

Pattern Usage in an Avionics Mission Processing Product Line

David Sharp and Wendy Roll, The Boeing Company, St. Louis, MO

1 Introduction

Within the Boeing Company's Phantom Works research and development organization the Open Systems Architecture (OSA) group has been researching and developing component-based mission critical avionics product line software since 1996 as part of the Bold Stroke project. The technologies developed by OSA have been transitioned to several platforms within the company. Patterns were used extensively to resolve many of the challenges inherent in developing large object oriented real-time embedded systems for a reusable software product line. This paper will explore those patterns, how they evolved and how they are currently used in production programs.

2 Application Domain Characteristics

The mission processing domain for military aircraft is characterized by a set of large-scale embedded applications that support the planning and execution of the aircraft's mission. Its basic role is to provide the user interface for the aircrew, including avionics system control and displays. Specifically, it manages the sensors on the aircraft; uses sensor data to determine integrated solutions depicting current and predicted status of both the aircraft and aircraft's situation; drives cockpit displays; processes pilot requests that tailor his display or command sensors and devices; and manages weapons delivery. Each individual program in this domain could require all or a subset of these functional characteristics. Safety critical aspects associated with flight control are typically handled by a separate system dedicated to that purpose.

3 Design Challenges in Avionics Mission Processing Architecture

Avionics mission processing systems are inherently large-scale, highly dynamic, distributed

real-time embedded systems, and as such exhibit a plethora of technical challenges.

From a development perspective, affordability, quality, and timeliness are critical parameters. The goal of developing a reusable product line accentuates these issues since the impact of decisions spans multiple products. Several distinct roles must be filled in this paradigm. Common Component Developers produce reusable component libraries for use throughout the product line, with possibly the inclusion of techniques for product specific tailoring. Project Specific Component Developers develop component libraries providing product specific features, as well as tailorings of common components. Finally, Project Specific Component Integrators select and configure common and product specific components to product a particular system, ensuring that all functional and extra-functional (e.g., performance, reliability) requirements are met. Given their size, over one hundred developers are frequently applied to the development of each mission processing system.

From a run-time perspective, there are a number of challenging forces. Performance and deadline requirements require careful attention to throughput, blocking, and scheduling to ensure that processing completes within periodic rates up to approximately 20 Hz. Reliability requirements require hardware redundancy, backup software modes, and pervasive error management. These and other qualities must be met in the context of a large-scale distributed system frequently exceeding one million lines of source code. Again, product line reuse considerations pose additional challenges. The architectures and frameworks must be flexible enough to expand and contract to different hardware platforms and applications. In our case, the architecture had to support single processor systems of a few hundred thousand lines of code up to multi-processor systems of roughly ten processors within two VME chassis connected via a Fibre Channel link. This necessitated balancing the needs of distributed and uni-processing systems.

4 Pattern Resolution to Challenges

Given these operational and developmental challenges faced by large development teams, problems must be solved in a repeatable way. Patterns have provided a formal way of resolving and communicating these solutions, as well as their applicability and rationale. Pattern development, as for our architecture and application software, has been iterative, evolving over time as our systems have matured.

Note that although most of the discussion centers around qualities such as flexibility and reuse, these issues always required balancing with that of real-time performance. Each of the solutions has been chosen based on specific design decisions that examine the pros and cons of available alternatives in resolving the forces present.

4.1 Structural Patterns

Structural patterns were used extensively and opportunistically within our components by component developers. We used structural patterns, however, to also solve some fundamental and pervasive architectural and design problems.

4.1.1 Layers

Due to the size of avionics systems and due to the product line reuse goals, we structured our system using the Layers pattern. We used a customized version of the Layers architectural pattern that surrounded the Model-View-Controllerⁱ pattern layers in between a Configurator layer at the top that controlled the configuration of the system, and an Infrastructure layer at the bottom which provided the run-time foundation for the application. The View and Controller layers were aggregated into a combined user interface layer named Operator as in the Document-Viewⁱⁱ variant of Model-View-Controller. The result is shown in Figure 1. We began using layering for our common software to represent different levels of abstraction and to contain change.

