Reducing Avionics Software Cost Through Component Based Product Line Development*

David C. Sharp
david.c.sharp@boeing.com

The Boeing Company
P.O. Box 516
St. Louis, MO 63166, USA

Title: Reducing Avionics Software Cost Through Component Based Product Line Development

Presenter: David C. Sharp

Track: Product Line Engineering/Y2K

Day: Thursday, 23 April, 1998

Keywords: Product Line, Reuse, Affordability, Component, Architecture

Abstract: Just as hardware integrated circuits, or components, can be used to inexpensively manufacture a product line of related hardware systems, reusable software components can be used to create software systems. This is accomplished by developing a common framework for a product line of related software systems that forms the component operating environment. A development architecture is presented based on our work using Object Oriented Analysis and Design techniques to create reusable software components that combine with aircraft specific customizations to form an avionics software system.

1. Introduction

In 1995, an initiative was launched at McDonnell Douglas (now merged with The Boeing Company) to assess the potential for reuse of operational flight program (OFP) software across multiple fighter aircraft platforms, and to define and demonstrate a supporting system architecture based upon open commercial hardware, software, standards and practices.[1] This initiative produced a set of reusable object oriented navigation software, which was flight tested on several different aircraft platforms hosted on different hardware configurations (both MIPS® and PowerPC® based).

The following year, this validation of the product line approach led to broader application of the techniques to the tactical aircraft mission processing domain. This paper describes key aspects of the component based logical architecture developed therein.

1.1 Software Project Goals

As illustrated in Figure 1, two primary software project level goals were defined:

* Presented at the Software Technology Conference, April 1998.

1. Development of a reusable architecture framework for the product line.
2. Development of reusable application components for the product line.

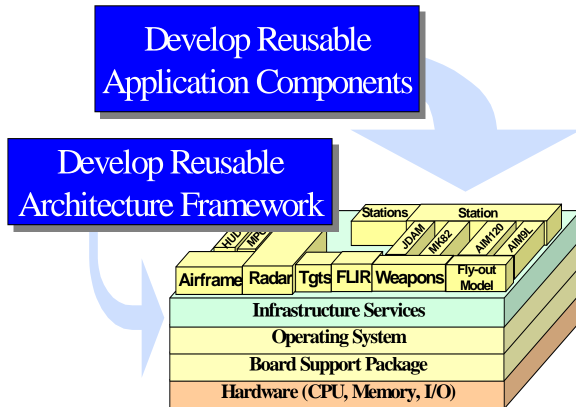

Figure 1: Software Project Goals

The project follows a two-pronged strategy: establish an architecture foundation that defines the overall system structure and component environment, and then create application components that reside therein. The combination of these two techniques provides the basis for an affordable approach to product line avionics software. This strategy follows Johnson's approach of decomposing reusable software systems into a framework and a component library.[2]

1.2 Architecture Goals

Given the driving project goals, there are two key goals of the software architecture itself:

1. Contain change.
2. Maximize reusability.

Naturally, there are many types of change that appear in a complex avionics mission computing product line:

1. Differences in avionics subsystems (e.g. radar, cockpit controls and displays, INS)
2. Differences in mission computing hardware – both in terms of number of processors and in terms of the exact microprocessor chosen.
3. Differences in system requirements.

In terms of the changes that need to be anticipated during design, system differences within the product line are similar to traditional changes to a single system. Therefore, in a product line project, there are two types of differences that must be contained:

1. Temporal differences – the evolution of a system over the course of time, and
2. Simultaneous differences – the differences between multiple products in the product line that must be concurrently supported.

The established adage of "encapsulate change" can be somewhat broadened to "encapsulate differences" when in a product line context.

Maximizing reusability relates to minimizing cost in a reuse-driven affordability process. One corollary to this goal is that the developed software must be granular enough that specific aircraft products can select the exact software elements applicable to their system. This requires elements with the ability to be plugged together to create a deliverable system.

Note that these architecture goals are a direct result of the original goal to develop software applicable to a full product line.

