

Reducing Maintenance Costs Through the Application of Modern Software Architecture Principles

Christine Hulse and
Scott Edgerton
United Defense, LP
Minneapolis, Minnesota
1.612.572.6109/6156
christine_hulse@udlp.com
scott_edgerton@udlp.com

Michael Ubnoske
Architecture Technology
Minneapolis, Minnesota
1.612.829.5864
mubnoske@atcorp.com

Louis Vazquez
Department of the Army
Picatinny Arsenal,
New Jersey
1.973.724.5259
lvazquez@pica.army.mil

1. ABSTRACT

Large software programs are usually long lived and continually evolve. Substantial maintenance effort is often extended by engineers trying to understand the software prior to making changes. To successfully evolve the software, a thorough understanding of the architect's intentions about software organization is required. Software maintenance costs can be reduced significantly if the software architecture is well defined, clearly documented, and creates an environment that promotes design consistency through the use of guidelines and design patterns. Building a maintainable system depends upon the consistent application of these architectural practices. This paper describes the application of modern software architecture methods to achieve a maintainable implementation of a large, distributed, real-time, embedded software system.

1.1 Keywords

Architecture, design patterns, software maintenance, modeling, real-time software

2. INTRODUCTION

The Crusader Field Artillery System is the U.S. Army's next generation 155-mm self-propelled howitzer (SPH) and its companion resupply vehicle (RSV). The Crusader is a software-intensive system that is being designed to last well into the next century. Ada95 is the implementation language for all software development on the Crusader program. Modern software architecture methods are being applied to capture the general principles used throughout the design and to provide a guide for further development of the software. The architecture is being modeled using the Unified Modeling Language (UML) to bring together the design vision of all of the key stakeholders. Our software architecture is a high level model of the software contained inside a Crusader self-propelled howitzer vehicle or a resupply vehicle.

The Crusader software has been designed with the following architectural objectives:

- Support for an object-oriented design
- Support for distributed objects
- Support for implicit invocation
- Support for the Joint Technical Architecture (Army)
- Resilience to change
- Support for meeting hard real-time requirements
- Support for domain reuse.

Past experience has shown that software maintenance accounts for about seventy percent of a weapon system's overall life cycle cost. Using modern software architecture methods can lead to substantial reduction in maintenance costs, improvements in software reuse, and an increase in the quality of the software. These techniques can increase the

asset value of software by making it easier to evolve to comply with new requirements and implement new capabilities. Software that is developed with a well-conceived architecture permits coarse grained software components to interact across interfaces without regard for internal implementation details. This level of abstraction and encapsulation permits the software to be upgraded and enhanced at a structural level.

This paper describes our experiences in developing an object-oriented software architecture for the Crusader system using modern software architecture methods and with system maintainability in mind.

3. ARCHITECTURE PROCESS

The overall approach that was used to define the Crusader software architecture was based on the software architecture methodology described in [4]. The software architecture was reflected by the different views of the Unified Modeling Language (UML) and was developed through several iterations. Our object-oriented (O-O) modeling approach is depicted in Figure 1.

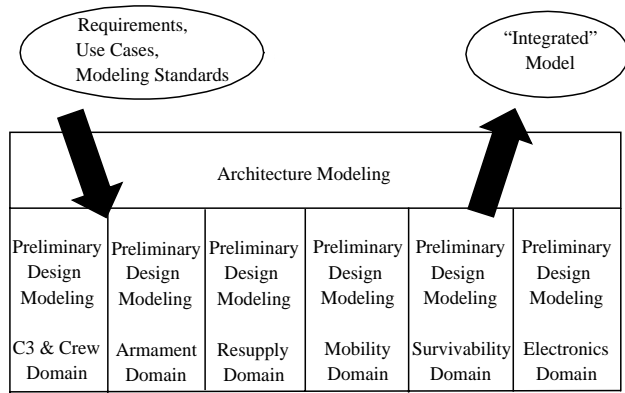


Figure 1. Iterative Development Method—Repeated Refinement of the Model Via Reorganization and Addition

An important role of our software architecture team was to provide guidelines, rules, and recommendations for Crusader software architecture design and O-O modeling of that architectural design. The guidelines we developed were short, heuristic, “rules of thumb” that embodied previous experience and cultural/organizational conventions about how to do high quality work. The overall goal of the guidelines was to develop a consistent, easily maintainable architectural model that supported requirements analysis, software design, and software development. This goal was achieved with our architecture through strict adherence to a minimal number of guidelines. It was very important to us that our O-O model be understandable and clear, not only to authors, but also to other users and future maintainers.

