairs. These indicators, or property names, were themselves atoms. Some of the indicators used by Lisp 1.5 were:

PNAME	The print name of the atomic symbol for I/O
EXPR	An S-expression defining a function whose name is the atomic symbol on whose property list the atom appears
SUBR	Function defined by a machine language subroutine
APVAL	Permanent value for the atomic symbol considered as a value

Lisp 1.5 used these functions to reference property lists:

`define[x]`	Define one or more functions using the EXPR properties
`deflist[x;ind]`	Define one or more entries for property ind
`attrib[x;e]`	Add a property pair e to list x
`prop[x;y;u]`	Search x for y, and return the rest of the list, or u if not found
`get[x;y]`	Search x for y, and return the value
`remprop[x;ind]`	Remove a property ind from x

The pattern-hood of PROPERTIES was first suggested by Beck [Beck 1997] in his collection of *Smalltalk Best Practice Patterns*. He referred to this pattern as VARIABLE STATE. In Smalltalk, one can implement a simple property facility by adding a `Dictionary` or `IdentityDictionary` to an object's class, or one of its superclasses, and add methods like the ones below to allow the properties to be created and referenced. The keys for these `Dictionaries` will usually be `Symbols`, and values may be any `Object` whatsoever.

```
propertyAt: aSymbol
propertyAt: aSymbol put: anObject
```

A more ambitious implementation of this pattern was presented in [Foote 1988]. It used a number of Smalltalk's reflective facilities to allow properties to be referenced using the same external accessor syntax as was used for normal variables. This `AccessibleObject` facility added a new pair of classes, `AccessibleObject` and `AccessibleDictionary`, to allow dictionary-like access to objects, and object-like access to dictionaries.

Accessible objects allow dictionary-style access to all their instance variables, along with record-style access to a built in dictionary. Hence, instance variables can be accessed using `at:` and `at:put:`, as well as the standard record-style access protocol (`name` and `name:`).

Both access styles are provided without any need to explicitly define additional accessing methods. The record-style access method is rather slow however, and should be overridden when efficiency is an important consideration.

If `name:` or `at:put:` storage attempt is made and no instance variable with the given name exists, an entry is made for the given selector in the `AccessibleObject's` item dictionary. Thereafter, this soft instance variable may be accessed using either access method. In this way, uniform access to hard and soft fields is provided. `AccessibleObjects` provide a way of adding associations to objects in a manner similar to that provided by Lisp's property list mechanisms. Any instance of any subclass of `AccessibleObject`, which inherits from `Object`, may add such dynamic fields, and iterate over all its fields, including its regular instance variables.

The example below shows some of the capabilities of `AccessibleObjects`:

AccessibleObject class methods for: examples

example
"AccessibleObject example"

```
| temp |
temp ← AccessibleObject new.
temp dog: 'Fido'.
temp cat: 'Tabby'.
Transcript print: temp dog; cr.
Transcript print: temp items; cr.
temp keysDo: [:key | Transcript print: key; cr].
Transcript print: (temp variableAt: #items); cr.
Transcript endEntry
```

[Doble & Auer 1997] presented an implementation of a property-like facility that supports the accessor syntax for properties in a similar fashion that they called EXTENSIBLE ATTRIBUTES. They used a variation of PROPERTIES to build up scaffolding in the development environment. This scaffolding allowed them to dynamically add and remove variables as they learned what the classes needed and then they were able to GENERATE ACCESSORS which converted these dynamic attributes to normal accessors once the layout of the objects had been decided.

In C++, a *Standard Template Library* `map` might be used to implement the key/value pair mappings between indicators and values that are necessary to implement properties.

In Java, the *java.util* package provides a Property class that provides String indicator to String value mappings. Java uses it to provide access to system properties. Users can use it for any purpose they please. One noteworthy feature of Java's implementation is that each property object uses two hashtables: a main hashtable, from which the property object inherits, and a hashtable of default values, which it owns. The property accessors are designed to refer requests for keys that are not found in the main dictionary to the default dictionary. This is a simple use of the CHAIN OF RESPONSIBILITY pattern. All new properties are added to the main hashtable, so tables of defaults are never modified by property references, and hence can be shared.