1.3 Patterns

Patterns have been widely used in the object-oriented community to capture best practices. They document the forces and issues providing the answer to the frequently

asked question of *why* software was designed a certain way. Documenting the tradeoffs made while arriving at a solution guides future development of the produced artifact and reuse of the techniques in other areas. Understanding the consequences of a design choice is especially key in those cases where multiple potential solutions exist whose selection is dependent on specifics of the problem context. It is in these situations that patterns prove most valuable.

There are several different classification techniques, or taxonomies, of patterns in the literature. Gamma et al. divide their design patterns into three categories[3]:

1. Creational – patterns used in flexibly constructing objects
2. Structural – patterns outlining class or object composition
3. Behavioral – patterns describing how objects interact and distribute responsibility

Schmidt[4] and others classify patterns as:

1. Tactical – patterns that resolve localized design forces in an application
2. Strategic – patterns that address application-wide issues and thus have broad impact

and also as:

1. Domain Independent – patterns that cross application domains
2. Domain Dependent – patterns applicable to a specific application domain

Buschmann et al. break patterns into the following divisions[5]:

1. Architectural – patterns providing the overall skeleton of a system
2. Design – patterns dealing with localized design choices
3. Idioms – patterns of usage within a particular language

Some of the numerous relationships between these classifications are:

- Design patterns are often tactical
- Architectural patterns are often strategic
- Strategic patterns are often domain dependent

Given this context, the development process relies heavily on leveraging domain independent patterns developed in other organizations and growing domain specific patterns within the home organization, particularly strategic ones. This combined set of patterns then forms a shared expertise base, which guides high level architecture development as well as individual development across the full organization.

2. Top Level Architecture

Two architectural patterns underlie the top level software architecture: the *Layers* pattern[6] and the *Model-View-Controller* pattern[7]. Using these patterns helps *generate* the overall architecture for the system[8].

2.1. Layers Pattern

The *Layers* pattern describes the widely used method of basing software systems on layers of abstraction and insulation. Low layers in the system provide hardware and operating system level facilities. Higher layers provide application functionality. The low layers form the foundation that insulates the application from specifics of the hardware configuration. This is an essential characteristic of a product line software system that is targeted to varying hardware configurations.

A key characteristic of layered architectures is that all dependencies flow downward. This helps avoid cyclic dependencies, and avoids low level

abstractions from depending on high-level application considerations. Jacobson et al. mention layered architectures as a central aspect of reusable software systems.[9]

2.2. Model-View-Controller Pattern

The Model-View-Controller (MVC) pattern was developed within the Smalltalk community as an architecture for graphical user interface based systems. Its architecture is shown in Figure 2, and is itself a layered architecture.

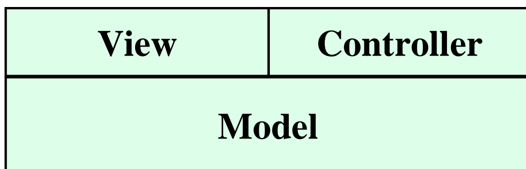

Figure 2: Model-View-Controller Architecture

One of the important characteristics of the MVC architectural pattern is the separation of the domain model from the user interface utilized to present domain information to users and allow them to manipulate it. This allows multiple views (i.e. displays) of the same information. It also separates the relative stability of domain models from the relative instability of user interfaces, resulting in better change encapsulation.

2.3. Top Level Architecture

The top level software architecture derives from applying the two aforementioned architectural patterns. In the case of the Layers pattern, it is applied recursively in hierarchical fashion. In all cases, a *relaxed* version of the Layers architecture is applied where a higher level layer may bypass intermediary layers and access low layers directly where necessary.