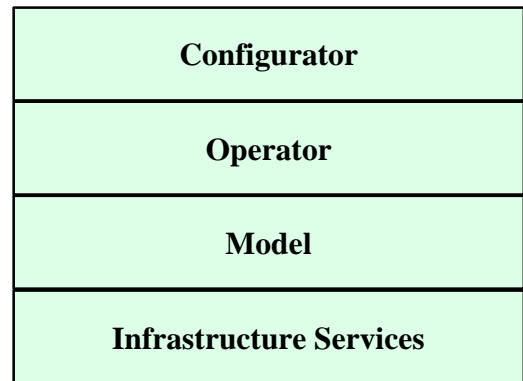


Figure 1: Top-Level Architectural Layers

When we began creating various project specific extensions to the common components as well as new project specific components, we developed layers into a second dimension, as shown in Figure 2. Specifically, a set of parallel isomorphic layers structured the project specific code that had one-way dependencies on common code in the common set of layers. In some respects, project specific software became a layer above common software, with an identical layered structure internally, but with relationships from lower project specific layers (e.g., Project Specific Model) to higher common layers (e.g., Common View) prohibited. In this way we were able to build multiple projects on top of the same set of common components by substituting the project specific set of layers. We referred to this as Multi-Dimensional Layers. It can be viewed as a compositional variant of the Dual Inheritance Hierarchyⁱⁱⁱ structure.

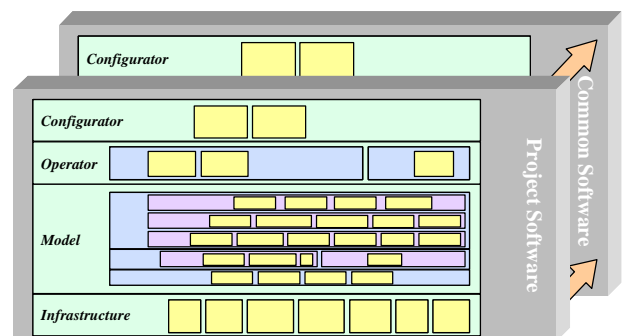


Figure 2: Multi-Dimensional Layers Structure

4.1.2 Component

In alignment with the Fine Grained Objects^{iv} pattern, our reuse objectives resulted in a greater number of smaller objects than might be otherwise possible. Smaller elements result in greater flexibility. This situation led to increased scalability concerns. To reduce the number of software elements which had to be considered, to raise the level of abstraction for developers and reusers, and to localize issues such as execution control, concurrency control, persistence, and distribution, we created a Component^v pattern that groups a set of objects presenting a cohesive set of services into a single entity. From the pattern, developers could determine what a component was and then leverage the encapsulation to centralize concerns. The Façade pattern was central to our Component pattern. While the concept of a component is strictly a part of the Component pattern, the implementation of the Component pattern relies heavily on the Façade pattern since the Façade became the component's external interface to other components. In addition, the façade became the interface for configuration and creation. Thus multiple logical interfaces (e.g., client, configuration, execution) were grouped into a single physical interface (i.e. the façade). One of the key challenges the Façade pattern aided was that of complexity. Viewing the system at the class level was an intractable problem challenge given that there were potentially many thousands of classes in the system. Components and facades allowed us to coarsen the granularity of pieces, easing both component software development and system configuration. In addition, change could be contained at a coarser level since client dependencies were localized to façade classes.

4.1.3 Adapter

When software components were first being developed and interfaced with the Event and Persistence Services provided by our Infrastructure middleware, the protocol for registration was unclear and potentially complex. Since the middleware services were written to support a wide range of real-time systems, they provided a great deal of flexibility. To provide a simplified and tailored architecture specific interface to application developers, and localize the interaction between the application and the middleware, we formed Object Adapters^{vi} around applicable middleware services.

In cases where Callbacks^{vii} were necessary to allow subsequent invocation of application components from the middleware, the External Polymorphism^{viii} pattern was used to provide reusable typesafe adapters.

In addition to these original motivations, the adapters provided important future flexibility as well. For example, performance concerns associated with fully scheduling each component led us to circumvent the full event service wherever wherever possible. In many cases the fact that suppliers and consumers were colocated and had the same rate and thread priority requirements removed the need for independent scheduling and dispatching. In these cases, the concrete adapter was changed but the application components were stable. These adapters performed callbacks to facades which were linked to our Façade pattern. The component façade was responsible for creating and initializing their adapters for events, persistence, distribution, or other aspects supported by middleware services.

