When Objects Collide: Experiences with Reusing Multiple Class Hierarchies

Lucy Berlin

Human-Computer Interaction Department
Hewlett-Packard Laboratories
1501 Page Mill Rd.
Palo Alto, CA 94304
berlin@hplabs.hp.com

Abstract

Well-designed reusable class libraries are often incompatible due to architectural mismatches such as error-handling and composition conventions. We identify five *pragmatic dimensions* along which combinations of subsystems must match, and present detailed examples of conflicts resulting from mismatches. Examples are drawn from our experiences of integrating five subsystem-level class hierarchies into an object-oriented hypertext platform. We submit that effective reuse will require that these pragmatic decisions be explicitly identified in descriptions of reusable software. Such descriptions will enable developers to identify and combine subsystems whose architectures are compatible.

1 Introduction

Object-oriented methodology promises greater productivity via reuse. The claim is that applications will be built by specializing and combining well-designed, reusable components, instead of repeatedly coding from scratch [9, 10]. Brad Cox, the developer of Objective-C, describes the promised benefits this way:

> The importance of object-oriented programming is comparable to that of Whitney's interchangeable part innovation. ... [Programmers will] produce reusable software components by assembling components of other programmers. These components are called Software-ICs to emphasize their similarity with the integrated silicon chip, a similar innovation that has revolutioned [sic] the computer hardware industry..."[3]

So far, object-oriented programming has had only limited success. Class libraries do provide a set of compatible components, plus the ability to specialize classes and protocols. There are testimonials to the productivity gained by using class libraries in integrated frameworks such as Smalltalk [4]. There also exist individual subsystems for window management, graphics, database management, etc. However, we still await general plug-compatibility – or the ability to grow reusable code libraries bottom-up. Incompatibility among well-designed but independently-developed components remains a major problem for reuse.

This article's contribution is to show that many compatibility problems in application integration lie not in the components, but in the current limited understanding of reuse. We identify five common architectural issues which frequently cause incompatibilities among components and expand the view of reusable components to include these *pragmatic* architectural decisions. These

pragmatic decisions need to be explicitly identified in descriptions of reusable software. Such descriptions will prevent unexpected architectual conflicts by enabling developers to identify and combine classes from compatible families of subsystems.

We support our conclusions via examples from our project's development of a multi-layer hypertext platform. In each architectural dimension we show that pairs of *independently sensible* pragmatic decisions can cause fundamental incompatiblities among components. Our hypertext platform was composed of well-designed subsystems which comprised over 200 CLOS classes. They included a user-interface management system (UIMS), a window management toolkit, an environment kernel, and a CLOS extension that provided object persistence and database access.

Section 2 of this article introduces the conflicting-code problem of application integration, defines design *pragmatics*, and shows how class decomposition distributes the pragmatic code and disperses its documentation. Section 3 introduces our hypertext system.

Section 4 describes each of the five pragmatic dimensions we identified, presents alternate design choices, and describes detailed examples of problems caused by sensible but incompatible decisions. We advocate that reusable components should explicitly describe their architectural model, so that future developers can avoid unexpected problems and intricate surgery.

2 The Problem of Conflicting Code in Reuse

A large software system is composed of generic components plus some application-specific code. Subsystems are components such as a window system, UIMS, database, or a graphics package. In an integrated system framework such as Smalltalk, these will fit together like matching puzzle pieces, or like nuts and bolts.

However, if each component is developed separately, they may conflict – rather like a metric nut and a British bolt. Here are two examples of software conflicts (which will be analyzed later):

1. One subsystem expects the functions it calls to return error values, while another propagates exceptional conditions via a non-local jump (e.g. longjmp in C, or error throw in Lisp). Exceptions thrown by the second subsystem cause an error, since there is no error-handling environment (setjmp environment or catch frame) on the procedure call stack.

