Reuse of Off-the-Shelf Components in C2-Style Architectures

Nenad Medvidovic, Peyman Oreizy, and Richard N. Taylor

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92697-3425
{neno,peymano,taylor}@ics.uci.edu

Abstract -- Reuse of large-grain software components offers the potential for significant savings in application development cost and time. Successful component reuse and substitutability depends both on qualities of the components reused as well as the software context in which the reuse is attempted. Disciplined approaches to the structure and design of software applications offers the potential of providing a hospitable setting for such reuse. We present the results of a series of exercises designed to determine how well "off-the-shelf" components could be reused in applications designed in accordance with the C2 software architectural style. The exercises involved the reuse of two user-interface constraint solvers, two graphics toolkits, a World Wide Web browser, and a persistent object manager. A subset of these components was used to construct numerous variations of a single application (thus an application family). The exercises also included construction of a simple development environment for locating and downloading a component off the Web and incorporating it into an application. The paper summarizes the style rules that facilitate reuse and presents the results from the exercises. The exercises were successful in a variety of dimensions; one conclusion is that the C2 style offers significant reuse potential to application developers. At the same time, wider trials and additional tool support are needed.[1]

Index Terms **-- software reuse, architectural styles, message-based architectures, component-based development, graphical user interfaces (GUI).**

I. Introduction

There are numerous difficulties inherent in reusing off-the-shelf (OTS) software components [BP89, Big94, Kru92, GAO95, Sha95]. Some common problems developers face when attempting OTS reuse are:

- OTS systems may not contain clearly identifiable components;
- component granularity is too coarse or too fine;
- components do not provide the exact set of functions required;
- specialization and integration of OTS components is unpredictably complex; and
- the costs associated with locating, understanding, and evaluating a component for use may be higher than writing the component from scratch.

1. This material is based upon work sponsored by the Air Force Materiel Command, Rome Laboratory, and the Advanced Research Projects Agency under contract number F30602-94-C-0218. The content of the information does not necessarily reflect the position or policy of the Government and no official endorsement should be inferred.

Software architecture research is directed at reducing the costs of developing applications and increasing the potential for commonality between different members of a closely related product family. One aspect of this research is development of software *architectural styles*, canonical ways of organizing the components in a product family [GS93, PW92]. Typically, styles reflect and leverage key properties of an application domain and recurring patterns of application design within the domain. As such, they have the potential for circumscribing the scope of OTS reuse, providing structure for it, and thus alleviating many of its difficulties [MG96].

However, all styles are not equally well equipped to support reuse. If a style is too restrictive, it will exclude the world of legacy components. On the other hand, if the set of style rules is too permissive, developers may be faced with all of the above-mentioned problems of reuse in general. Therefore, achieving a balance, where the rules are strong enough to make reuse tractable but broad enough to enable integration of OTS components, is a key issue in formulating and adopting architectural styles.

Our experience with the C2 style [TMA+95, TMA+96] indicates that it provides such a balance. In a series of exercises, we were able to integrate several OTS components of various granularities into architectures that adhere to the rules of C2. A subset of these components was used to create a large number of variations of a single application, i.e., an application family. We were also able to build a simple development environment according to the style. Using this environment, we enacted a scenario where a component is located on the World Wide Web (WWW), downloaded, and integrated into an already executing C2 application.

In these initial exercises, we focused on the following issues:

- requirements for incorporating an external component into a C2 architecture;
- support provided by C2 and its accompanying tools for overcoming the common problems of reuse listed above;
- issues in substituting one C2 component for another, providing the same or similar functionality;
- partial utilization of the services a component provides, as a byproduct of using legacy components in new, unforeseen contexts; and
- using C2 as a basis for development tool integration, as well as application component integration.

The remainder of the paper is organized as follows: Section II discusses C2 concepts with an emphasis on those that promote reuse. The material in this section is condensed from a more detailed exposition on the style, given in [TMA+95, TMA+96]. Section III describes the support tools used in our reuse exercises, while Section IV provides a detailed overview of the exercises and emphasizes those C2 concepts that contributed to their success. In Section V we discuss lessons learned and related work. A discussion of future work rounds out the paper.

II. Support for Reuse in C2

C2 is a component- and message-based architectural style for constructing flexible and extensible software systems. A C2 architecture is a hierarchical network of concurrent components linked together by connectors (message routing devices) in accordance with a set of style rules. The top of a component may be connected to the bottom of a single connector and the bottom of a component may be connected to the top of a single connector. There is no bound on the number of components or connectors that may be attached to a connector (see Fig. 1).

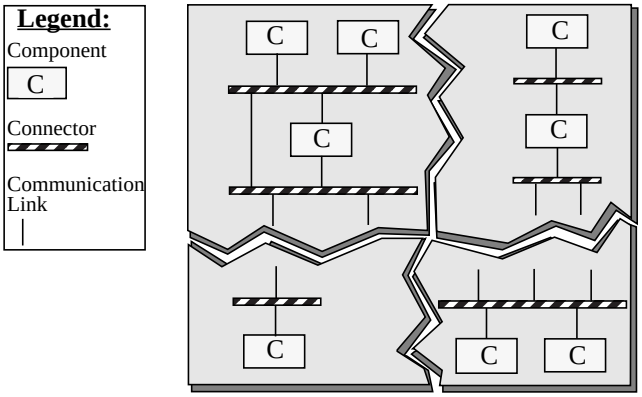


Fig. 1. A sample C2 architecture. Jagged lines represent the parts of the architecture not shown.