4.1.4 Proxy

Flexible distribution was a key challenge in our product line design. The processor configurations for different project deployments varied greatly, from single processor configurations up to roughly ten processors on multiple backplanes. Therefore, components had to support distributed access when it was needed without penalizing those platforms that did not need it. In addition, the nature of mission processing functionality coupled with the patterns for data flow, discussed below, resulted in a system where data is typically accessed more than it is modified and that access is performed by the consumer pulling data from its suppliers. Given these characteristics, we used the Cache Proxy^{ix} pattern to optimize the performance of reads on physically remote components. A replication service was created to update the caches. Again, as with the adapters, we linked this pattern with the previously applied Component and Façade patterns. We targeted the component façade as the common interface between the original "master" component and the cache proxy so that clients could be transparently configured to use either depending on the distribution of a specific system.

4.2 Creational Patterns

Creation and initialization of systems within the product line presents unique and programmatically significant challenges. The Creational Patterns used in the product line have evolved as a result of other architectural patterns that have been adopted as well as the design forces described herein.

4.2.1 Factory Patterns

Although we explored other creational solutions such as the Factory Method^x pattern, the configurability and extensibility required by our product line led to use of the Abstract Factory^{xi} pattern. The concept of a component as a group of objects providing a cohesive set of services and also representing the reusable entities in the product line is a common one today, and was one of the initial structural decisions made in our architecture development. However, not only did these components need to be pluggable, but due to varying platform requirements they had to be extensible and variable. We paired the Abstract Factory pattern with the Façade^{xii} pattern such that façades had an interface accepting an abstract factory which provided product specific tailoring of applicable objects within the component. Projects creating concrete factories could get the flexibility they needed using this interface. Common component code often included concrete factories that could be used as defaults by Project Component Developers developing unique application components for specific products. Common Component Developers also used these default factories for testing.

While this discussion focuses on creation, destruction of objects is also important within our C++ system. In many cases, the methods in the Abstract Factory create single objects, or single-rooted trees of related objects. Destruction can then follow the containment tree. In some cases, however, the object dependencies are more complex directed graphs that benefit from direct destruction of intermediary objects which may be project specific and therefore unknown to common façades. In these cases, component configurators were created which control all lifecycle functions and provide necessary object references to the common façade.

4.2.2 Distributed Configuration

The factory solutions described above address component-centric concerns about creation and initialization. Solutions involving the configuration and connections of the system as a whole also evolved. Originally, we leveraged our use of the Layers^{xiii} pattern and instantiated and configured the system layer by layer from the lowest layer (fewest dependencies) to the highest layer with the exception of the Configurator layer at the top, which was the manager of the configuration execution tree. In this way, creation and resolution of direct local dependencies between façades could be accomplished in one pass, specifically calling the façade constructor of each component to be instantiated and creating the necessary connections to components in the same or lower layers.

As our architecture evolved to fully address issues of distribution, the need for a much more flexible approach arose. We therefore eliminated the ordering dependencies for configuration by developing a Distributed Configuration pattern. In this solution, component facade creation and component supplier registration with Infrastructure services occur in one pass and consumer registration and consumer local and remote dependencies on other components are resolved in the second pass. This provided the assurance that a remote supplier component existed before attempting to resolve a dependency. As we began to use the Proxy^{xiv} pattern extensively to support clients of distributed components, the Distributed Configuration pattern allowed masters and proxies to be created in the first pass and connected in the second pass.

While this approach provided more flexible configuration, its primary motivation was performance. Remote communication is typically orders of magnitude slower than local communication. The multi-pass initialization sequence requires much less remote communication and synchronization between processors. In the ideal case, a single set of instantiated component information from the first pass can be exchanged just prior to the second pass. This greatly increases initialization performance.

This approach also provided other benefits. We gained greater flexibility by supporting limited bi-directional dependencies and relationships to higher layer component objects (through interfaces

defined in lower layers). In addition, this approach facilitated failure handling. The second pass is initiated via a `FinishInit()` function in the facade. The `FinishInit()` function, unlike the constructor used for the first pass, returns a status indicating the relative success of initialization, thus allowing fault recovery behavior to be inserted in the configuration process.

4.3 Behavioral Patterns

The behavior of reusable mission processing systems must resolve opposing goals of deterministic verifiable real-time operation and support for product specific configurability in support of reuse. The use of behavioral patterns within the system aimed to balance these forces, and were primarily used to support the architecture's control and data flow policies^{xv,xvi}.