2. UIMS and inter-process-communication (IPC) subsystems each have separate event-handling loops, so that when the UIMS's handler is blocked waiting for user events, IPC messages are not heard by the IPC's handler.

Note that each subsystem's architecture is valid individually, but glue code cannot make them compatible. The problems are not in the mainline functionality, but in the assumed flow of control. Therefore, a fix requires breaking encapsulation and redesigning the internals of one or both modules. So, even if each component is well-designed, robust, and has the specified functionality, it may fundamentally conflict with other code.

In general, there are three causes of conflict between software components: missing code, redundant code, and conflicting code. The first two are relatively minor. Glue code can fill in missing code via protocol translation – coercing argument types, combining arguments, even translating method names. Redundant code results from more generality than required by the composite system. Examples are multiple validations

of arguments, and multiple representations for data in different software layers. Such redundancy may reduce performance, but it is an expected price of using general, context-independent pieces.

Conflicting code, which is the third type of mismatch, can be fatal. This is the focus of our article. It occurs when multiple components each make sensible choices about *how* to implement functionality, but the choices conflict in fundamental ways.

2.1 Pragmatics – A New Issue in Reuse

The standard view of reuse has a developer select a reusable component based on its mainline semantics and count on glue code to fix minor differences. To address problems with conflicting code, we expand the view of reusable component interfaces to include choices about information flow that are generally viewed as syntactic or encapsulated.

We use the term *pragmatics* to describe the global architecture of a component that affects *how* it provides its functionality. Examples include the difference between (a) a constraint-based and procedural architecture, and (b) two databases that provide the same semantics (locks, caches...) but handle exceptions differently: (throws and catches vs. error values). Our categorization fits Hartson's [6] model of *dialogue - semantics* separation of software. Hartson focuses on the semantics required in a user interface for task-oriented behavior and analyzes the resulting couplings between a UIMS and an application. Pragmatics describe what dialogue can be supported between application layers, and thus between the user and the application.

Global information flow alternatives are often discussed in workshops for cutting-edge practitioners in a technology (e.g. window management [7], remote-procedure calls, hypertext). However, the tradeoffs and domain needs discused are often never written down – the insights are shared just among the cutting-edge developers of that technology. We need to recognize that these tradeoffs cut across technologies, low-level choices impact user-level capabilities, and choices must be compatible between components in a system. Systematic reuse of multiple components requires a systemic analysis of the pragmatic issues. As we show below, the decomposition of object-oriented systems that enables reuse makes principled choices in these dimensions even more crucial.

2.2 Pragmatics and Object-Oriented Programming

The pragmatic dimensions all describe information flow decisions. The main concepts in object-oriented programming [9, 15, 16, 18] are inheritance, polymorphism, and abstractions with encapsulation. These allow reusability and customization of behavior. Inheritance and polymorphism provide compatibility *within each* class hierarchy, with customizability via subclassing and common protocols. However, we show that this decomposition of problems into abstract classes disperses and de-emphasizes control flow, and obscures the pragmatic decisions of a class hierarchy.

First, class hierarchies disperse code that embodies pragmatic decisions such as error-handling style. Imagine a window module with four classes, each of which responds to six mouse events. Testing for errors and errorhandling may occur in each of the 24 method definitions, since each subclass may have to test a slightly different set of arguments, or may have different constraints on argument values. Changing errorhandling from throws to error-returns would require more than simply breaking the module's encapsulation and a local fix: it may require understanding and re-coding *each* event-handler method.

Second, global flow of control and behavior is much less visible in object-oriented code than in procedural code. Procedural programs are process-oriented, so the pragmatic decisions are made explicit. As Rosson and Gold's studies show, object-oriented developers tend to decompose problems by proposing relevant classes, and identifying the behaviors each individual class should have [13]. As Hartson also points out, a major shortcoming of object-oriented programming is that "it lacks a procedural representation of control flow from which a behavioral perspective can be obtained at the global control level" [6].