Several characteristics of the C2 style enable it to better support reuse. Although most of these characteristics are not unique to C2, our approach of combining them is. We believe the style rules are restrictive enough to make reuse easier while flexible enough to integrate components built outside the style:

- *component heterogeneity* - the style does not place restrictions on the implementation language or granularity of the components.
- *substrate independence* - a component is not aware of components below it, and therefore does not depend on their existence.
- *internal component architecture* - the internal architecture of a C2 component, shown in Fig. 2, separates communication from processing. The internal *dialog* receives all incoming notifications and requests and, in turn, maps them to *internal object* operations. By localizing this mapping, the dialog isolates the internal object from changes in the rest of the architecture. When the state of the internal object changes, a notification is broadcast down the architecture. Dialogs of components below will interpret this notification and invoke their internal objects as appropriate. This implicit invocation reduces dependencies between communicating components. The *domain translator* is used to resolve incompatibilities between communicating components, such as mismatches between message names, parameter types, and ordering of parameters [YS94, TMA+96]. Domain translation reduces the dependence of a component on components above it; the component can use different domain translators in different architectures.
- *asynchronous message passing via connectors* - all communication between components is achieved by exchanging asynchronous messages through connectors. Since all message passing is done asynchronously, control integration issues are greatly simplified. This remedies some of the problems associated with integrating components which assume that they are the application's main thread of control [GAO95].[2]

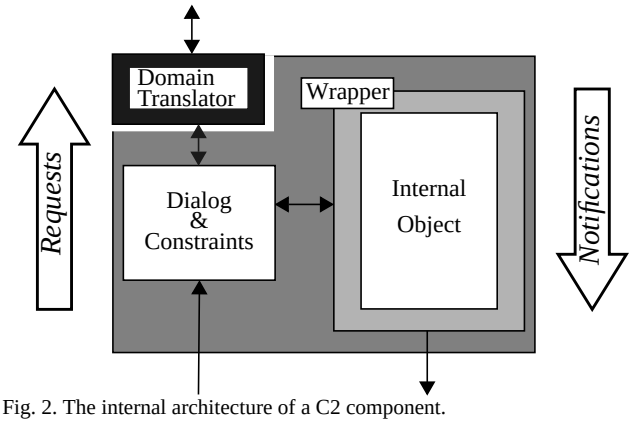


Fig. 2. The internal architecture of a C2 component.

- *no assumption of shared address space* - components cannot assume that they will execute in the same address space as other components. This eliminates complex dependencies such as components sharing global variables.
- *no assumption of single thread of control* - conceptually, components run in their own thread(s) of control. This allows multi-threaded OTS components, with potentially different threading models, to be integrated into a single application.
- *separation of architecture from implementation* - many potential performance issues can be remedied by separating the conceptual architecture from actual implementation techniques. For example, the C2 style disallows direct procedure calls and any assumptions of shared threads of control or address spaces in a conceptual architecture. However, substantial performance gains may be made in a particular implementation of that architecture by placing multiple components in a single process and address space where appropriate. We have found that we can isolate such implementation decisions in the C2 framework, discussed in Section III.A. For example, if two components are placed in the same address space, a connector between them can use direct procedure calls to implement message passing.

III. Role of C2 Development Tools in Enabling Reuse

We have constructed several development tools to aid in the design and implementation of C2 architectures and in reuse of OTS components. This section describes a class framework and a runtime manipulation tool for C2 architectures.

III.A. C2 Class Framework

To support implementation of C2 architectures, we developed an extensible framework of abstract classes for C2 concepts such as components, connectors, and messages, shown in Fig. 3. This framework is the basis of development and OTS component reuse in C2. It implements component interconnection and message passing protocols. Components and connectors used in C2 applications are subclassed from the appropriate abstract classes in the framework. This guarantees their interoperability, eliminates many repetitive programming tasks, and allows developers of C2 applications to focus on application-level issues. In order to incorporate OTS components into a C2 architecture, they are wrapped inside framework components, as shown in Fig. 4. The framework supports a

2. While the style does not forbid synchronous communication, the responsibility for implementing synchronous message passing resides with individual components.

variety of implementation configurations for a given architecture: the entire resulting system may execute in a single thread of control, or each component may run in its own thread of control or operating system (OS) process.

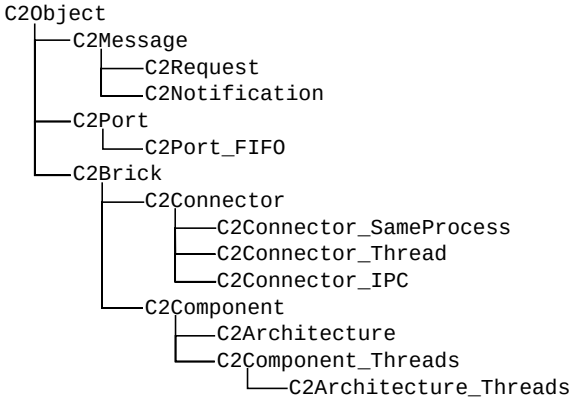


Fig. 3. C2 object-oriented class framework.

The framework has been implemented in C++ and Java;[3] its subset is also available in Ada. We have been able to successfully reuse the Q interprocess communication (IPC) library [MHO96] to enable message exchange between C2 components implemented in C++ and Ada. Similar functionality for Java C2 components is under development.

The Java implementation of the framework is particularly significant in that it represents the first step in our attempt to (partially) automate domain translation. A common form of interface mismatch between communicating components is different ordering of message parameters [TMA+96]. The Java implementation of C2 messages eliminates this problem by allowing components to access message parameters by name, rather than by position. We are currently exploring the possibility of incorporating other domain translation techniques into the framework.