Figure 3 shows the top level architecture. The Model layer represents the core domain application as defined in the MVC architecture. The Operator layer contains both the View and Controller portions of the MVC pattern, sometimes referred to as a *Document-View* variant of the MVC architecture.[10]

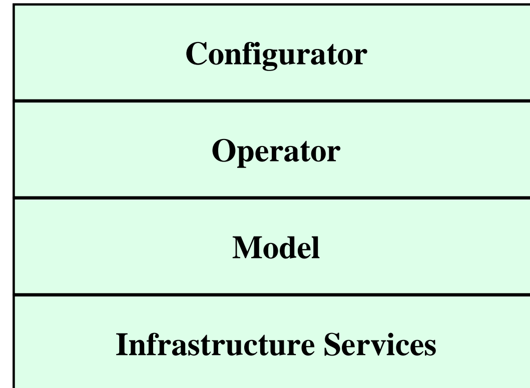

Figure 3: Top Level Layered Architecture

To these MVC based application layers, a foundational Infrastructure layer is added to provide a CORBA based run-time system and insulate the application from hardware configuration specifics. At the top, a Configurator layer is added which represents a notional "software system circuit board" and is responsible for instantiating all of the objects and establishing their relationships. The Configurator layer is responsible for taking all of the applicable software elements and plugging them together into a functional system.

Especially at the architectural level, the boundaries between software elements should foster change encapsulation and separate independent concerns. This inspection reveals the following benefits of the top-level architecture:

- The *Infrastructure Services-Model* boundary isolates hardware, run-time, and operating system issues from the application.

- The *Model-Operator* boundary isolates the core application from user interface volatility.
- The *Operator-Configurator* boundary isolates the software elements from their integration into a complete system.

2.4. Model Architecture

As stated earlier, the Layers pattern is applied recursively. One example of this is within the top level Model layer. Figure 4 illustrates this second level layered architecture.

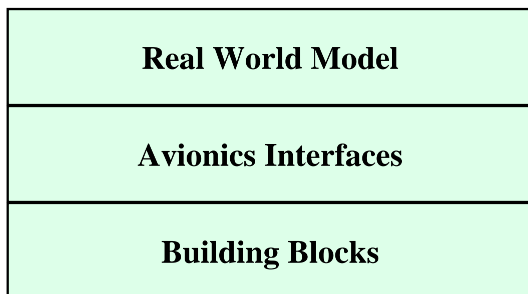

Figure 4: Model Layered Architecture

In this case, the Building Blocks layer represents general-purpose objects like vectors and matrices. Avionics Interfaces represents the mission computing interface to other avionics subsystems. The Real World Model contains the software representation of the problem domain. Examples of elements in the Real World Model are weapons, stations, targets, the ownship airframe, waypoints, and routes.

Applying the boundary inspection process as before at this level reveals the following benefits:

- The *Building Blocks-Avionics Interfaces* boundary separates more general-purpose domain independent elements from domain specific elements.
- The *Avionics Interfaces-Real-World Model* boundary forms a key separation isolating product line elements from the specifics of a single avionics system on a particular aircraft. This boundary is directly related to the original goal of containing avionics system specifics.

Summarizing, combining repeated application of these architecture patterns with domain specific considerations develops the overall system structure and leverages proven architecture experience within the product line software architecture.

3. Operational Flight Program Component Overview

Once the overall logical architecture structure is established, issues concerning software element development within this architecture surface. An important consideration is the granularity of the software elements. If an extremely fine granularity is chosen, the canonical software elements can be defined as being individual objects. This results in a multitude of elements that must be configured and integrated at the Configurator level. Near the other extreme, a single software element can be created for each of the layers. This fails, however to provide the configurability required by individual aircraft projects both in terms of independently selecting software elements applicable to their system and in allowing any required customizations to those elements. As Roberts and Johnson note in their Fine-Grained Objects pattern,[11] making software elements more reusable tends to make them finer-grained. The opposing force is that systems with more elements are more difficult to understand.

In the subject software system, the term Component was adopted to represent a software element somewhat in the middle of

the granularity continuum. Given the sometimes ambiguous usage of the term, and the need to clearly specify its meaning within the project, a set of defining patterns was developed. Because of their pervasive usage within the system, they are classified as strategic patterns. These patterns can be simplistically stated as providing software elements that are as large as possible while still being reusable within the product line.