Third, not only is global behavior difficult to identify by reading the code, it is also de-emphasized in module documentation. Class documentation is semantics- and syntax-oriented. Modules are documented by specifying class dependencies, plus exported methods and their arguments and return values [3, 10]. Thus, there is no standard place or terminology to describe the pragmatic decisions that underlie global behavior. These are only implied by code syntax or exposed piece-meal in component-level documentation.

In conclusion, understanding and exposing the pragmatic decisions becomes especially important in object-oriented code. Object-centered decomposition inherently replicates and disperses pragmatic code, and inheritance complicates and obscures global control flow. Moreover, the methodology currently disregards the global pragmatics in its method-centered interface descriptions.

3 Hypertext Platform

In this section we introduce the hypertext features used in later examples and the platform layers we used to construct the system. Our project's goal was to provide groups with effective access to large amounts of complex, dynamic information. Our focus was on information organization and user interface strategies, with hypertext as an enabling technology.

Hypertext helps users manage complex information by providing flexible connections (links) between objects (nodes). Conklin [2] is a good introductory survey. Users browse follow relevant references in text or data by selecting link icons in the user interface. Users add their own information by adding links and nodes. To help manage information, we provided aggregate nodes, browsers to display the hypertext network, and database queries. Links could be global (referring to the entire node) or local (pointing directly at a word or phrase). We provided multi-user functionality using a shared database with lock-based access control. For efficiency, each user's process also cached a portion of the database. Each process therefore also managed contention and updates from other users' activities.

Our project seemed to be an excellent candidate for component-based integration. We had available multiple subsystems that would be useful in a hypertext platform, each developed in a highly object-oriented style by experienced programmers. All were developed in CLOS [1, 10], a flexible object-oriented language based on Common Lisp [17]. CLOS provides multiple inheritance, dynamic dispatch via generic functions, and a powerful meta-object protocol. Most subsystems were research prototypes already used in other projects. The subsystems were functionally complementary, as they were intended to be eventually used together. Any mismatches were intended to be fixable via glue code.

Our subsystems included (a) "policy-free" window system interface [5] to the X Window system [14]; (b) a UIMS, which provided customizable event-handlers, emacs-like text presenters, etc; (c) an object-oriented environment kernel; (d) the PCLOS [11] persistence extension to CLOS; and (e) an inter-process-

communication (IPC) module. Together they comprised over 200 classes, each with some 20–200 methods. We combined these into a multi-user platform for building hypertext-based applications.

We used these subsystems to build our platform. The platform included both a hypertext object model and example user interface classes for manipulating system. The platform itself was validated by a computer conferencing application [8]. We found the available subsystems very useful in developing our hypertext prototype. We instantiated, subclassed, and combined platform classes. A hypertext data model, test user interface, and conferencing application prototype required only about 40 new classes, and most classes made use of dozens of inherited methods.

As developers, we cared about providing a useful system with a full range of appropriate presenters and functions, not just about mainline "proof-of-concept" functionality. This goal made us more representative of reuse needs in product development than in proof-of-concept prototyping. Our goal of broad and task-driven functionality made us emphasize appropriate user interface style, group operations, error-handling, and performance, and thus led us to analyze the conflicts we found between subsystems.

We identified dozens of conflicts in combining the components. Problems included insufficient decomposition, missing extensibility hooks, bugs as well as pragmatic architectural conflicts. This article's focus is on problems where the mainline functionality was appropriate, yet the classes turned out to be incompatible. These conflicts added over a third to our development time, and were among the hardest to fix retroactively. The pragmatic problems under discussion could not be fixed by glue code – they required either major degradation of the user-visible functionality, or breaking encapsulation and substantial surgery of a platform layer.

4 Five Pragmatic Issues in Component Reuse

The major conflicts between component architectures occurred along the following five dimensions:

1. argument validation
2. error handling
3. composite object handing
4. control and communication
5. group and compound operations

In this section we discuss each pragmatic issue, present common alternative choices, and describe examples of the conflicts we encountered. These issues and examples of typical conflicts are tabulated in Figure 1.