4. ARCHITECTURE STYLE

Our architecture style is embodied in the design patterns we have chosen to implement. The goal of the Crusader architecture patterns is to help software developers resolve common difficult problems encountered throughout the software. The architecture patterns express fundamental structural organization schemes for the Crusader software. They provide a set of software components, specify their responsibilities, and include rules and guidelines for organizing the relationships between them.

Our architectural patterns are high-level strategic patterns that concern large-scale components and the global properties and mechanisms of the Crusader software. They have wide-sweeping implications that affect the overall skeletal structure and organization of the software. Although many useful patterns have been defined in the literature, we found that only a small subset are applicable in the design of real-time, embedded systems. Selecting the right patterns was a difficult task. We began by characterizing the problem we were attempting to solve.

4.1 Problem Characterization

The system consists of hardware devices (e.g., turret, gun) that operate at widely differing speeds and are controlled by software. This facet of the system is driven by sophisticated command and control software to support the crew as they perform a defined mission. Due to its inherent complexity, mapping the Crusader functionality onto computing platforms is not straightforward and performance is an important consideration as this mapping is performed.

The problem domain logically defines both a vertical and horizontal structuring of the software. There is a vertical structure in that our software system has high level command and control operations that rely on low level operations. High level operations include things like: perform a fire mission, move vehicle from location ‘x’ to location ‘y,’ and resupply the vehicle. Examples of low level operations are: process sensor input, send output to an actuator, and control a motor. Within the software, communication flows from high level to low level and vice versa. In general, crew requests move from high level to low level and answers to requests move from low level to high level. There is a horizontal structuring of the software as well—several operations are at the same level of abstraction but are largely independent. For example, planning for vehicle movement is independent of the functionality to perform a fire mission.

The system is long lived and the software will continually evolve. As a consequence, there is a desire to make parts of the system interchangeable so that it is easy to replace selected components with alternate implementations. Finally, the software development work needs to be subdivided among several contractors, some of which had not been identified when the architecture effort commenced.

4.2 Software Layers

One of our fundamental design patterns was to arrange the software into logically related packages that are organized in a hierarchical, client-server fashion. The hierarchy is divided into a small number of layers, with the higher layers containing application-domain packages and the lower layers containing interfaces to the operating system and the hardware. Strict adherence to the layers is paramount since it helps us with the complex task of managing the inter-object dependencies of a large, object-oriented system; in our case estimated to require the development of over a million source lines of Ada95 code. Figure 2 depicts the five layers identified in the architecture.

More concretely, when we describe the architecture as layered, we mean:

- Components are allocated to groups called layers. A layer is defined by the type of information that it encapsulates
- The layers are ordered, with the hardware-dependent layers in the lower layers and application-specific layers in the upper layers

- Interactions between layers are restricted. A component can only access other components in its own or lower layers.

Layers should be viewed primarily as pre-runtime entities. That is, assignment of a component to a layer primarily describes the knowledge, or lack of knowledge, that is allowed to be reflected in the coding of that component.

The layers are defined as follows (high to low):

Layer 5 – This layer contains the classes needed to support a dialog with the crew.

Layer 4 – This layer contains the capabilities that perform the core Crusader applications and the capabilities required to support embedded training and vehicle management.

Layer 3 – This layer contains the key abstractions that represent capabilities found on an artillery vehicle. This is where the major domain abstractions associated with software controlled equipment (e.g., load arm, conveyer) are located.

Layer 2 – This layer defines the distributed infrastructure and contains mechanisms for communication, data persistency, error handling, configuration support, graphical