This sample program illustrates the Property class in action:

```
import java.util.*;

public class PropertyTest
{
    public static void main(String args[])
    {
        //Get the system properties, and print them to stdout...
        Properties props = System.getProperties();
        props.list(System.out);

        //Create an default property list, and add a couple of keys...
        Properties defaults = new Properties();
        defaults.put("one", "one");
        defaults.put("two", "two");

        //Create a property list with our defaults…
        Properties test = new Properties(defaults);
        test.put("three","I'm a three");
        test.put("one","Override the one");

        //List dumps 'em all, and save just dumps the main list...
        test.list(System.out);
        test.save(System.out,"--Property Test--");
```

```
        //Let's remove one...
        test.remove("four");

        //Enumerate the names, and print each.
        //Unlike keys, propertyNames takes defaults into account...
        for (Enumeration e = test.propertyNames();
            e.hasMoreElements();)
        {
                String name = (String) e.nextElement();
                System.out.println("Key:  " + name);
                System.out.println("Get:  " + test.get(name));
                System.out.println("Prop: " + test.getProperty(name));
        }
    }
}
```

The first Lisp implementation of the PROPERTY pattern used linked lists (naturally) of indicator/value associations.

Most contemporary implementations use dictionaries or hashtables. An interesting variation on the hashtable approach was used in Objectiva [Anderson 1998]. It used the `Descriptor` objects themselves as Indicator look up keys.

When properties are extremely rare, the overhead of providing an additional field to store a `List` object can be avoided by storing a mapping between instances and their property lists elsewhere. This approach trades the additional runtime overhead of a second dictionary lookup for space, and, if the property accessors are implemented elsewhere, the need for a new subclass.

The use of the PROPERTY pattern can have a number of desirable *consequences*:

You avoid a proliferation of subclasses

Since fields may be added as needed on a per-instance basis, there is no need for a plethora of simple subclasses to add these fields. Where an arbitrary mix of such fields might be possible, creating and maintaining a mix of such subclasses may range from merely cumbersome to combinatorially impractical.

Fields may be added to individual instances

Since property lists are per-instance resources, each instance behaves like a lightweight, dynamic subclass as far as state is concerned.

Fields may be added and removed at runtime

There is no need to anticipate all the possible fields in advance. What's more, a field that is no longer need can be expunged.

You may iterate across the fields

Since properties are stored in random access data structures like dictionaries and hashtables, you may iterate over them using ENUMERATORS.

Metainformation is available to facilitate editing and debugging

Because properties use symbolic indicators that can be manipulated at runtime, property editors are easy to build.

Properties and their descriptors can serve a useful locus for validation, constraints, serialization, and editing.

Properties, in conjunction with SCHEMA objects and SMART VARIABLES, can allow programmers to build validators, constraint satisfiers, serializers, and editors to suit their needs. You can build variables your way. Nested namespaces, defaults, triggers, events, listeners, you-name-it ... you can build it.

Properties can graduate to first-class fields as an application evolves.

They are a finishing school for fields. If you find that most or all instances of a class add a particular property, promoting it to field status can be contemplated. Of course, you may still want to employ a DESCRIPTOR or SCHEMA to expose it at runtime.

Of course, PROPERTIES are not an unqualified plus. The following negative *consequences* may be encountered. Consult a metaphysician before using this pattern.

Syntax is more cumbersome in the absence of reflective support

Access to properties will normally use a different, more verbose syntax than normal variable references do.

Property access code is more complex that that for real fields

Property code must cope properly with indicators, dictionaries, and descriptors. Clients cannot depend on a fixed set of properties, and must test, or otherwise be prepared to deal with absent properties. The need to code for the possibility of absent properties can clutter your code as well. Where default mechanisms are not available, default selection must be coded by clients explicitly.

Reflective mechanisms, where they are available, can be slower

Mechanisms such as Smalltalk's `doesNotUnderstand:` mechanism, which can be used to trap unimplemented accessor messages and convert them into property references, are an order of magnitude or more slower than standard instance variable references.

Idiomatic implementations, when reflective support is not available, are also slow

Dictionaries and hash tables require hashing calculations and probes, which are slower than direct field references in most object-oriented languages.

Access to heterogeneous collections can be expensive

Property lists share the same disadvantages seen with other heterogeneous collections in typed languages such as C++ and Java. There is overhead associated with downcasting.

A field must be added to all objects, while only a few ever use it

There is the danger that many will be asked to pay a one field tax in storage overhead while few objects actually play the property game. Furthermore, inheriting from a property-enabled subclass can complicate the design of class hierarchy, particularly in systems without multiple inheritance. If this is an important consideration, an external map can be used to avoid this problem.

A tangle of properties is no substitute for an orderly factoring.

Properties are useful during the early stages of an applications evolution. There may be a temptation to use properties (as well as dynamic methods) as the basis for unrestricted prototype-style programming, of the sort seen in Self and ObjectLisp. A gaggle of properties that recur in recognizable clusters may be a good candidate for full object-hood. You should refactor such code to take advantage of such opportunities.

Properties are effective tools for exploring the design space early in a design's evolution. They are also an effective way of coping with the occasional need for lightweight, per-instance annotations. They should be used sparingly, though. They are no substitute for a well-factored design.

The following are but a handful of the known systems that use properties.