III.B. ArchShell: A Tool for Runtime Manipulation of Software Architectures

ArchShell [Ore96] supports interactive construction, execution, and runtime modification of C2-style architectures. ArchShell is implemented in Java, using the Java C2 framework. It functions in a manner similar to the way in which a UNIX shell (e.g., csh) allows construction and execution of pipe-and-filter architectures. However, unlike a UNIX shell, ArchShell also supports modification of the architecture at runtime by dynamically loading and linking new architectural elements into the architecture. Furthermore, while the application is running, users can interactively send C2 requests and notifications to architectural elements.

ArchShell enables designers to experiment with components without building complete applications. Designers can construct partial architectures and experiment with their behaviors by sending events to components. Candidate components for reuse can be more easily understood and evaluated by integrating them into the application architecture and experimenting with the resulting runtime behavior. As a result, techniques for specializing and integrating OTS component may become more easily apparent.

3. The C++ and Java frameworks and several simple applications developed with them are available at http://www.ics.uci.edu/pub/arch/.

IV. Reusing OTS Components in a C2 Architecture

In order to evaluate C2's support for reuse, we have thus far conducted a number of exercises in which existing components of various granularities were integrated into C2-style architectures. [TMA+96] describes several initial reuse exercises, such as that of a spell checker. This section focuses on more recent efforts, in which the following OTS components were reused:

- Xlib graphics toolkit [SG87];
- Java AWT toolkit [CL96];
- SkyBlue constraint solver [San94];
- One-way formula constraint solver from the Amulet user interface system [MM95];
- JFox WWW browser [Wen96]; and
- JOP persistent object package [Rot96].

The remainder of this section describes each exercise in more detail, focusing on C2 features that directly contributed to its success.

IV.A. Reusing OTS Graphics Toolkits

C2's particular focus is on architectures having a significant GUI aspect. However, commercial user interface toolkits have interface conventions that do not match up with C2's notifications and requests. In a C2 architecture, the toolkit is always at or near the bottom, since it performs functions conceptually closest to the user [TMA+96]. As such, it must be able to receive notifications from components above it and issue requests in response. Typically, however, toolkits will generate events of the form "this window has been selected" or "the user has typed the 'x' key." These events need to be converted into C2 requests by C2 bindings and sent up the architectures. Conversely, notifications from a C2 architecture have to be converted to the type of invocations a toolkit expects.

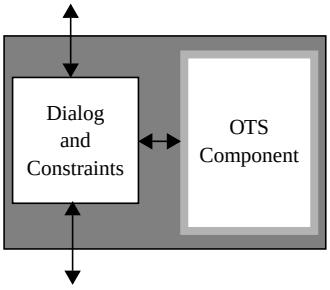


Fig. 4. An OTS component is wrapped inside a C2 component.

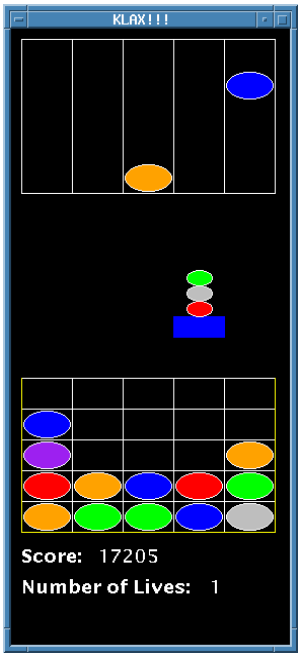
In order for these translations to occur and be meaningful, careful thought has to go into the design of the bindings to the toolkits such that they contain the required functionality and are reusable across architectures and applications. Our experience with reusing OTS components in C2 architectures and with building bindings for Motif and OpenLook in Chiron-1 [TNB+95] suggested an approach depicted in Fig. 4: a C2 component is created such that the graphics toolkit becomes its internal object, while the C2 message traffic is handled by its dialog. Graphics binding's dialog accepts notifications from C2 components above it and reifies them as calls to toolkit methods. It also transforms user events, generated in the graphics toolkit, into C2 requests. A C2 component's internal architecture, its reliance on message-based communication, and no assumption of shared address space or thread of control eliminate the need to internally modify the toolkits in any way.

We have built C2 bindings for two graphics toolkits: Xlib [SG87] and Java's AWT [CL96]. The Xlib and AWT bindings are subclasses of C++ and Java frameworks' *component* classes,

respectively. This enables their easy integration into C2 architectures built using the two frameworks. The bindings are extensible. They currently support manipulation of windows, panels, buttons, text fields, arcs, lines, ovals, rectangles, and text strings. They export identical message interfaces, making them interchangeable in an architecture.[4]

IV.B. Reusing OTS Constraint Solvers

We successfully integrated two externally developed constraint solvers into a C2 architecture: SkyBlue [San94] and Amulet's one-way formula solver [MM95].[5] In doing so, we were able to create several constraint maintenance components in the C2 style, enabling the construction of a large family of applications. This subsection summarizes our experience with SkyBlue and Amulet and the results we obtained; a detailed description of the project can be found in [MT96].



KLAX Chute
Tiles of random colors drop at random times and locations.

KLAX Palette
Palette catches tiles coming down the Chute and drops them into the Well.

KLAX Well
Horizontal, vertical, and diagonal sets of three or more consecutive tiles of the same color are removed and any tiles above them collapse down to fill in the newly-created empty spaces.

KLAX Status

Fig. 5. A screenshot and description of our implementation of the KLAX video game.