These patterns form a small language whose names can be used within the organization to easily communicate and guide development. There are potentially many different types of components that fit in this language, including:

1. Component – the basic component definition that applies to all of the more specific types
2. Configurable Component – components that allow customization by users
3. Distributable Component – components that are distributable when in a multi-processing hardware configuration
4. Real Time Component – components that provide real time quality of service information for scheduling purposes[12]
   a) Active Component – components that have specific requirements under which they must run
   b) Passive Component – components that execute only as a service to other components

This paper focuses on the foundational Component pattern itself and the Configurable Component pattern.

3.1. Component Pattern

Roughly following the pattern outline of Gamma et al.[13], the following sections define the Component pattern.

3.1.1 Intent

Define software entities which are similar to objects but which can be composed of multiple smaller objects that together provide a set of services to a client.

3.1.2 Motivation

In multithreaded, multiprocessor applications, objects must deal with a number of complexities. Among these complexities are concurrent threads, shared resources, and multiple address spaces. In addition, large object oriented applications can easily comprise thousands or tens of thousands of objects. Components provide a practical way to group closely coupled objects that together provide a set of services to other objects. Grouping objects in this way allows for localization of certain concerns and provides a higher level of abstraction when considering software collaboration.

3.1.3 Applicability

Use the Component pattern when

- closely coupled objects provide a set of cohesive services to clients,
- a group of objects is reusable as a single entity, and
- especially when collaborating objects share similar synchronization requirements or thread priorities (e.g. processing rate)

3.1.4 Structure

Figure 5 shows the Component pattern structure using the Unified Modeling Language (UML).

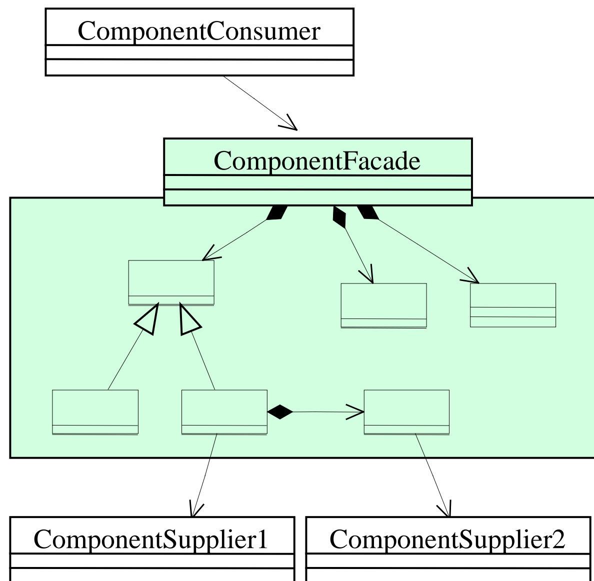

Figure 5: Component Pattern Structure

3.1.5 Participants

- **ComponentConsumer**
  - uses the component via the facade
- **ComponentFacade**
  - provides the public interface for the contained objects

3.1.6 Collaborations

- Consumers communicate with the component via the façade, which forwards them to the appropriate contained objects as applicable.
- Consumers do not access internal objects directly without first going through the facade.

3.1.7 Consequences

The Component pattern has the following benefits and drawbacks:

1. *Encapsulates change.* Promotes weak coupling between components by defining a clear interface via the component façade. The façade actually provides an interface to potentially many enclosed objects.
2. *Defines distribution boundaries.* Components form the potentially distributable software entities at a level higher than individual objects. Note that components are not always distributable, but all distributable items are components. Objects within components are guaranteed to reside within the same address space, fully supporting pointer sharing.
3. *Localizes concurrency control.* Localizes concurrency control issues to the component facades so that most objects do not have to be concerned with thread locking mechanisms.
4. *Identifies pluggable entities.* The pluggable software entities, which developers may choose to incorporate or not incorporate into a specific executable, are represented by components. While some components are foundational and essential, the majority is selectable and pluggable.
5. *Enhances reusability.* Makes it easier to configure resulting executables by reducing the number of software entities that must be considered.
6. *Reduces complexity.* Similar to the reason for *Enhances reusability*, Component reduces complexity by reducing the amount of detail that component users must consider.