4.1 Argument Validation

Argument validation in the user interface requires cooperation among many layers of an application. For example, the command to move a link requires the user to indicate valid source and destination locations. At one extreme, if a direct-manipulation user interface visually indicates potentially valid link locations, then locations must be validated real-time, before the command is completed. We will examine the coupling required for different types of user interfaces and the effect of the choices on software complexity.

There are three ways for a user to specify arguments: declaratively (all arguments validated together), interactively (the user gets feedback after each), or implicitly (based on the user's preferences, the operation, or the context). The appropriate interface depends on the application, but what can be done cleanly depends on the underlying layer's creation style.

Pragmatic Issue	*Example Conflicts*	
	Caller Expects:	**Callee Expects:**
Argument Validation	Interactive validation.	Declarative validation.
Error Handling	Error value; Responsibility for errors.	Non-local jump (throw); Responsibility for errors.
Composite Objects	Top-down creation; Outside-in event handling.	Bottom-up creation; Inside-out event handling.
Control and Communication	Responsibility for event loop; Responsibility for cleanup.	Responsibility for event loop; To do cleanup / notification.
Group and Compound Operations	Single setup / cleanup for a group operation.	Each element encapsulates own setup / cleanup.

Figure 1: Five Pragmatic Issues and Example Conflicts

The declarative style is common in command interfaces. It greatly simplifies the module's interface and can simplify validation if the arguments interact. The problem of the declarative style of argument validation is that even if only the first argument of several is invalid, the user only finds out after specifying all arguments. In a `link-nodes` operation this would mean that if the first node is locked, or the user does not have permission to modify, the user still has to specify both endpoints before any feedback is provided. To avoid this frustration, modern interfaces tend to validate arguments interactively.

Interactive validation requires the interface to know more about the semantics of the underlying application in order to provide feedback and validate input real-time. This requires quite close coupling and a much larger surface area of each of the two subsystems. For example, to allow independent argument selection and validation, the semantics layer must export separate argument validation functions. Form-based interfaces which present defaults and do immediate validation require the above separate validation functions, plus separate functions that access the default value of each argument.

We tried to combine an interactive user interface (that expected argument-validation functions and default-value accessors) with a declarative subsystem (which provided a single function for each command, and encapsulated the validation and creation). To provide validation functions and default-value accessors we repeatedly copied and adapted internal *very* low level functionality. Not only did this require us to open up reusable modules and trace internal functions a number of levels deep to find the validation functions, but the functions usually required complete recoding to be useful.

Why the recoding? The internal validation functions were generally optimized for a single sequence of validation and creation steps. They expected arguments which had already been parsed, massaged, or combined so they could be tested together. And, because they assumed an information flow sequence, internal functions sometimes had side effects that were unacceptable in our context (e.g. starting a database transaction, setting a default value, or creating an object).

So, the mismatch between the argument validation styles of the interface and the underlying system forced us to repeatedly break encapsulation, painstakingly an-

alyze the assumed context of low-level functionality, divide up the validation steps into separate functions, and then generalize the validation functions so they do not presume a fixed context.

The argument validation issue identifies tradeoffs in the amount of coupling between layers. Narrow coupling via declarative interfaces is easier to provide and understand, but is too constraining in many domains. Direct-manipulation-style interfaces provide early semantic feedback, but multiply many times the number of lower layer functions which have to be exported and connected to a matching upper layer. Direct-manipulation interfaces are powerful in part because of the amount of semantic information they have available. However, especially in domains that require tight coupling, it is unlikely that two independently developed layers will match.

The goals of plug-compatibility and domain-appropriateness conflict. If we can categorize different application domains' needs, we may be able to resolve this tension by classifying needs and specifying domain-specific frameworks for components which agree not just in the basic semantics, but in the amount of coupling between layers.