One such systems is the Caterpillar Financial Modeling Framework (http://www-cat.ncsa.uiuc.edu/~yoder/financial_framework/).

Three others were discussed at the 1998 UIUC Metadata Workshop (http://www-cat.ncsa.uiuc.edu/~yoder/Research/metadata/UoI98MetadataWkshop.html).

These were:

1. Hartford Insurance Framework by Jeff Oakes
2. Objectiva Telephone Billing System by Francis Anderson
3. Argo Belgium School System by Michel Tilman

❖ ❖ ❖

Not only does the notion of PROPERTY have a long history, but it casts a wide shadow. The name "property" has been used to describe *three distinct intents*. Each of these is described herein as a separate pattern. These intents, and the corresponding patterns are:

PROPERTY	You want to add and remove attributes on a per-instance basis at runtime
SMART VARIABLE	You want to augment the behavior of variable references and assignment, to implement constraints, listeners, etc.
SCHEMA	You want a map of your variables so that you can enumerate them, manipulate them en masse, and reference them indirectly, using symbolic names

Properties can be used in conjunction with the CHAIN OF RESPONSIBILITY to build prototypes and namespaces.

PROPERTIES can, and often do, use METADATA.

Beck's VARIABLE STATE pattern is a variant of the PROPERTY pattern.

Doble and Auer's EXTENDED ATTRIBUTES pattern is another variant of the PROPERTY pattern. It emphasizes the dynamic creation of attributes during development. These are stripped away before the final version of an application is deployed.

PROPERTY LIST has also been nominated for pattern-hood in [Riehle 1997], [Sommerlad 1997] and in early drafts of [Sommerlad & Rüedi 1998].

SMART VARIABLES

also known as
ACTIVE VARIABLES
SLOTS

Now I have to display my true schizophrenia. Having convinced you in **Direct Variable Access** that just using variables is good enough, I'm going to ask you to ignore that crazy bastard and listen to me talk some sense here.
Kent Beck, in rebuttal to **Kent Beck**
Indirect Variable Access
Smalltalk Best Practice Patterns, p. 91

❖ ❖ ❖

The VisualWorks Smalltalk GUI associates editable attributes with most of the widgets it displays on the screen. Whenever one of these values changes, you want the displayed value to reflect that change. One way to do this is to place an object called a `ValueHolder` in the application variable that would usually house the value being displayed. This object is accessed using a special `value/value:` accessing protocol. It houses the value, and also contains code to notify the GUI when a change is made.

If you ignore the additional protocol, these value holders are really playing the role of "smart variables", that not only house values, but perform additional chores when these values are referenced or changed.

How do you allow programmers to control the effects of references to their variables?

Objects are abstractions that bundle together a clump of *state* and a collection of *functions* or *behaviors* that operate on this state. The upkeep of this state information is one an object's fundamental duties. As such, *references* to this state information, and assignment to these variables, are critical "choke points" in any object. Indeed, without its state memory, an object's behavioral repertoire is nothing more than a collection of stand alone-functions. Because state is such an important part of what an object is about, variable references and assignments frequently attract additional responsibilities.

As object's evolve, the semantic burden placed on elements of its representation can grow. An assignment to a variable indicating an employee's age may require not only that this value be preserved in memory, but that it be stored safely away in a database, and that it appear as part of a form in a runtime display of employee data.

A variety of chores will naturally be tied to changes in state, or dependent on notification that a variable has been referenced. For this reason, if you can go to a single place in your program to control what happens whenever such references or assignments are made, your job is simplified.

***Therefore*, provide a means for intercepting references to variables.**

Among the ancillary duties that might be tied to state activity are:

Dependent Notification: Programmers frequently want to ensure that notifications to dependents are made when an object's state is modified. Indeed, this is the essence of the OBSERVER [Gamma et al. 1995] pattern.

Persistence: You might what to keep track of assignments to an object's state to keep a persistent representation of it up-to-date.

Distribution: You likewise might want ensure that a change to an object propagates changes in its state to remote copies.

Caching: A remote copy of an object might cache current copies of its representation, to avoid the overhead of referencing a master version every time a value is requested of it.

Constraint Satisfaction: Assignments will often call code to test compliance with constraints, which may veto or modify the assignment.

Synchronization: Since an object's state is itself a resource that must be protected from synchronization problems, such constraints are often best handled using critical sections tied to variable references.

Despite the importance of variable reference and assignment, very few languages provide the necessary reflective mechanisms to allow programs to intercept variable access.

The most powerful facilities for gaining control over variables are found in the Lisp World. For instance, InterLisp provided hooks to allow code to be run when variables were read and written. The Common Lisp Object System's Metaobject Protocol [Kiczales et al. 1991] treats an object's instance and class variables as instances of `SLOT-DESCRIPTION` metaobjects. The metaobject protocol is designed so that all references to an instance's *slots*, or variables, are made through accessor methods defined in object's metaclass and in that object's slot descriptions. As a result, classes may elect to use variables that adhere to rules they define themselves. In particular, they can override their accessors to incorporate whatever additional behavior they may deem fit.