4.3.1 Control Flow Related Patterns

In a real-time embedded system, all aspects of control flow are of prime importance. The overall execution flow policy, the scheduling paradigm, the use of threads, and the allocation of application components to threads all represent critical issues. Since the dominant cost in system development is application development, our architectural development approach, in general, followed a top-down methodology where application forces drove architecture and middleware development. So, while many of the execution control aspects are heavily supported and/or fully implemented by the middleware, the architectural approach was designed to optimize the application architecture.

One of the first considerations, therefore, was to define the overall control flow policy as seen by the application components. For many tactical reasons primarily associated with project reuse objectives, a non-threaded event-driven execution architecture was selected. This policy avoided the complexities and scalability issues associated with approaches such as Active Objects^{xvii}, and leveraged the concepts associated with Passive Objects^{xviii} to yield an architecture that was scalable to large numbers of application components, a requirement of our large-scale applications. The event-driven nature of the application is based on the Observer^{xix} pattern, as implemented by a Real-Time CORBA Event Service^{xx}. In our system, the mission processing subsystem is the primary

controller of the avionics network, and schedules messages between the mission processor and other subsystems (e.g., radar, GPS, radios) at periodic, typically static, rates. Within the application, the reception of new data results in events being generated to notify consumers that new data is available, resulting in a chain reaction of processing until all results associated with a particular rate is complete. At that point, the output messages for that rate are transmitted. Thus, the overall execution framework is an aggregate of timed input/output behaviors which stimulate an event-mediated internal architecture. The overall application component execution is typically cyclic, therefore, without being designed with unnecessarily restrictive cyclic processing assumptions. There are a number of different Component types associated with event delivery and execution control. Some components are event suppliers. Components which are only event suppliers are typically associated with the input subsystem. Some components are event consumers. Components which are only event consumers are often found within the View layer, associated with display updating. Consumer only components are frequently referred to as Terminal Consumers to emphasize that they are the leaves of event dependency and execution trees. Most components are both Event Suppliers and Consumers, and are referred to as Event Driven Components.

The scheduling of components is determined by the specification of quality of service attributes outside the components. In current systems, threads are created for each periodic Rate Group of processing and Event Consumers are statically allocated to them according to rate monotonic scheduling methods. With Washington University in St. Louis, and under Air Force Research Laboratory (AFRL), DARPA, and Open Systems Joint Task Force (OSJTF) support, several research programs and demonstrations have extended this approach with more dynamic support for soft real-time operation through multi-level adaptive resource management and multi-level maximum urgency first scheduling. Some of these programs have benefited from Real-Time Adaptive Resource Management technology from Honeywell Laboratories and Quality Object technology from BBN Technologies^{xxi}.

The use of multi-threaded execution architectures leads to the need for concurrency control. Components are designated as either Single Threaded or Multi-Threaded Components, based on the threads within which they are invoked in a particular configuration. While it remains an area of further work, some basic concurrency control patterns have been identified. Single Threaded Components use the No Locking pattern, where either no locks or Null Locks (a variant of Null Object^{xxii}) are used. Multi-Threaded Components are designed to be Internal Locking (i.e. where all public methods are internally guarded by locks, a simplified variant of the Monitor Object^{xxiii} pattern), External Locking (i.e. where clients must specifically invoke a lock interface to support simultaneous locking of multiple supplier components), or Synchronization Cache Proxies (i.e. where a Cache Proxy is created for each client thread and refreshed from a single master component. In this case, thread boundaries are crossed between the original server component and the server proxy so that locking is not required between clients and servers. This is in contrast to Synchronization Proxy^{xxiv}).

4.3.2 Data Flow Related Patterns

In contrast to the *push* style control flow where control emanates from suppliers to consumers, the data flow policy is predominantly *pull*. Clients read data from suppliers via `Get...()` method interfaces. This data flow policy results provides more adaptivity and configurability that was critical in providing support for single and multi-processor hardware architectures. This flow policy was an additional motivation for some earlier mentioned patterns, including Cache Proxy and Synchronization Cache Proxy.

5 Conclusions and Future Work

Patterns and pattern languages have been a critical element of our architecture development and dissemination process. The patterns briefly mentioned here represent only a subset of those used and developed on the program. Additional sets of patterns were created in the areas of error management and project specific tailoring of common components^{xxv}. To harvest lessons associated with the last several years of application

development, we are currently pursuing an effort of mining resulting applications to identify additional patterns and understand how designed and taught patterns were tailored or modified in practice.