4.2 Error Handling

Error handling is an inherently multi-layer activity. It involves detecting resource failures and incorrect values, identifying the type of problem, and either passing the problem up (with some cleanup if appropriate), or handling the problem. Error handling is not a rare event if one includes recovering from multi-user resource-contention problems.

Each subsystem made independent assumptions about the validity of its arguments, the errors it expected, error actions, and the error values it would pass back up. We ran into the following problems.

1. *Passing the buck.* For example, a window manager could not find a shared resource (e.g. a font), and returned NIL, expecting that the enclosing creator would test for NIL, and recover. The UIMS expected the window manager to "do something reasonable". The result was a crash with the uninformative message of "read error in ..".

2. *Not passing the buck.* Sometimes the higher layer knew how to handle an error, but wasn't given the chance. For example, a lower layer decided to allocate a default color whenever a color wasn't specified. Our application did not want that default color, but had no control. The error-handling occurred within the body of an internal function in the platform. In order for the application to have control, the error would have had to be passed up, or an error-handler method exported. If the method were available, the offending class could be subclassed to redefine the error-handling method.

3. *Incompatible conventions.* Cross-layer communication requires a common protocol. Each layer may handle or propagate errors, but if two layers do not share the same conventions then errors may fall between the cracks. If a layer expects error values but calls methods that do error throws, then there's a conflict. To force a fit, all higher-layer methods would have to be modified to provide appropriate catches, and do whatever is the expected cleanup on error throws.

Error-Handling requires recovering from problems at any level in the system. Recovery may include clean-up of inconsistent data and releasing resources, undoing partial operations, as well as communicating the error value up to the user or to a higher error handler. Thus,

it requires shared understanding of the side effects of actions, division of responsibilities, and of course conventions such as whether to look for error values or error throws. The developer expects each layer to "do the right thing". If the layers' expectations do not match, the result is missed signals, and misguided operations that compound rather than fix or communicate the errors.

4.3 Composite Object Handling

Composite objects are structured objects whose elements are other objects. Examples are browsers, menus, multi-window presenters, and hypertext nodes and their links. The choice in designing composites is direction of control – should the component be ruled bottom-up or top-down? A conflict occurs if an element tries to exercise bottom-up control in a top-down composite, or conversely, if an element expects but fails to get instructions. This conflict commonly manifests itself in two issues: (a) creation style and (b) event-handling style.

Creation style

There are two separate philosophies on creating composites based on how much control the caller has of component types and component attributes.

The declarative (outside-in) style lets the developer specify a set of abstract choices, (e.g. `(create-node :type reply :reply-to message-75)`). The composite is created in an encapsulated form, with component values based on the abstract choices. Attributes not in the composite's interface cannot be specified without breaking encapsulation and redefining the entire creator function. This outside-in style is generally associated with outside-in control of component attributes such as size, colors, behaviors. Top-down control makes possible uniformity by decree, rather than by each component separately specifying the same choices.

The top-down style makes inside-out control difficult. Common requests such as these become cumbersome: "make this button's size be based on its label", or "make this browser large enough to handle the number of elements". This is solved by an interactive (inside-out) style of component-creation, which exports creators for each of the elements, and builds the composite once the elements have been constructed. So, a browser can be easily laid out based on the widths of its elements, and its length can be based on the number of elements. Since the elements are created first, it makes it more difficult to specify common attributes among components (such as colors, fonts).

A hybrid control style (of bottom-up hints and top-down control) is possible, and we wound up using that for creation. However, such bi-directional information flow requires even tighter coupling between subsystems; it required redesign of both layers.

Event-handling style

Event handling refers to the system's actions in response to mouse, keyboard, or inter-process-communication events. Incompatibilities occur if events are incorrectly intercepted or dropped, if the set of events are not the appropriate ones, or if the application cannot set aside a software interrupt.