Smalltalk represents class, pool, and global variables as `Association` or `VariableBinding` objects. As such, references in compiled byte code retrieve values by sending a value message to these objects at runtime. Kent Beck has noted that the clever programmer can substitute other objects that conform to this protocol to wrap whatever behavior he or she wishes around such references. [Beck 1993]

Smalltalk does not normally provide a means to intercept instance variable references. However, here to, the ubiquitous Beck's fingerprints can be found. [Messick & Beck 1985] describes a scheme that exploited the Smalltalk `Compiler` to introduce what they called *Active Variables*. These variables were declared much like normal instance variables, but permitted the introduction of *daemon* methods that were run when they were accessed.

Alas, such facilities are rare. The vast majority of programmers who want to intercept variable references are deprived of such language support. Instead, they must resort to ***idiomatic*** approaches to implement the SMART VARIABLE pattern's intent.

The dominant idiom for implementing SMART VARIABLES is to employ *ACCESSOR METHODS*, or *GETTERS* and *SETTERS*.

If all references to your variables, including public references, private references, and everything in between, go through such accessor functions, then this idiom will effectively realize the pattern's intent. Achieving such accessor hygiene can require a degree of effort that can range from trivial to tedious, depending on the language, programming tools, and objects involved.

For instance, in CLOS [Bobrow et al. 1988], every slot reference is made through an accessor function, and achieving this intent is simple. In Smalltalk, only an object's own methods, and those of its subclasses, can reference its instance variables. Hence, enforcing discipline on external references via methods is a necessity in Smalltalk. Of course, any method may change any variable, so some internal discipline is required. If all references, even internal ones, go through accessor methods, the single point of reference/change requirement of our idiom can be maintained. However, since Smalltalk does not

distinguish private and public methods (except by convention) these accessors effectively make the state that they protect visible to the entire world.

C++ [Stroustrop 1991] and Java [Chan et al. 1998] allow both instance variables, and their accessors to be declared as `public`, `protected`, or `private`. (Java and C++ both have per-file "package" scope peculiarities as well). Indeed, Java beans "properties" depend on accessor functions to tie bean editors and beans together at runtime.

It may seem to the reader that accessors are idiom enough for most SMART VARIABLE tasks. Do we really need separate objects? Is it not enough to code the additional chores that must be performed when a variable is read or written in the accessor methods for it?

Up to a point, this is so. In simple cases, hand coding smart accessors will suffice. However, at what point does managing these accessors become tedious. Were one to want to change a few dozen variables wholesale to hook them into a metering or diagnostic regimen, would one prefer to do this by hand, or en masse, via a change to a single, per-class definition? This is where separate variable objects can help.

❖ ❖ ❖

SMART VARIABLES can work together with SCHEMA / DESCRIPTOR objects to allow the code that coordinates references, assignments, and the activities that depend on them to be reused. These objects can have their own hierarchies. Indeed, a family of such stock variable classes might evolve to meet a range of SMART VARIABLE requirements.

SMART VARIABLES are frequently asked to handle OBSERVER notifications.

You may want to add SMART VARIABLES to attributes that have been added through PROPERTIES.

METADATA can be used to help implement SMART VARIABLES.

Wrappers to the Rescue [Brant et al. 1998] discusses how accessors methods themselves might be wrapped to allow additional behavior to be associated with them.

QUERY OBJECTS [Brant and Yoder 1996] use SMART VARIABLES to make sure query dependencies are properly propagated.

The Palette System [Golin 19xx] uses CLOS `:before` and `:after` methods to generate automatic notifications upon variable reads and writes .

SCHEMA

also known as
DESCRIPTOR
MAP
DATABASE SCHEME
LAYOUT

❖ ❖ ❖

Information tied to the layout of particular objects or data structures is all too often buried in source code, where it is difficult to comprehend and change. As a result, adding new functionality, such as a graphical editor, for such objects can require painstaking, object specific, thankless work. This can be particularly galling when the coding task that needs to be done varies only in the specific details of how these objects are laid out. If only this information were itself data, general routines to exploit these maps could allow many chores to be coded once and for all.

How do you avoid hard-wiring the layouts of structures into your code?
How do you describe the layout of a structure, object, or database row?

A number of forces encourage the emergence of layout metadata.

Once is the inexorable evolution of a system's objects themselves. The sort of code that will use layout information, such as GUI code, is precisely the sort of code that tends to be tightly coupled to layout decisions otherwise. By reading layout information from a map, a single version of the code performs layout specific chores for all the objects that use it. Problems with this code should become evident quickly, since flaws will affect all the applications that use it.