The architecture into which SkyBlue and Amulet were integrated is a version of the video game KLAX. A description of the game is given in Fig. 5. The system architecture is depicted in Fig. 6. The components that make up the KLAX game can be divided into three logical groups. The *game state* components are at the top of the architecture. These components receive no notifications, but respond to requests and emit notifications of internal state changes. The *game logic* components request changes of game state in accordance with game rules and interpret the resulting notifications to determine the state of the game in progress. The *artists* also receive notifications of game state changes, causing them to update their depictions. Each artist maintains a set of abstract graphical objects which, when modified, send state change notifications in hope that lower-level graphics components (in this case, *GraphicsBinding*) will render them. *GraphicsBinding*, in turn, translates user events, such as a key press, into requests to the artist components.[6]

We identified the following UI constraints in KLAX:

- *Palette Boundary:* The palette cannot move beyond the chute and well's left and right boundaries.
- *Palette Location:* Palette's coordinates are a function of its location and are updated every time the location changes.[7]
- *Tile Location:* The tiles which are on the palette move with the palette. In other words, the x coordinate of the center of a tile always equals the x coordinate of the center of the palette.
- *Resizing:* Each game element (chute, well, palette, and tiles), is maintained in an abstract coordinate system by its artist. This constraint transforms those abstract coordinate systems, resizing the game elements to have the relative dimensions depicted in Fig. before they are rendered on the screen.[8]

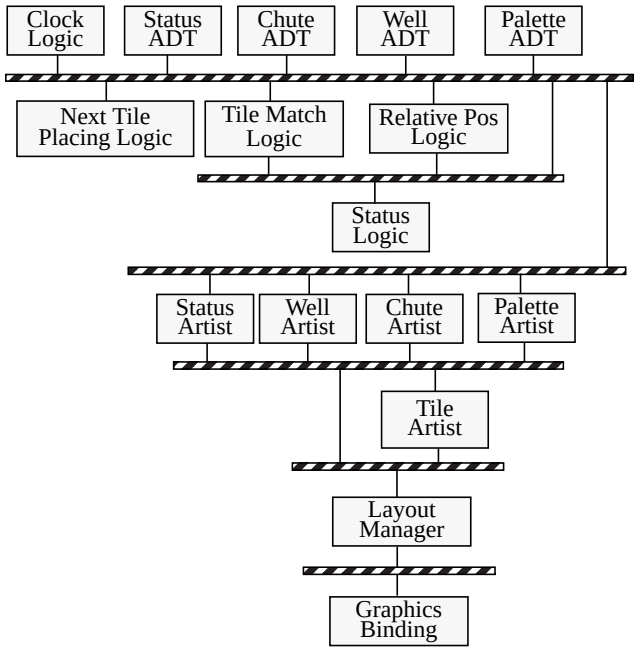


Fig. 6. Conceptual C2 architecture for KLAX.

IV.B.1. Integrating SkyBlue and Amulet with KLAX

In the original design, the *LayoutManager* component was intended to serve as a constraint manager. However, in the initial implementation, the constraints were solved with in-line code locally in the *PaletteADT* and *PaletteArtist* and the sole purpose of the *LayoutManager* was to properly line up game elements on the screen. The implemented version of the *LayoutManager* also placed the burden of ensuring that the game elements have the same relative dimensions on the developers of the *PaletteArtist*, *ChuteArtist*, and *WellArtist* components.

In order to render *LayoutManager*'s implementation more faithful to its original design, we decided to incorporate constraint management functionality into the component, as shown in Fig. 7.

4. Note that identical interfaces are not a requirement; two bindings with different interfaces could easily be substituted for one another by using a domain translator.
5. For the purpose of brevity, in the remainder of the paper Amulet's one-way formula constraint manager will be referred to simply as "Amulet."
6. A detailed discussion of the KLAX architecture is given in [TMA+96]
7. Location is an integer between 1 and 5.
8. This constraint would be essential in a case where the application is composed from preexisting components supplied by different vendors. A similar constraint could also be used to accommodate resizing of the game window, and hence of the game elements within it.

SkyBlue, the OTS constraint solver reused here, has no knowledge of the architecture of which it is now a part. It maintains the constraints, while all the request and notification traffic is handled by *LayoutManager*'s dialog. *LayoutManager* thus became a constraint management component in the C2 style that can be reused in other applications by only modifying its dialog to include new constraints.[9]

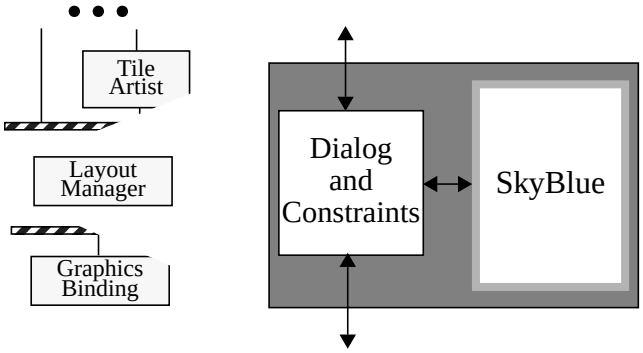


Fig. 7. The SkyBlue constraint management system is incorporated into KLAX.

PaletteADT, *PaletteArtist*, *ChuteArtist*, and *WellArtist* also needed to be modified. Their local constraint management code was removed. Furthermore, their dialogs and message interfaces were expanded to notify *LayoutManager* of changes in constraint variables and to handle requests from *LayoutManager* to update them. Only 11 new messages were added to handle this modification of the original application and there was no perceptible performance degradation. The entire exercise was completed by one developer in approximately 45 hours.