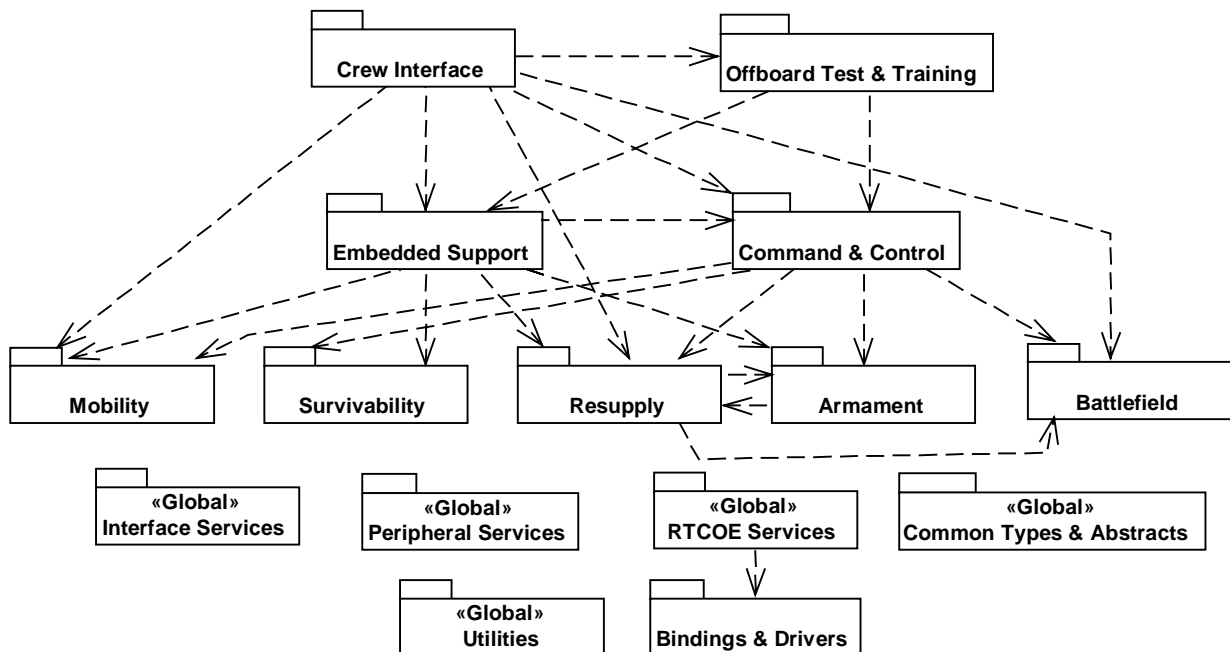


Figure 2. Main Class Diagram Showing the Architecture Layers

libraries, and general services to support low level interfaces to the vehicle.

Layer 1 – This layer contains all of the computer and operating system specificity. It also contains general reusable utilities.

This pattern helps support portability of the software, particularly at the lower layers since they provide an abstraction buffer between the application software and the underlying operating system and hardware. This approach directly supports the Army’s architectural goal of creating a real-time common operating environment (RTCOE) that can be reused on similar Army systems. The RTCOE is a crucial part of the lowest two layers of the architecture. It is envisioned to serve as a foundation for all of the Army’s future weapon systems and is being designed to support reuse. The RTCOE provides a level of abstraction to applications through a set of application programming interfaces (APIs). These APIs allow applications to be developed without concern for the underlying operating system and the actual physical distribution of the software among multiple computers. The RTCOE creates a virtual machine that is able to execute a number of Ada programs in parallel, as if each program had its own private processor. Components in the higher layers are designed to operate as a set of cooperating Ada programs.

The RTCOE components provide a multi-processing environment, distributed services, system reconfiguration, utilities, and common data structures for application developers. From an application perspective, the RTCOE components offer consistent services, provide access to basic services, provide platform independence, and offer system-wide stable performance.

RTCOE services can be viewed as an extension to the Ada95 runtime in that the services become part of an application’s executable image. Figure 3 depicts the relative structure between the various software components and the RTCOE.

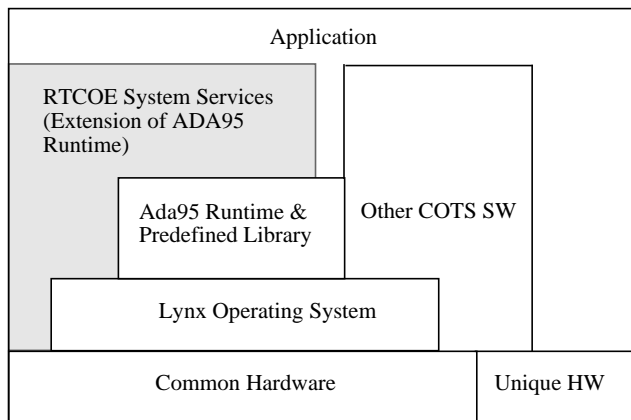


Figure 3. Real-Time Common Operating Environment

The software layers form large-grained abstractions with the goal being to minimize layer interactions so that changes in one layer affect at most two other layers: the layer above it and the layer below it. Given the real-time requirements for our system, this was sometimes hard to achieve due to performance considerations.