3.1.8 Implementation

Consider the following issues when implementing the Component pattern:

1. *Allowed calls.* No external component is allowed to call member functions of objects internal to the component without first going through the facade. Doing otherwise defeats the goal of localizing concurrency control and providing a single public interface for the whole component. Objects within the component are allowed to invoke member functions on external component façades. Summarizing, *calls in* go through the façade. *Calls out* have no such restrictions.
2. *Contained objects.* Components can contain references to internal objects. If a façade returns a reference to another object for direct use by a consumer, then some other mechanism must be provided for serialization, such as:
   a. the returned object must manage its own concurrency mechanism, etc, and therefore takes on the responsibilities of a component facade, or
   b. the component façade must provide a request/release locking mechanism for access to the reference to ensure that access is serialized.
3. *Façade usage*. Objects within the component should not call public methods in the Façade. Since the façade normally contains the concurrency locking mechanism, components are typically not reentrant. If an internal object calls a façade member function, it would potentially block indefinitely waiting for the current thread (its own) to release the lock. For this reason, the façade should not contain algorithms or other functionality in public member functions that objects internal to the component would want to invoke.

3.1.9 Related Patterns

This pattern is a specialization of the Façade pattern.[14] Façade pattern issues apply here as well.

3.2 Configurable Component Design Pattern

The following sections define the Configurable Component pattern.

3.2.1 Intent

Define components that are configurable to increase reusability.

3.2.2 Motivation

There are many cases where multiple components can share interfaces but where all or part of the implementations differ. Configurable components provide all of the shared implementation and allow flexible incorporation of differences.

This establishes a new kind of component interface for the purposes of plugging in customizations. The original notion of a component interface is referred to as the *User Application Programmer Interface (API)*. This interface provides the services used during normal component operation. Configurable Components add the idea of a *Configuration API*, which allows component creators (referred to as Configurators) to customize behavior. This *Configuration API* is used during component initialization. Figure 6 offers a conceptual view of a configurable airframe

component representing the navigational state of the aircraft.

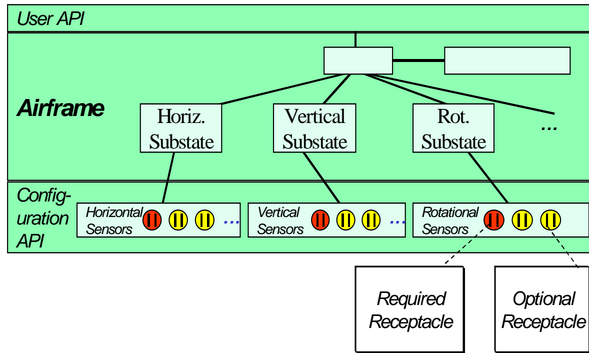

Figure 6: Conceptual Airframe Component

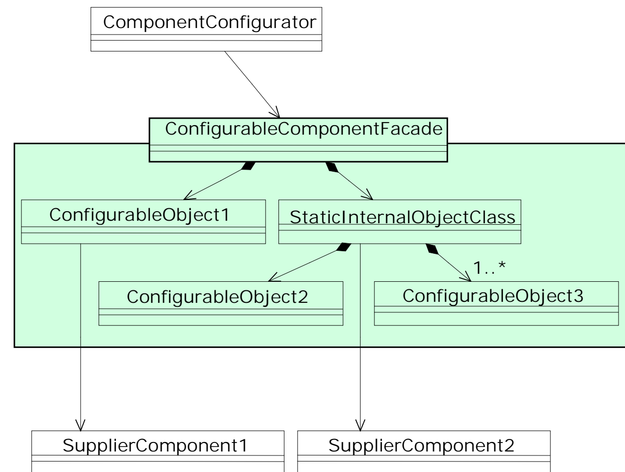

Figure 7: Configurable Component Pattern Structure

3.2.3 Applicability

Use the Configurable Component pattern when all of the following hold true:

- components in different applications or within a single application represent the same logical entity and can share the same interface
- flexible customization of the components enables or increases reuse, either within a single application for different component objects or between different applications
- interfaces to the differences can be defined
- the additional software or architecture reuse justifies the additional complexity

3.2.4 Structure

Figure 7 shows the Configurable Component pattern structure.