This initial exercise enabled us to test C2's support for component substitutability and localization of change. In [MORT96] we showed that behaviorally equivalent C2 components can always be substituted for one another and that behavior-preserving modifications to a component's implementation have no architecture-wide effects. To demonstrate this, we substituted SkyBlue with Amulet inside *LayoutManager*. C2 concepts, such as component-based development, message-based communication, and substrate independence, as well as the internal architecture of a C2 component, made this a relatively simple task that was completed by one developer in 75 minutes. As anticipated, no architecture-wide changes were necessary. The look-and-feel of the application remained unchanged and there was again no performance degradation.

IV.B.2. KLAX Application Family

Integrating SkyBlue and Amulet with KLAX provided an opportunity for building multiple versions of *PaletteADT*, *PaletteArtist*, and *LayoutManager* components. The two integrations described above resulted in three versions of the *LayoutManager*: the original, SkyBlue, and Amulet versions. Two versions each of *PaletteADT*, *PaletteArtist*, *ChuteArtist*, and *WellArtist* were created as well: original components maintaining local constraints with in-line code and components whose constraints were managed elsewhere in the architecture.

Table 1: Implemented Versions of PaletteADT, PaletteArtist, ChuteArtist, WellArtist, and LayoutManager KLAX Components

	Version Number	Constraints Maintained	Constraint Managers
Palette ADT	1	Palette Boundary	In-Line Code
	2	None	None
	3	Palette Boundary	SkyBlue
	4	Palette Boundary	Amulet
Palette Artist	1	Palette Location Tile Location Palette Size	In-Line Code
	2	None	None
	3	Palette Location Tile Location	SkyBlue
	4	Palette Location Tile Location	Amulet
Chute Artist	1	Chute Size	In-Line Code
	2	None	None
Well Artist	1	Well Size	In-Line Code
	2	None	None
Layout Manager	1	None	None
	2	All	SkyBlue
	3	All	Amulet
	4	Resizing	SkyBlue
	5	Resizing	Amulet
	6	All	SkyBlue & Amulet

The two initial integrations also suggested other variations of these components, such as replacing in-line constraint management code with SkyBlue and Amulet constraints in *PaletteADT* and *PaletteArtist* (see Footnote 9). Also, a version of *LayoutManager* was implemented that maintained only the *Resizing* constraint, in anticipation that other components will internally manage their local constraints. This resulted in a total of 18 implemented versions of the five components, as depicted in Table 1.[10]

The four versions of *PaletteADT* and *PaletteArtist*, two versions of *ChuteArtist* and *WellArtist*, and six versions of *LayoutManager*, described in Table 1, could potentially be used to build 384 different variations of the KLAX architecture. Three such variations were described above: (1) the original architecture, (2) the architecture with SkyBlue, and (3) with Amulet. Below, we discuss several additional implemented variations of the architecture that exhibit interesting properties.

Table 2: Multiple Instances of SkyBlue

Component	Version Number	Constraints Maintained	Constraint Managers
PaletteADT	3	Palette Boundary	SkyBlue
PaletteArtist	3	Palette Location Tile Location	SkyBlue
ChuteArtist	2	None	None
WellArtist	2	None	None
LayoutManager	4	Resizing	SkyBlue

In the architecture depicted in Table 2, multiple instances of

9. In the remainder of the paper, when we state that a constraint solver is "inside" or "internal to" a component, the internal architecture of the component will resemble that of the *LayoutManager* from Fig. 7.

10. In the remainder of the paper, a particular version of a component will be depicted by its name followed by the version number (e.g., *PaletteADT-2*).

SkyBlue maintain the constraints in different KLAX components. A related variation of the architecture is one where multiple constraint managers are employed in a single architecture.[11] Such an architecture is shown in Table 3. In this architecture, *Palette Boundary* and *Resizing* constraints are maintained by SkyBlue, and *Palette Location* and *Tile Location* by Amulet. Since the sets of constraint variables managed by the two solvers are disjoint, there are no interdependencies between SkyBlue and Amulet. Hence, this modification to the architecture was a simple one.

Table 3: Multiple Constraint Solvers

Component	Version Number	Constraints Maintained	Constraint Managers
PaletteADT	3	Palette Boundary	SkyBlue
PaletteArtist	4	Palette Location Tile Location	Amulet
ChuteArtist	2	None	None
WellArtist	2	None	None
LayoutManager	4	Resizing	SkyBlue

Particularly interesting are components that are used in an architecture for which they have not been specifically designed, i.e., they can do more or less than they are asked to do. This is an issue of reuse: if we build components a certain way, are their users (designers) always obliged to use them "fully"; furthermore, can meaningful work be done in an architecture if two components communicate only partially, i.e., certain messages are lost? The architectures described below represent a crossection of exercises conducted to better our understanding of partial communication and partial component service utilization in C2.

A variation of the original architecture was implemented by substituting *LayoutManager*-2 into the original architecture, as shown in Table 4. *LayoutManager*-2's functionality remains largely unused as no notifications are sent to it to maintain the constraints. The application still behaves as expected and there is no performance penalty. Note that this will not always be the case: if *LayoutManager*-2 was substantially larger than *LayoutManager*-1 or had much greater system resource needs (e.g., its own OS process), the performance would be affected.