Acknowledging this, we adopted a relaxed layer approach that did not strictly enforce communication between adjacent layers; that is, an object was allowed to access an object in any lower layer if this was necessary to meet performance requirements. This was done to avoid an implementation where the execution path is forced to pass through several layers in order to invoke the required service, when it is more efficient to call the service directly.

4.3 User Interface

From a structural perspective, the architecture is an instance of a Model-View-Controller (MVC) design. The Crusader system maintains a model of the world that is controlled via updates in response to new data sources, and is viewed by displays. This pattern quite naturally fits the problem domain. Vehicle properties rely upon a number of different sensors, controls, guidance rules, and artillery policies. These properties are used to generate outputs in several ways. Clear separation of concerns via an MVC implementation greatly simplifies system evolution and maintenance.

A number of objects in our system have orthogonal components that accept and respond to user generated events, maintain application data, and display data to the user. If these components are combined into a single monolithic object, they are artificially tightly coupled. This results in more work to maintain these objects and limits the reusability of the components. A very common set of orthogonal components of objects is:

- Model – the data component of the object
- View – how the data values are displayed
- Controller – receives and processes events from the view.

By separating these concerns, a small set of collaborating objects can work together to realize an application’s behavior. Each of these objects is concerned with a different subject matter as it applies to an application. This pattern is applicable when application data must be controlled and displayed. The components of the Model-View-Controller pattern are:

- Model – The application packages on layers three and four
- View – The Crew Interface package on layer five
- Controller – The Interface Services package on layer two.

In our architecture, this pattern is implemented with a publish-subscribe and proxy mechanism.

4.4 Communication

A number of the client/server interactions in our design are implemented with a direct call interface (i.e., a procedure or function call) between the client and server. However, when the server object resides in another thread or process a direct call interface is not an option. We were faced with a situation where some of the object relationships defined in our design model spanned threads and processes. Further, we knew that process and thread boundaries would change as the design evolved. To minimize the impact of these changes, we decided that a client object should not have to be aware that a server object is on another thread or in another process. The details of messaging would be abstracted from the client object.

The Proxy pattern was introduced to facilitate communication between objects in different threads and processes. In this pattern, when a client object requires a service from a server object in another thread or process, it invokes the service on its local proxy. The proxy then marshals a request to the original server object. Since our system is distributed, this pattern is useful because it facilitates changes in the hardware and software architecture and its degree of distribution. As well, it abstracts the details of messaging from the applications, thereby insulating them from changing implementation.

This pattern provides several substantial benefits for our architecture. It encapsulates messaging by providing a pattern for inter-object communication. This encapsulation allows the common messaging software to be written once and “inherited” by all software developers. We created an abstract proxy class that developers would inherit from to create an instance of a proxy class (refer to Figure 4).

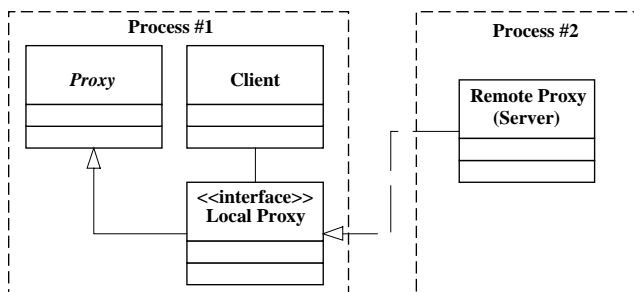


Figure 4. Proxy Class Diagram

This abstract class and the associated inheritance was captured in our O-O model and code generation scripts were created to automatically generate the code associated with marshalling of parameters and inter-object messaging. Use of these automatic code generation capabilities resulted in substantial savings to the program and helped ensure that our

object model maintained consistency with the coded implementation.

In addition to proxy interfaces, we found that our system had numerous examples where changes of one object required dependent objects to change accordingly. If an object needs to explicitly inform every dependent object about its state changes, the object interfaces and implementations become intertwined. This greatly hampers system evolution and maintenance. In Crusader, it is common that a single source of information acts as a server for multiple clients who must be updated autonomously when the data value changes. This is often the case with real-time data acquired through sensors. We wanted to design an efficient means for all clients to be notified of a server’s state change. A desire to solve this problem is what led to our Publish-Subscribe pattern (also referred to as the Observer pattern).