By contrast, writing handcrafted code for each object is an error prone endeavor.

Hand-written code has some advantages. It is usually relatively efficient, and can be straightforward as well. Code to read metadata can be slow and complex.

***Therefore*, make a schema or map describing your data structures available at runtime.**

The following *participants* come can come into play:

`Schema`

The schema itself is logically a set of descriptors. In the simplest cases, it can be implemented as a set, array, hashtable, or collection of `Descriptors`. However, in some systems, this *role* will be played by more elaborate or general data structures. For instance, in languages that support first-class `Class` objects at runtime, these objects will play the role of Schemas along with fulfilling their other duties. A class, together with its superclasses, may be looked upon as a compound schema that use the CHAIN OF RESPONSIBILITY pattern to serve up descriptors.

`Descriptor`

These objects describe the layout of each element of a schema. Often, they will provide additional *attributes*, such as display names, constraint hooks, type information, default values, access flags, etc. However, in the simplest cases, the Descriptor may only supply the element's symbolic name.

`Subject`

The `Subjects` are the objects being mapped by a SCHEMA. In class-based object -oriented languages, all the instances of a class will have the same layout, and hence can use the same map. In prototype-based languages that support dynamic slots, schema may be more complex, per-instance entities that map a single instance, or a handful of single instances.

`Grapples`

A schema must have a way to map from symbolic references to actual objects. These references are not direct. Instead, they are used to construct "grapples" that let the actual `Subject` be manipulated indirectly. In C or C++, these might be indices or offsets that provide the grist for a brief foray into unsafe pointer arithmetic. In Smalltalk, these might be blocks, or selectors that can be sent as messages using `perform:`. They might even be `CompiledMethods`. In Java, `Method` and `Field` objects might play this role

`Attributes`

Often, designers seem to design descriptors as if they were thinking "as long as I'm reinventing variables, I'll add a few things I've always wanted while I'm at it." Descriptor attributes may include type information, size information, constraints, access information, presentation information, support for debuggers and editors, and the like.

`Client`

Code that employs schema objects to indirectly manipulate the objects they describe is usually complicated, since the resources to make such calls must be assembled at the call sites.

Database systems usually allow programmers to retrieve database schema meta-information at runtime. This information can be used to build editors, forms, and accessors to map objects to databases.

Class-based object oriented languages usually roll the responsibility for exposing layout information at runtime into their `Class` objects. In Smalltalk and Java, Classes can be asked to supply information about their variables. In Smalltalk, these are be simple lists of names. In CLOS, they are `SLOT-DESCRIPTION` metaobjects. This layout information has been used over the years to support a variety of features, such as distributed marshalling, and persistence [Paepcke 1989].

In Java, the *java.beans* package supports a family of `Descriptor` objects that work in conjunction with the Beans `Introspector` and editors to allow components to be assembled dynamically.

Beans supplies `FeatureDescriptors`, `PropertyDescriptors`, `IndexedPropertyDescriptors`, `BeanDescriptors`, `EventSetDescriptors`, `MethodDescriptors`, and `ParameterDescriptors`.

Beans is interesting because its `Descriptors` contain additional functionality to help support SMART VARIABLES. For instance, a `PropertyDescriptor` permits constraints, display name, short descriptions, and custom editors, as well as hidden and expert flags, to be supplied for each `Subject`.

When `Descriptors` are not supplied explicitly by programmers, Beans uses *introspection* to construct default `Descriptors`. The `Introspector`, in turn, uses the `Field` and `Method` descriptors that are found in Java's `Class` objects.

Object brokers, such as CORBA, and Microsoft's COM, provide API's that supply runtime schema information.

❖ ❖ ❖

METADATA is often used to describe SCHEMAS.

The use of a SCHEMA greatly simplifies the implementation of the SERIALIZER [Riehle et al. 1998] pattern..

SCHEMAS can be used in conjunction with SPECS, and a present in some form in many GUI systems.

ACTIVE OBJECT-MODEL

also known as
DYNAMIC OBJECT-MODEL
RUNTIME INTERPRETER
LIVE OBJECT-MODEL
DYNAMIC PROGRAM
DYNAMIC BUSINESS RULES
PROGRAM TREE

> Inside every domain-specific framework, there is a language crying to get out.
> *Thomas Jay Peckish II*

❖ ❖ ❖

An ACTIVE OBJECT-MODELS is an object model that provides "meta" information about itself so that it can be changed at runtime. ACTIVE OBJECT-MODELS usually arise as domain-specific frameworks evolve to address an ever widening range of domain-specific needs. Ultimately these models can become general enough to span several domains (for example, think of a graphing framework that originated in one domain but then was enhanced so that it could be used by any application needing graphs).