3.2.5 Participants

- **ComponentConfigurator**
  - passes configuration parameters to the ComponentFacade via the *Configuration API* to customize component behavior
- **ConfigurableComponent Facade**
  - instantiates objects directly for those internal objects that are not configurable
  - provides the *Configuration API* which defines the complete set of allowed customizations for the ComponentConfigurator to tailor operation as desired
- **SupplierComponent1, SupplierComponent2**
  - represent external components to which the Configurable Component must be related during construction

3.2.6 Collaborations

The collaborations listed here apply when the Abstract Factory pattern[15] is

applied for configuring the objects internal to the component as described in *3.2.8 Implementation*.

- The Configurator creates an Abstract Factory object, passing in any references to supplier components as constructor arguments.
- The Configurator creates the component by constructing the façade and providing the factory as a constructor argument.
- The façade invokes methods on the factory to create the internal objects and receive references to both the created internal objects and possibly external suppliers.
- Consumers then receive references to the created component via their configurators and invoke member functions in the *User API* to access operational capabilities of the component.

3.2.7 Consequences

The Configurable Component pattern has the following benefits and drawbacks:

1. *Increases reusability.* Allows components to be used in more situations by allowing customization.
2. *Identifies potential tailorings.* The *Configuration API* defines the receptacles that provide customization hooks. The receptacles allow flexible configuration of functionality or performance customizations via aggregation during component construction and initialization.

3.2.8 Implementation

Consider the following issues when implementing the Configurable Component pattern:

1. *What parts are configurable?* One of the most basic issues is identifying the component internals that should be made customizable. Directly contained objects are inflexible and are typically only acceptable for those parts that are extremely stable and would never require customization in any reasonably expected usage. Any internal objects that might require customization should be pluggable to aid reusability. The hooks provided by Configurable Components clearly indicate which parts are customizable.
2. *Tailoring functionality or performance.* Customizations may be desirable for any reason, not only to affect the component's functional behavior. Tailoring may also be advantageous in cases where simplified implementations can be plugged in for streamlined performance.
3. *Abstract Factory.* A clear and concise way to define the *Configuration API* is through use of the Abstract Factory pattern.[16] In this method, an abstract *factory* class is created that has member functions for creating or otherwise returning a reference to each configurable entity. Users then inherit from the base class and fill in the implementation with the specific desired configured items. The factory is passed into the façade constructor for use during component construction.

This method also helps avoid lengthy constructor argument lists when there are many configurable items.

4. *Required or optional tailoring.* The receptacles defined by the *Configuration API* may be either required or optional for component operation. Receptacles may be designed as optional by providing a default plug internal to the component or by virtue of the fact that the plug will simply not be used if not present. The later typically occurs when there is a variable length list of plugs in which case the component simply iterates over the provided plugs.
5. *Receptacles are relationships.* The receptacles in the *Configuration API* are actually special cases of relationships. In this case, they are relationships to pluggable customizations. Recognizing the different kinds of component relationships aids understanding:
   a. *Static relationships.* These are set during construction and remain unchanged during system operation. The *Configuration API* is solely comprised of these static relationships. There are two types of static relationships:
      i. To other components – these allow tailorable component *environments* where surrounding components are flexible, not tailorable component internals themselves.
      ii. To plugged in customizations – these are related to actual component tailorings where objects internal to the component are plugged in during construction.
   b. *Dynamic relationships.* These are changed during normal system operation. An example of this is selecting the target towards which a specific weapon is currently aimed. Since these relationships are an intrinsic part of using the component, they are considered part of the *User API*.
6. *Pushing or pulling plugs.* Plugs may be provided either by a configurator passing the plugs in during construction or by the component retrieving them itself. Pushing plugs makes the configurator smarter at the expense of the component. Conversely, pulling plugs makes the component smarter. Since configurators are focused on the component level, and the component façade is more concerned with issues internal to the component, the following recommendation is made:
   a. Static relationships to other components are established by passing the references into the component façade constructor.
   b. Static relationships to plugged in objects and the creation of the objects themselves are provided by the configurator creating an Abstract Factory and passing its reference to the component façade for use during component construction. This re-