Table 4: None of *LayoutManager*'s Constraint Management Functionality is Utilized

Component	Version Number	Constraints Maintained	Constraint Managers
PaletteADT	1	Palette Boundary	In-Line Code
PaletteArtist	1	Palette Location Tile Location Palette Size	In-Line Code
ChuteArtist	1	Chute Size	In-Line Code
WellArtist	1	Well Size	In-Line Code
LayoutManager	2	All	SkyBlue

Another architecture that was built is shown in Table 5. This exercise explored heterogeneous approaches to constraint maintenance in a single architecture: some components in the architecture maintain their constraints with in-line code (*WellArtist* and *ChuteArtist*), others maintain them internally using SkyBlue (*PaletteADT*), while *PaletteArtist*'s constraints are maintained by an external constraint manager. *LayoutManager*-2 is still partially utilized, but a larger subset of its services is used than in the preceding architecture.

Table 5: *LayoutManager*'s Constraint Management Functionality is Only Partially Utilized

Component	Version Number	Constraints Maintained	Constraint Managers
PaletteADT	3	Palette Boundary	SkyBlue
PaletteArtist	2	None	None
ChuteArtist	1	Chute Size	In-Line Code
WellArtist	1	Well Size	In-Line Code
LayoutManager	2	All	SkyBlue

In the architecture shown in Table 6, *PaletteADT* expects that some external component will maintain the *Palette Boundary* constraint. However, *LayoutManager*-1 does not understand and therefore ignores the notifications sent by *PaletteADT* (partial communication). Movement of the palette is thereby not constrained and the application behaves erroneously: the palette disappears when moved beyond its right boundary; the execution aborts when the palette moves beyond the left boundary and the *GraphicsBinding* component (see Fig. 6) attempts to render it at negative screen coordinates.

Table 6: *Palette Boundary* Constraint is not Maintained

Component	Version Number	Constraints Maintained	Constraint Managers
PaletteADT	2	None	None
PaletteArtist	1	Palette Location Tile Location Palette Size	In-Line Code
ChuteArtist	1	Chute Size	In-Line Code
WellArtist	1	Well Size	In-Line Code
LayoutManager	1	None	None

The above example architectures seem to imply that partial service utilization generally has no ill effects on a system, while partial communication does. This is not always the case. For example, an additional version of each component from the original architecture was built to enable testing of the application. These components would generate notifications that were needed by both components below them in the architecture and the testing harness. If a "testing" component was inserted into the original architecture, all of its testing-related messages would be ignored by components below it, resulting in partial communication, yet the application would still behave as expected.

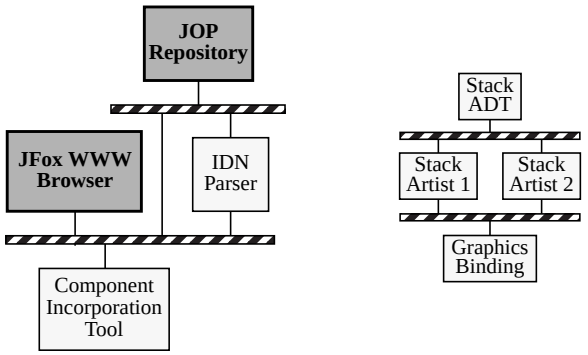
IV.C. Using OTS Components to Build a Simple Development Environment in the C2 Style

In [WRMT95] we discussed the requirements for a software component marketplace and argued the suitability of C2 as a basis for such a marketplace. In order to further explore some of those ideas, we devised a prototypical experiment in which a desired component would be located on the WWW, downloaded, and incorporated into an application built according to C2.

This experiment also represents an initial, though limited, attempt to apply C2 concepts to software tool integration. The simple environment used in this project was built in the C2 style, using the Java class framework discussed in Section III.A. The portion of

11. Combining multiple constraint solvers in a single system has only recently been identified as a potentially useful approach to constraint management [San94, MM95].

the environment architecture relevant to this discussion is shown in Fig. 8a. Two OTS components were reused in the environment: the JFox WWW browser [Wen96] and the JOP persistent object package [Rot96]. Both were wrapped inside C2 components in the manner depicted in Fig. 4.



(a) Environment Architecture (b) Application Architecture

Fig. 8. A simple environment built according to the C2 style and a C2 application architecture developed in the environment. OTS WWW browser and persistent object package were used in constructing the environment.

Using this environment, a software developer can search the WWW for a component with desired features. Once the component is found, it can be downloaded, in which case *JFox* emits a notification with the component's location. *ComponentIncorporationTool* receives the notification, unpackages the component and sends requests to *IDNParser* to parse the component's interface and to *JOP* to store the component's executable.[12] Once the interface definition is parsed, *IDNParser* sends a request to *JOP* to store the abstract syntax tree for the component's interface. At this point, the component can be accessed and used in a C2 application.

This environment was used by ArchShell, described in Section III.B, to download a stack visualization component and add it to an already executing application. The resulting architecture is shown in Fig. 8b.

Several C2 concepts contributed to the success of this exercise. As in previous cases, the internal architecture of a C2 component enabled relatively easy integration of JFox and JOP. In our design of the environment, both JFox and JOP were placed at the "top" of the architecture, as shown in Fig. 8a, so they had no dependencies on components above. This, combined with substrate independence, greatly simplified the implementation of their dialogs. Finally, C2's flexibility in supporting components with their own threads of control enabled us to incorporate JFox, which expects to execute in its own threads.

V. Discussion and Lessons Learned

The full potential of component-based software architectural styles cannot be realized unless reusing code developed by others becomes a common practice. A new architectural style can become a standard in its domain only if it makes reuse easier. We believe that C2 is such a style for GUI software, with the potential for broader applicability. C2's influences include a number of existing approaches to reuse. We have tried to build on their successes, while avoiding their shortcomings.