Order of handling of events is a classic difference between window systems. It occurred between our window library and our UIMS, and it occurs between X11 and Andrew [12]. We describe the Andrew/X11 conflict, since it is more broadly known. In X11, the innermost presenter which knows how to handle the event gets it, and the outermost only sees the event if it is passed

through. In Andrew, the outermost presenter (view) has the responsibility of intercepting events, giving the child the input focus, and controlling the key bindings and menus in effect at each point.

Thus, in Andrew, it is easy to make a window insensitive to mouse events or to provide a common set of commands. However, an Andrew subwindow cannot redefine its parent's event actions. In X11, it is simple to provide specialized functionality as a replacement of the default.

In general, a philosophy that the enclosing object is in charge of its components it leads to a style of object creation and event handling which makes it simple to specify common behaviors and attributes. However, it is more difficult to describe composite objects whose attributes (shape, behavior) are dynamically calculated from their contents. Each style is appropriate in different contexts, but they do not combine easily.

4.4 Control and Communication

As Hartson points out in his article *User-Interface Management Control and Communication* [6], an ideal separation between components at runtime is hard to define and even harder to achieve. The problems of "Who's in charge?", "Am I just a messenger or does the buck stop with me?" and "How do I coordinate?" are general ones between layers, although they're much more visible in the user interface layer.

Let us look at control. User interface functions can be called by the application (internal control) or the interface can call the application when the user gives a command. The first is used by interface toolkits and the second by the more powerful UIMSes, since it lets the interface handle scheduling and sequencing. The application is structured as a set of objects and methods (callbacks) that are called by the interface.

Besides agreeing on who is in control, layers must agree on the granularity of communication – when constructing objects, validating input, and reporting interesting internals changes to higher layers. Again, let us look at the UIMS interface to the application. Semantic feedback is important to users, but it requires tight semantic coupling. For example, direct manipulation interfaces may require application feedback at each refresh, in order to indicate whether the object under the mouse is a valid argument. As shown by Szekely [19], this test may require semantic computation, and the computation may be dynamic – choosing one argument may constrain subsequent choices.

A common problem of platform layers was not knowing when to pass the buck, and when to do setup and cleanup actions. For example, link deletion must notify all affected objects – the collections and each endpoint node. A link may be deleted by *link-delete* messages to three types of objects: a collection, either node, or the link itself. Since messages are associated with objects, it is difficult to implement an encapsulated *delete* operation associated with some object, yet have the notification take into account the context in which the command originated (e.g. don't notify the node whose death triggered the *link-delete* in the first place.) The simple solution of a *link-delete* message in each class duplicates the core *delete* operations, but can provide context-sensitive locking, error-handling and notification.

Another example involves two roles played by a class. The class *color* is useful by itself, but it may also be an ancestor of class *approximate-color*. CLOS does not provide clean mechanisms to specify that some of *color's* initialization actions (e.g. calculating default values) are not to be done when the method is called via the subclass's *call-next-method*. One strategy would have been to redesign the UIMS to separate *XX-abstract* and

XX-concrete classes. We did not want to do this — especially not in the window system and UIMS layers, where many classes were useful in two roles. With over 100 classes in the two layers, we did not want the extra conceptual complexity of duplicate classes and methods.

Composite operation problems were due to (a) multiple method combinations among inter-related objects and (b) multiple roles played by methods and classes. Without knowing how a class will be used, or how methods will be combined, it is difficult to separate out the code to be done only if one is the concrete class, or the primary method. As shown repeatedly, the more shared understanding there is between provider and client, the simpler the communication, but the less generality in directions other than the ones for which flexibility was explicitly provided. Very flexible interfaces would require more decomposition, and as a result more cognitive complexity – more terminology; more classes, arguments, and defaults; and more complex control flow.

4.5 Group and Compound Operations

Group operations such as "delete these five items", "move text and links within this region" posed problems. The problems were those of inefficient or redundant actions, extra error possibilities, and of complex undo actions in case of error. These problems were caused by encapsulation at the wrong level, but the "right" level was not possible to foresee.