duces configurator complexity, localizes the creation issues associated with a specific component, and clearly identifies to users what items are tailorable within the component.

3.2.9 Related Patterns

The Abstract Factory pattern is used as the means for flexibly creating objects internal to components.

4. Conclusion

This paper focuses on two important characteristics of the software architecture:

- Layered Architecture
- Medium Grained Architecture

It is important to note that both of these characteristics derive directly from the original goals of encapsulating change and maximizing reuse. The layering aspect especially relates to developing an application that is independent of the hardware and avionics specifics of a single aircraft. The medium granularity of the components provides a system that can be deployed on varying numbers of processors in single processor or distributed hardware configurations. Both of these goals directly relate to the original decision of developing a product line software system.

Patterns form a very powerful technique for reusing and spreading *expertise*. The pattern driven architecture described herein leverages previous industry experience to address the original goal of defining a software architecture applicable to a fighter aircraft mission processing product line. The Component pattern language defines the reusable software elements and provides the complete development community with understanding of the underlying rationale, tradeoffs, and implementation concerns involved. The resulting component library embodies reusable application elements available for specific aircraft products. All of these elements, including:

- Architecture,
- Components,
- Patterns, and
- Developer expertise

provide reusable artifacts, which are exploited to reduce life cycle avionics software cost.

5. Acknowledgements

The concepts described herein were developed within the stimulating confines of the project Software Core Architecture Team. Their contributions, and especially the many insights provided by founding member Bryan Doerr, are gratefully acknowledged. Our work also reflects an extremely fruitful collaboration with Dr. Douglas C. Schmidt and his colleagues at Washington University in St. Louis, Missouri. Thanks also go to Tim Popp, who authored the original abstract for this paper.

---

[1] Winter, Don C., "Modular, Reusable Flight Software For Production Aircraft", *15th AIAA/IEEE Digital Avionics Systems Conference* Proceedings, October, 1996, p. 401-406.

[2] Johnson, Ralph, *How to Develop Frameworks*, Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 1997.

[3] Gamma et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995, p. 10.

[4] Schmidt, Douglas C., "A Family of Design Patterns for Application-Level Gateways", *Theory and Practice of Object Systems*, Wiley & Sons, Vol. 2, No. 1, December 1996.

[5] Buschmann et al., *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley & Sons, 1996, p. xiv.

[6] Ibid., p. 31-52.

[7] Ibid., p. 125-144.

[8] Beck, Kent, and Johnson, Ralph, "Patterns Generate Architectures", European Conference on

Object-Oriented Programming (ECOOP) Proceedings, 1994.

[9] Jacobson et al, *Software Reuse: Architecture Process and Organization for Business Success*, Addison-Wesley, 1997, p. 170-212.

[10] Buschmann, p. 140.

[11] Roberts, Don, and Johnson, Ralph, "Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks", *How to Develop Frameworks*, Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 1997.

[12] Harrison, Timothy, Levine, David, and Schmidt, Douglas, "The Design and Performance of a Real-time CORBA Event Service", Conference on Object-Oriented Programming, Systems, Languages, and Applications Proceedings, October 1997, p. 184-200.

[13] Gamma et al., p. 6-8.

[14] Ibid., p. 185-193.

[15] Ibid., p. 87-95.

[16] Ibid., p. 87-95.