Since the architecture is heavily rooted in patterns, architectural evolution and refinement often takes the form of modified patterns. For instance, the original Component pattern was heavily based on the concept of a single interface embodied in a façade. As we leverage concepts from newer component models such as the CORBA Component Model within DARPA Information Technology Office (ITO) sponsored programs, this is changing. Associated work in progress is expanding this concept to physically separating the multiple logical interfaces through the Extension Interface^{xxvi} pattern. These extensions and associated tool related research promise greatly improved modeling, analysis, and automated generation of highly configurable embedded middleware and applications. Our long-term goal in this area is to advance the state of the art and practice by being part of a community effort in developing a standard distributed real-time embedded system component model to form a critical mass for targeted research and development of technologies, tools, middleware and other products.

6 Acknowledgements

Much of our patterns work reflects an extremely fruitful collaboration with Dr. Douglas Schmidt, originally at Washington University in St. Louis and now on leave from UC Irvine serving as a DARPA ITO Program Manager, and the ACE and TAO communities therein. The definition of the concurrency patterns briefly mentioned herein was led by Ed Pla and Brian Mendel within the context of the project's software Core Architecture Team. Many of our emerging extensions are being developed as part of the AFRL, DARPA, and OSJTF sponsored Weapon System Open Architecture program, the AFRL sponsored Adaptive Software Technology Demonstration program, and several DARPA ITO embedded system programs.

ⁱ F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *Pattern-Oriented Software Architecture—A System of Patterns*, Wiley, 1996, p. 125-144.

ⁱⁱ Ibid., p. 140.

ⁱⁱⁱ Martin, Robert C., “Design Patterns for Dealing with Dual Inheritance Hierarchies in C++”, <http://www.objectmentor.com/publications/dih.pdf>.

^{iv} Johnson, Ralph, *How to Develop Frameworks*, Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 1997.

^v Sharp, David C., “Reducing Avionics Software Cost Through Component Based Product Line Development”, *Software Technology Conference Proceedings*, April, 1998.

^{vi} E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994, p. 139 – 150.

^{vii} Jakubik, Paul, “Callback Implementations in C++”, <http://www.primenet.com/~jakubik/callback.pdf>, 1997.

^{viii} C. Cleland, D.C. Schmidt, T. Harrison, *External Polymorphism: An Object Structural Pattern for Transparently Extending C++ Concrete Data Types*, Pattern Languages of Programming Conference, Allerton Park, Illinois, September 4-6, 1996.

^{ix} Buschmann, p. 268.

^x Gamma, p. 107 – 116.

^{xi} Ibid., p. 87-95.

^{xii} Ibid., p. 185-193.

^{xiii} Buschmann, p. 31-52.

^{xiv} Gamma, p. 207 – 217.

^{xv} Sharp, David C., “Avionics Product Line Software Architecture Flow Policies”, *Digital Avionics Systems Conference*, October 1999.

^{xvi} Doerr, Bryan S., and Sharp, David C., “Freeing Avionics Software Product Lines from Execution Dependencies”, *Software Technology Conference*, May 1999

^{xvii} D.C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture (Vol. 2): Patterns for Concurrent and Networked Objects*, Wiley, 2000, p. 369 – 398.

^{xviii} Pryce, Nat, “Passive Object”, <http://www.doc.ic.ac.uk/~np2/patterns/concurrency/passive-object.html>, 1997.

^{xix} Gamma, p. 293 – 303.

^{xx} 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.

^{xxi} Corman, David E., Gossett, Jeanna, “Weapons System Open Architecture - Using Emerging Open

System Architecture Standards to Enable Innovative Techniques for Time Critical Target Prosecution”, to be presented at *Digital Avionics Systems Conference*, October 2001.

^{xxii} Woolf, Bobby, “Null Object”, Pattern Languages of Program Design (PloP), <http://citeseer.nj.nec.com/woolf96null.html>, 1996.

^{xxiii} Schmidt, p. 399 – 422.

^{xxiv} Buschman, p. 270.

^{xxv} Sharp, David C., “Containing and Facilitating Change Via Object Oriented Tailoring Techniques”, *Software Technology Conference Proceedings*, May, 2000.

^{xxvi} Buschmann, p. 141 – 174.