As an example, consider a simple operation to delete nodes N1 and N2 from a collection. Our platform provided a "delete-node" operation. Thus, it should have been trivial to implement "delete-nodes" as a loop calling the exported operation "delete-node". From a hypertext object model perspective, that is correct. However, this ignores side effects. Let us examine a simple encapsulated implementation of "delete-node". The steps are to:

1. open a transaction to the database,
2. re-cache and lock a group of objects: node, node's links, the neighboring nodes (other end-points of the links), and the enclosing collection.
3. delete the node, links and link-anchor information from the adjacent nodes,
4. save the changes to the database,
5. commit the transaction, unlocking the objects,
6. notify the presenters of the nodes, and notify the collection.

There are multiple problems with repeating this for each node to be deleted.

1. The side effects of locking the collection, locking the neighboring nodes (N3 and N4), and notifying the presenters of the nodes and collection are done after each node.
2. The collection and neighbor nodes are unlocked after each element is deleted. This increases the likelihood that at some point one of the objects won't be available.
3. If the user's intent is to do all or none of the actions, then this approach fails, since some of the side effects (to the user interface and to the other users' processes) occur immediately after each element's operation.

We ran into this problem in many different guises – implementing a *move-link-endpoint* as a combination of *delete-link-endpoint* and *insert-link-endpoint*; creating a composite presenter (via an *add-element* operation that re-checked sizes and colors of siblings and parent); retrieving links within a range of text (given a primitive that only fetches one link at a character position).

One could say that it was poor design of the platform to not provide these group operations. We do not believe this to be true. Composites may be defined at many different application levels, each of which requires combining or deferring some shared operations. A platform should not be expected to provide hooks for clean implementations of all possible specialized composite operations – its goal must be a good set of fundamental operations. There must be some composites which fall outside of its realm, else the platform would have to be infinitely complex.

The general problem is that group operations miss opportunities for "parallelism" in validation, modifying shared data structures (such as collections, link anchor tables) and postponing and combining clean-up actions (such as notifying interested parties). By not doing validation and locking up front, clean-up is complicated, since operations (including side effects) on some elements are completed by the time an error is found.

5 Conclusions and Implications

This paper shows that a broader view of reuse is required to successfully integrate multiple class hierarchies. Using examples from our experiences integrating subsystems comprising 200+ classes and 40,000 source lines into a hypertext platform, we showed that:

- Independently sensible architectural decisions can lead to fundamental conflicts among complementary reusable components.
- These *pragmatic* decisions are pervasive yet neglected by object-oriented methodology.
- Software descriptions should include not only the *semantics* but also the global *pragmatic* decisions of the subsystem's architecture, since these decisions must be compatible across subsystems.
- Five of these decisions include: argument validation; error handling; composite object handling; control and communication; and group and compound operations.

An important point is that the differences among modules were not flaws. They represented choices of what would be the simplest, most useful, and most efficient architecture. In each subsystem the decisions were based on the target application domain and user interface, the designer's design methodology and personal style, low-level issues such as language capabilities, and the features and speeds of even lower layers (database, operating system).

Different semantic choices on these issues (e.g. direct manipulation interfaces versus form-based interfaces, event-handling and dialogue styles) each fit a need in different application domains. Thus, these choices will not go away, and diverse solutions will continue to exist. We need to learn to manage them by (a) classifying them into families, (b) understanding the interactions among the dimensions, and (c) matching application needs to appropriate pragmatic dimensions. This article provides a beginning terminology and classification, but we need more analysis of other systems and application domains.