Krueger points out some common problems in basing reuse on software architectures [Kru92]. His criticism mainly applies to those approaches that do not identify higher-level abstractions applicable across applications. C2, on the other hand, is a style that attempts to exploit commonalities across systems and reuse successful structural and communication patterns.

Attempts at constraining reuse by focusing on architectures in particular domains of applicability (domain-specific software architectures) have been relatively successful, but have tended to be too restrictive. GenVoca [BO92] has been particularly successful in producing a large library of reusable components. However, those components have been custom built for the GenVoca style. In order to reuse them, one must adhere to GenVoca's formalism and its hierarchical approach to component composition, which may result in a high degree of dependency between communicating components. On the other hand, C2's style rules and underlying formalism [Med95, MTW96, MORT96] are more flexible; C2 eliminates assumptions of shared address spaces and threads of control, allows both synchronous and asynchronous message-based communication, and separates the architecture from the implementation.

Several aspects of object-oriented (OO) programming provide valuable lessons as well. In [MORT95] we demonstrate how concepts from OO typing can be applied to software architectures. The work on OO design patterns [GHJV95] has similarities with architectural styles. However, OO design patterns support reuse of structures at a much lower level of abstraction than do styles.

Garlan, Allen, and Ockerbloom classify the causes of problems developers commonly experience when attempting OTS reuse and give four guidelines for alleviating them [GAO95]. Our experience shows that C2 is well suited to address these problems. The first two guidelines deal with the internal architecture of OTS components, and are thus outside the scope of C2. The third guideline proposes techniques for building component adaptors, which is subsumed by C2 wrappers and domain translators. Finally, the authors emphasize the need for design guidance, which is a significant aspect of our approach to C2 [RWMT95, RR96].

Shaw discusses nine "tricks" for reconciling component mismatch in an architecture [Sha95]. Several of the tricks are related to reuse techniques employed in C2. For example, transformations, such as "Change A's form to B's form", "Provide B with import/export converters", and "Attach an adapter or wrapper to A," are subsumed by C2's wrappers and/or domain translators. The need for other transformations is eliminated altogether by C2 style rules. For example, "Make B multilingual" is unnecessary, as C2 assumes that components will be heterogeneous and multilingual.

The exercises discussed in Section IV have enabled us to devise a set of heuristics for OTS component integration in C2. The only assumption made across our exercises was that OTS components provided application programmable interfaces (APIs). As our experience with reusing OTS components grows, we expect that this list will be expanded and refined:

- If the OTS component does not contain all of the needed functionality, its source code must be altered. In our exercises, this was the case with the JFox WWW browser, which required a simple modification to include the necessary information in a message. In general, this is a difficult task, whose complexity is well recognized [Kru92, GAO95, MG96].[13]
- If the OTS component does not communicate via messages, a C2 wrapper must be built for it. This was the case with all components described in this paper.

12. We made several simplifying assumptions for the purpose of this exercise. One of them was that a downloaded component consisted of an executable and a definition of its interface. The interface is specified using a subset of C2 SADL, an architecture description language for C2 architectures [MTW96].

13. Note that the component can still be reused "as is" if the developers are willing to risk degraded or incorrect performance, due to partial communication and partial component service utilization in the architecture. This was the case with several variations of KLAX, discussed in Section IV.B.

- If the OTS component is implemented in a programming language different from that of other components in the architecture, an IPC connector must be employed to enable their communication. We have accomplished this task for C++ and Ada components using the Q software bus [MHO96], giving us confidence that we can do so for other languages. Other software bus technologies, such as [Rei90, Cag90], could also be used. We are also considering the use of software packaging technologies [Pur94, CP91], which decouple a component's functionality from its interfacing requirements and automate a significant portion of the work associated with adapting a component for use in new environments.
- If the OTS component must execute in its own thread of control, an inter-thread connector must be employed. This was accomplished in the case of the Java AWT graphics toolkit and the JFox browser.
- If the OTS component executes in its own process, an IPC connector must be employed. While we do not have direct experience with such components, we have implemented an IPC connector using Q [MHO96].
- If the OTS component communicates via messages, but its interface does not match interfaces of components with which it is to communicate, a domain translator must be built for it. Although we have done some preliminary work on domain translation in our Java class framework, this area needs further exploration.

The information above is summarized in Table 7.

Table 7: OTS Component Integration Heuristics for C2

Problem with OTS Component	Integration Method
Inadequate Functionality	Source Code Modification
No Message-Based Communication	Wrapper
Different Programming Language	IPC Connector
Different OS Process	IPC Connector
Message Interface Mismatch	Domain Translator

The series of exercises described in this paper demonstrate that C2 isolates changes inside components and limits any global effects of those changes through message-based communication. Furthermore, C2's principles of substrate independence and domain translation enable component substitutability. Finally, its component- and message-based nature allows partial communication and service utilization of components, which are essential to cost-effective reuse.

The direct benefit of this work is that we now have constraint management, WWW browser, persistent object, and graphics binding components, as well as IPC and inter-thread connectors, in the C2 style that will be reused across future applications. Beyond this, we have learned an invaluable lesson on the intricacies of incorporating OTS components into C2 architectures. We will build upon this experience in our exploration of other facets of C2.

VI. Future Work

One aspect of our future work involves attempting to incorporate into C2 architectures OTS components that exhibit characteristics different from those described above. In other words, we intend to gain more experience in every category depicted in Table 7. For example, a component that requires its own OS process would be a good candidate.

In addition, we intend to expand our development tool support for C2. The following tools are either in the design stage or already under construction:

- tools for (semi)automatic generation of domain translators and wrappers;
- tools for C2 component subtyping and type checking; and
- tools to support dynamic architecture changes.