Being dynamic and configurable allows tools to be developed to allow decision makers and administrators to introduce new products and changes to their business models at runtime. This can reduce time-to-market of new ideas from months to days, if not hours. It can place the power to customize the system in the hands of those who have the business knowledge to do it effectively.

How do you let your users build programs without "programming"? How do you let your users customize and change the behavior of what they do at run-time?

Some issues that arise are:

- Both systems and their users must adapt quickly to changing requirements .
- Building Dynamic Objects is hard.
- Once built, Dynamic Objects allow for rapid alterations to your program.
- You can "program" without programming.
- Changing a program to meet new business requirements is usually slow and complicated.
- Users want the ability to change what they do on-the-fly.
- ACTIVE OBJECT-MODELS can be difficult to develop, hard to understand, and hard to maintain.

***Therefore*, develop an** ACTIVE OBJECT-MODELS **that can define the objects, their states, the events, and the conditions under which the objects changes state. Also include editors and other tools to assist with developing and manipulating the object model.**

A system with an ACTIVE OBJECT-MODELS has an explicit object model that it interprets at run-time. If you change the object model, the system changes its behavior. For example, a lot of workflow systems have an ACTIVE OBJECT-MODELS . These objects have states and respond to events by changing state. The ACTIVE OBJECT-MODELS defines the objects, their states, the events, and the conditions under which an object changes state. Suitably privileged people can change this object model "without programming". Or are they programming after all? Business rules can be stored in an ACTIVE OBJECT-MODELS . This makes it easy to change the way a company models its business.

Building a new software product typically requires dedicated software development and support. This can take time. When a simple modification to a business rules requires the mobilization of a platoon of

programmers, and a sustained campaign of weeks or months to make, it is easy to not bother at all. For example, in the insurance business, rules for the manner in which rates are calculated change quite frequently. It might take several months before a new application could be deployed and released to agents in the field. In fact, by the time you released the application, new rates might be in effect. Maintenance costs escalate, while agents are faced with a situation where the system is never quite up-to-date.

ACTIVE OBJECT-MODELS are certainly harder to build than a conventional systems. They usually evolve out of frameworks. [Roberts & Johnson 1998] gives a simple overview of the process by which frameworks evolve.. If your system is only going to change a couple of times over its lifetime, then the effort entailed in constructing a framework may not be worth the cost and bother. However, when business rules change frequently, there is a decided advantage to be gained from letting changes to the system be released rapidly, and an Active Object-Model may right for you.

Power never comes without a price. When you confer the power to program on users, you give them the power to make mistakes. Just as certainly as ants follow picnics, where programs go, bugs shall surely follow. It is for this reason that the construction of ACTIVE-OBJECT MODELS should not be undertaken without a solid infrastructure of editing, programming, and support tools.

However, you don't want to simply expose a full-featured programming languages to your hapless users. Most users will rightfully consider programming as beyond their pay-grade. Instead, the key to design ACTIVE OBJECT-MODELS is to expose only those aspects of the problem domain that users need to change. The concepts these objects models expose should make sense in terms of business notions users will understand. The consequences of manipulating these objects should be in accord with the expectations that a user familiar with the business, but unfamiliar with programming, might have. This is one reason why such power should be exposed only as business requirements demand it.

How do you build ACTIVE OBJECT-MODELS ? Well, you use parameterization. Metadata is read from databases and objects are generated from schema descriptions at runtime. The ACTIVE OBJECT-MODEL pattern sits at the apex of a hierarchy of supporting patterns. Indeed, most of the patterns described in this paper support the emergence of ACTIVE OBJECT-MODELS.

Examples include the Objectiva Telephone Billing Framework, the Hartford UDP Framework, the Argo System, and the Caterpillar Financial Modeling Tool.

❖ ❖ ❖

An ACTIVE OBJECT-MODEL emerges as a bevy of lower-level patterns are applied in support of it.

Dynamic manipulations of the model's state vocabulary can be made using PROPERTIES. Behavior can be manipulated using the TYPE OBJECT, STRATEGY, STATE, and DECORATOR patterns. A range of creational patterns may come into play to assemble an ACTIVE OBJECT-MODEL. TEMPLATE METHODS, FACTORIES, BUILDERS, and PROTOTYPES may all be brought into play.

ACTIVE OBJECT-MODELS will often begin as COMPOSITES, and employ one or both of the INTERPRETER and VISITOR patterns.

Programmers may call upon a VISUAL BUILDER to construct their ACTIVE OBJECT-MODELS. Since the data that constitute an ACTIVE OBJECT-MODEL are, in effect, its program too, these must be saved and restored in an orderly fashion.

The SERIALIZER [Riehle et al. 1998] pattern, and in particular its variants that employ METADATA, can be of use here.