A general issue which permeates these conflicts is "Who's in charge?" and the related question of "How do components coordinate?". In information-flow areas such as event-handling, error-handling, or cleanup, developers wishfully expect a reusable component or class to be able to act in two roles – as a decision-maker who provides reasonable behavior, and as a sub-part that propagates values and errors, but does not take charge. As we have seen, this does not work. We should acknowledge that components do serve in these two roles, and provide mechanisms to express the differences: First, programmers must be able to declare which code serves which role. Second, at integration time, they must be

able to indicate the role they need a component to fit. Only this way will we get components that can be successfully used as either messenger or decision-maker layers.

We believe that plug-compatibility of multiple independent components is possible, but requires support for the global issues in reuse. It will require that developers of reusable components consider and explicitly document components' pragmatics as well as main-line semantics. This will allow families of architecturally compatible subsystems to develop, each appropriate for some set of problem domains. Eventually application developers will be able to describe their domain's requirements along pragmatic decisions, and select an appropriate family of plug-compatible subsystems.

Acknowledgements

Valuable discussions with Robin Jeffries, Nancy Kendzierski, Jarrett Rosenberg and Glenn Trewitt have encouraged this work and helped clarify my understanding of the issues. I also thank my fellow Hoopertext and object-oriented platform developers in HP Labs' Software and Systems Laboratory, without whom there would not be software to analyze. The developers included Mike Creech, Cathy Fletcher, Dennis Freeze, Warren Harris, Shari Jackson, Bob Leichner, Andreas Paepcke and Jarrett Rosenberg.

References

[1] D. G. Bobrow, L. DeMichiel, R. P. Gabriel, G. Kiczales, D. Moon, and S. Keene. The Common Lisp Object System specification: Chapters 1 and 2. Technical Report 88-002R, X3J13 standards committee document, 1988.

[2] J. Conklin. Hypertext: An Introduction and Survey. *IEEE Computer*, September 1987.

[3] B. J. Cox. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Reading, Mass., 1986.

[4] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison Wesley, 1984.

[5] W. Harris. Xenon user's guide. Internal STL-TM-88-02, Hewlett-Packard, May 1988.

[6] R. Hartson. User-interface management control and communication. *IEEE Software*, 6(1):62–70, January 1989.

[7] F. Hopgood, D. Duce, E. Fielding, K. Robinson, and A. Wiliams, editors. *Methodology of Window Managment.* Eurographic Seminars. Springer-Verlag, April 1986. Proceedings of an Alvey Workshop on Window Management, Abingdon, UK, April 1985.

[8] S. L. Jackson. Hypertext for computer conferencing. Master's thesis, Massachusetts Institute of Technology, June 1989.

[9] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June 1988.

[10] S. E. Keene. *Object-Oriented Programming in Common Lisp.* Addison-Wesley Publishing Company, 1989.

[11] A. Paepcke. PCLOS: A Flexible Implementation of CLOS Persistence. In S. Gjessing and K. Nygaard, editors, *Proceedings of the European Conference on Object-Oriented Programming*, pages 374–389. Lecture Notes in Computer Science, Springer Verlag, 1988.

[12] A. Palay. The Andrew Toolkit – an overview. In *Proceedings of the Winter 1988 USENIX*, 1988.

[13] M. B. Rosson and E. Gold. Problem-solution mapping in object-oriented design. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 7–10. ACM, 1989.

[14] R. Scheifler and J. Gettys. The X Window system. Technical Report MIT LCS Memo LCS-TM-368, MIT, 1986. .

[15] A. Snyder. The essence of objects. Technical Report STL-89-25, Hewlett-Packard, September 1989.

[16] A. Snyder. Inheritance in object-oriented programming languages. Technical Report STL-89-34, Hewlett-Packard, November 1989.

[17] G. L. Steele Jr. *Common Lisp: The Language.* Digital Press, 1984.

[18] B. Stroustrup. An overview of C++. *SIGPLAN Notices*, 21(10):7–18, October 1986.

[19] P. Szekely. Modular implementation of presenters. In *Proceedings of SIGCHI+GI'87*, pages 235–240, March 1987.