Our long-term goal is to integrate all the tools into a C2 development environment.

VII. References

[Big94] T. J. Biggerstaff. The Library Scaling Problem and the Limits of Concrete Component Reuse. *IEEE International Conference on Software Reuse*, November 1994.

[BO92] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, pages 355–398, October 1992.

[BP89] T. J. Biggerstaff and A. J. Perlis. *Software Reusability*, volumes I and II. ACM Press/Addison Wesley, 1989.

[Cag90] M. R. Cagan. The HP Softbench Environment: An Architecture for a New Generation of Software Tools. *HP Journal*, pages 36-47, June 1990.

[CP91] J. Callahan, J. Purtilo. A Packaging System for Heterogeneous Execution Environments. *IEEE Transactions on Software Engineering*, pages 626-635, June 1991.

[CL96] P. Chan and R. Lee. *The Java Class Libraries: An Annotated Reference.* Addison-Wesley, 1996.

[GAO95] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch, or, Why It's Hard to Build Systems out of Existing Parts. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, WA, April 1995.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[GS93] D. Garlan and M. Shaw. *An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing, 1993.

[Kru92] C. W. Krueger. Software Reuse. *ACM Computing Surveys*, pages 131-183, June 1992.

[Med95] N. Medvidovic. Formal Definition of the Chiron-2 Software Architectural Style. Technical Report UCI-ICS-95-24, Department of Information and Computer Science, University of California, Irvine, August 1995.

[MG96] R. T. Monroe and D. Garlan. Style-Based Reuse for Software Architecture. In *Proceedings of the Fourth International Conference on Software Reuse*, Orlando, FL, April 1996.

[MHO96] M. J. Maybee, D. H. Heimbigner, and L. J. Osterweil. "Multilanguage Interoperability in Distributed Systems: Experience Report." In *Proceedings of the Eighteenth International Conference on Software Engineering*, Berlin, Germany, March 1996. Also issued as CU Technical Report CU-CS-782-95.

[MM95] R. McDaniel and B. A. Myers. Amulet's Dynamic and Flexible Prototype-Instance Object and Constraint System in C++. Technical Report, CMU-CS-95-176, Carnegie Mellon University, Pittsburgh, PA, July 1995.

[MORT96] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N.

Taylor. Using object-oriented typing to support architectural design in the C2 style. In *Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, San Francisco, CA, October 1996.

[MT96] N. Medvidovic and R. N. Taylor. Reuse of Off-the-Shelf Constraint Solvers in C2-Style Architectures. Technical Report UCI-ICS-96-28, Department of Information and Computer Science, University of California, Irvine, July 1996.

[MTW96] N. Medvidovic, R. N. Taylor, and E. J. Whitehead, Jr. Formal Modeling of Software Architectures at Multiple Levels of Abstraction. In *Proceedings of the California Software Symposium 1996*, pages 28-40, Los Angeles, CA, April 17, 1996.

[Ore96] Peyman Oreizy. Issues in the Runtime Modification of Software Architectures. Technical Report, UCI-ICS-96-35, University of California, Irvine, August 1996.

[Pur94] J. Purtilo. The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems*, pages 151-174, January 1994.

[PW92] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, pages 40-52, October 1992.

[Rei90] S. p. Reiss, Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, pages 57-66, July 1990.

[RR96] J. E. Robbins and D. Redmiles. Software architecture design from the perspective of human cognitive needs. In *Proceedings of the California Software Symposium (CSS'96)*, Los Angeles, CA, USA, April 1996.

[RWMT95] J. E. Robbins, E. J. Whitehead, Jr., N. Medvidovic, and R. N. Taylor. A Software Architecture Design Environment for Chiron-2 Style Architectures. Arcadia Technical Report UCI-95-01, University of California, Irvine, January 1995.

[Rot96] D. Rothwell. Java Object Persistence Package. http://www.magna.com.au/

[Sha95] M. Shaw. Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging. In *Proceedings of IEEE Symposium on Software Reusability*, April 1995.

[San94] M. Sannella. SkyBlue: A Multi-Way Local Propagation Constraint Solver for User Interface Construction. In *Proceedings of the Seventh Annual ACM Symposium on User Interface Software and Technology*, Marina del Ray, CA, November 1994, pages 137-146.

[SG87] R. W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, June 1987.

[TMA+95] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., and J. E. Robbins. A Component- and Message-Based Architectural Style for GUI Software. In *Proceedings of the 17th International Conference on Software Engineering (ICSE 17)*, Seattle, WA, April 1995, pages 295-304.

[TMA+96] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, pages 390-406, June 1996.

[TNB+95] R. N. Taylor, K. A. Nies, G. A. Bolcer, C. A. MacFarlane, K. M. Anderson, and G. F. Johnson. "Chiron-1: A Software Architecture for User Interface Development, Maintenance, and Run-Time Support." *ACM Transactions on Computer-Human Interaction*, pages 105–144, June 1995.

[Wen96] T. Wendt. JFox WWW Browser. http://www.uni-kassel.de/fb16/ipm/mt/java/jfox.html

[WRMT95] E. J. Whitehead, Jr., J. E. Robbins, N. Medvidovic, and R. N. Taylor. Software architectures: foundation of a software component marketplace. In David Garlan, editor, *Proceedings of the First International Workshop on Architectures for Software Systems*, pages 276-282, April 1995.

[YS94] D. M. Yellin and R. E. Strom. Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 176-190, Portland, OR, USA, October 1994.