Conclusion

In biology, there used to be a now-discredited notion that *ontogeny* recapitulates *phylogeny*. The idea was that a developing embryo progresses through developmental stages that mirror evolutionary history. For instance, immature embryos develop, and then discard, gill-like features, and limbs emerge as flipper-like appendages that mature into arms and legs.

At times, it seems that the evolution of individual applications is driven by a microcosm of the same forces that have driven the evolution of programming languages. As systems mature, they demand more and more of the power of the tools that were used to build them. As a result, features seen in compilers and program development systems seem to migrate into applications themselves.

That this should be so is not surprising. Programmers will create worlds in the image of the ones in which they live. As they do so, they restore, one feature at a time, a wealth of metainformation that the programming environments used to build these systems once themselves gathered, and then discarded. For instance, serialization is easier to construct in a generic manner if you have maps of how your objects are laid out. Chances are that your programming system had just the information you needed, and threw it away as your program was compiled and linked. Hence, your debuggers and GUIs lose the benefit of this knowledge, and application programmers have to conspire to reconstitute it instead. Would it not be better to reuse these objects, and the tools that built them, at runtime, rather than reinventing them? Indeed, these hardworking, mature metadata objects are beginning to emerge from their anonymity and play these broader roles.

Ralph Johnson has observed that programmers today find themselves in a position reminiscent of the one that telephone operators were in the '30s. Then, it was calculated, that if telephones were to become universally available, the number of operators required would rapidly exceed the population of the United States. Everyone would have to become an operator. Of course, this is, in effect, exactly what happened. Telephone users find their own phone numbers, and complete the calls themselves. This, in turn is possible, because labor intensive plug-board operation and by-hand directory assistance has been replaced by direct dialing. The user interface has improved to the point where anyone is exposed only to those aspects of the task that are germane to their purpose. Hence, anyone can operate the telephone network, and most everybody does.

The same phenomenon can be seen today in the software world. Computer-phobes and Luddites who shunned technology during the '70s now lead hostile takeovers armed with easy-to-use spreadsheets. They call what they do "number crunching". Is it really programming? Does it matter anymore?

Acknowledgments

We are grateful to our PLoP '98 shepherd, Neil Harrison, as well as our PLoP '98 program committee overseer Jens Coldewey, and PLoP '98 program chair Steve Berczuk, for their faith, forbearance, and consul during this paper's somewhat protracted gestation. Its inadequacies, such as they are, are solely the responsibility of its authors.

We also owe a debt to the participants at the May Metadata Workshop at the University of Illinois, whose work, ideas, and insights have helped us to frame our own notions about these issues.

Discussions with Ralph Johnson, Dragos Manolescu, and Dirk Riehle helped shape our thinking about these issues, and their pattern-hood.

References

[Alexander 1979]

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

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

[Auer & Doble 1997]
Ken Auer & James Doble
Expedient Smalltalk Programming
Smalltalk Scaffolding Patterns
Proceedings of PLoP '97
Monticello, IL, October 1997

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

[Bobrow et. al. 1988]
D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel,
S. E. Keene, G. Kiczales, and D. A. Moon
Common Lisp Object System Specification
X3J13 Document 88-002R
SIGPLAN Notices, Volume 23,
Special Issue, September 1988

[Bobrow & Kiczales 1988]
Daniel G. Bobrow and Gregor Kiczales
The Common Lisp Object System
Metaobject Kernel -- A Status Report
Proceedings of the 1988 Conference on Lisp
and Functional Programming

[Borning 1986]
Alan Borning
Classes versus Prototypes in
Object-Oriented Languages
Proceedings of the ACM/IEEE
Fall Joint Computer Conference
Dallas, TX, November 1986, pages 36-40

[Brant et al. 1998]
John Brant, Brian Foote, Ralph E. Johnson,
and Donald Roberts
Wrappers to the Rescue (4/1/98 FINAL)
Proceedings of the 12th European Conferences on
Object-Oriented Programming (ECOOP '98)
Brussels, Belgium, 20-24 July 1998
To appear as part of the
Springer-Verlag *Lecture Notes in Computer Science*
series

[Chan et al. 1998]
Patrick Chan and Rosanna Lee
The Java Class Libraries
Second Edition, Volume 2
java.applet, java.awt, java.beans
The Java Series
Addision Wesley Longman, 1998
ISBN 0-201-31003-'

[Chan et al. 1998]
Patrick Chan, Rosanna Lee, and Douglas Kramer
The Java Class Libraries
Second Edition, Volume 1
java.io, java.lang, java.math, java.net,
java.text, java.util
The Java Series
Addision Wesley Longman, 1998
ISBN 0-201-31002-3

[Drescher 1985]
G. L. Dresher
The ObjectLISP USER Manual (preliminary)
Cambridge: LMI Corporation

[Ducassse et al. 1995]
S. Ducasse, M. Blay-Fornarino, A. M. Pinna-Dery
A Reflective Model for First Class Dependencies
OOPSLA '95, Austin Texas
SIGPLAN Notices, Volume 30, Number 10
pp. 265-
October, 1995

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

[Foote & Johnson 1989]
Brian Foote and Ralph E. Johnson
Reflective Facilities in Smalltalk-80
OOPSLA '89, New Orleans, LA
October 1-6 1989, pages 327-335

[Foote & Yoder 1995]
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

[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

[Gamma 1998]
Erich Gamma
Extension Object
Pattern Languages of Program Design 3
edited by Frank Buschmann,
Dirk Riehle, and Robert Martin
Addison Wesley Longman, 1998

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

[Gosling et. al. 1996]
James Gosling, Bill Joy, and Guy Steele
The Java™ Language Specification
Addison-Wesley, Reading, MA, 1996

[Johnson & Foote 1988]
Ralph E. Johnson and Brian Foote

Designing Reusable Classes
Journal of Object-Oriented Programming
Volume 1, Number 2, June/July 1988
pp. 22-35

[Kiczales, et al. 1991]
Gregor Kiczales, Jim des Rivieres,
and Daniel G. Bobrow
The Art of the Metaobject Protocol
MIT Press, 1991

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

[Maes 1987b]
Pattie Maes
Concepts and Experiments in
Computational Reflection
OOPSLA '87 Proceedings
Orlando, FL, October 4-8 1977 pages 147-155

[McCarthy et al. 1962]
John McCarthy, Paul W. Abrahams,
Daniel J. Edwards, Timothy P. Hart,
and Michael I. Levin
Lisp 1.5 Programmer's Manual, 2nd Edition
MIT Press, 1965, ISBN 0-262-12011-4

[McCarthy 1978]
John McCarthy
History of Lisp
ACM SIGPLAN History of Programming
Languages Conference
Los Angeles, CA June 1-3 1978
pages 217-223

[Messick & Beck 1985]
Steven L. Messick and Kent L. Beck
Active Variables in Smalltalk-80
Technical Report CR-85-09
Computer Research Lab, Tektronix, Inc., 1985

[Paepcke 1990]
Andreas Paepcke
PCLOS: Stress Testing CLOS
OOPSLA/ECOOP '90 Proceedings
Ottawa, Ontario, Canada

[Riele et al. 1998]
Dirk Riehle, Wolf Siberski,
Dirk Baeumer, Daniel Megert,
and Heinz Zuellighoven
Serializer
Pattern Languages of Program Design 3
edited by Frank Buschmann,
Dirk Riehle, and Robert Martin
Addison Wesley Longman, 1998

[Roberts & Johnson 1998]
Don Roberts and Ralph E. Johnson
Evolve Frameworks into Domain-Specific
Languages
Pattern Languages of Program Design 3
edited by Frank Buschmann,
Dirk Riehle, and Robert Martin
Addison Wesley Longman, 1998

[Smith 1982]
Brian Cantwell Smith
Reflection and Semantics in a
Procedural Programming Language
Ph. D. Thesis, MIT
MIT/LCS/TR-272

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

[Smith & des Rivieres 1984]
Brian Cantwell Smith and Jim des Rivieres
Interim 3-LISP Reference Manual
Xerox Intelligent Systems Laboratory ISL-1
Xerox Palo Alto Research Center
June 1984

[Steele 1984]
Guy L. Steele Jr.
Common Lisp: The Language
Digital Press, 1984

[Steele 1990]
Guy L. Steele Jr.
Common Lisp: The Language
Second Edition
Digital Press, 1990

[Stein et. al. 1988]
Lynn Andrea Stein, Henry Lieberman,
and David Ungar
A Shared View of Sharing: The Treaty of Orlando
Object-Oriented Concepts, Databases, and
Applications
edited by Won Kim and Frederick H. Lochovsky
ACM Press, New York, New York, 1989

[Stroustrup 1991]
Bjarne Stroustrup
The C++ Programming Language
Second Edition
Addison-Wesley, Reading, MA, 1991

[Ungar & Smith 1987]
David Ungar and Randall B. Smith
Self: The Power of Simplicity
OOPSLA '87 Proceedings
Orlando, FL, October 4-8 1977, pages 227-242

[Watanabe & Yonezawa 1988]
Takuo Watanabe and Akinori Yonezawa
Reflection in an Object-Oriented Concurrent
Language
OOPSLA '88 Proceedings
San Diego, CA, September 25-30, 1988
pages 306-315

[Yonezawa 1989]
Akinori Yonezawa, editor
ABCL: An Object-Oriented

Concurrent System
MIT Press, Cambridge, MA
1989