In this pattern, a single object, called the publisher, provides the data automatically to its clients, called subscribers. In our O-O model, an abstract Publisher class can be subclassed to add the specialized behavior to deal with the specific information being published. When the publisher updates its data, the subscribers are automatically notified. Upon notification, the subscribers get the updated data from the publisher. To address efficiency concerns, we overlapped this pattern with the Proxy pattern. A proxy of the publisher is created when the subscriber is in a different process. The publisher sends updates to its proxies and subscribers in different processes are notified via the publisher’s local proxy. Subscribers then get the updated data from their local proxy.

Similar to what was done for the Proxy pattern, we created an abstract Publisher class in our O-O model that was subclassed by the developers when they wished to create an instance of a publisher. Code generation scripts were created to automatically generate the code specifications and skeleton bodies. The required messaging is inherited from the Publisher abstract class. The Proxy and Publish-Subscribe communication patterns became the way all application level messaging was performed in our system.

4.5 Managing States and Modes

From an architecture perspective, the most important consideration we faced in designing the supervisory structure for the Crusader system was the degree of centralization needed. In a fully centralized approach, a single supervisor would assess the status of every component and manage state as appropriate. This leads to simple control mechanics but complex logic in the supervisor component. As well, this approach is likely to have bottlenecks that cause unacceptable performance. In a fully decentralized approach, each component would assess and manage its own control state, as well as that of components it uses in the course of its primary operations. This avoids the need for centralized supervision but requires complex

synchronization operations in nearly every component in the system.

We selected a hierarchical approach, as depicted in Figure 5, that finds the middle ground between the centralized and decentralized approaches. With our approach, each component manages strictly local concerns. Intermediate supervisory components manage groups of components, but are in turn managed by more centralized supervisors.

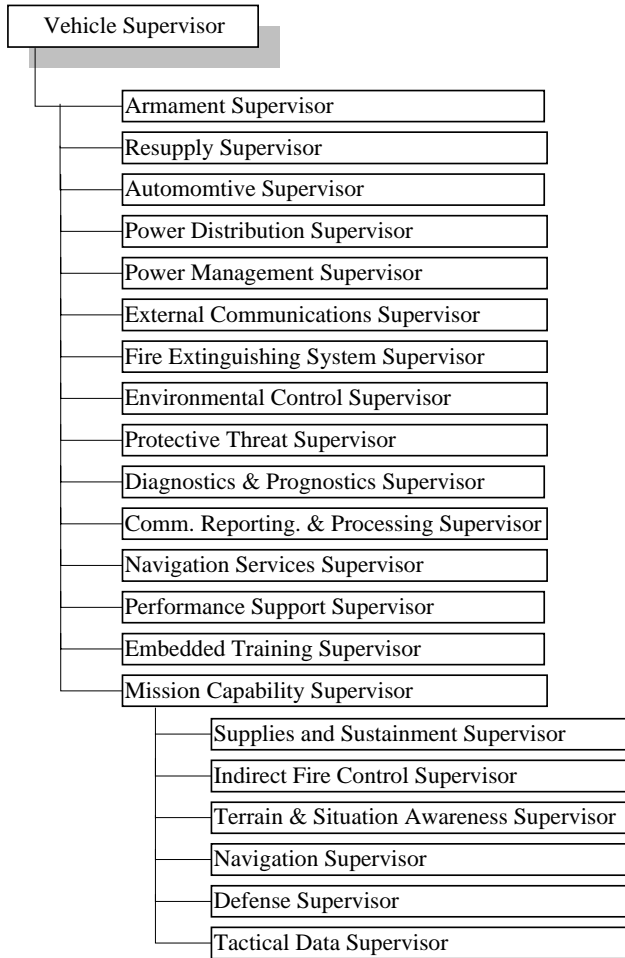


Figure 5. Supervisor Hierarchy

We have identified Supervisor objects that interact with and manage states of components under their domain. Supervisors are special kinds of state machines. Their states reflect in-the-small models of selected aspects of the system, and their transitions result in actions that change the state of components under their domain.

This supervisory approach formed the basis of our Monitor and Control pattern. It was designed to address the problem of how to manage groups of cooperating software components safely and efficiently. This pattern provides a framework for the monitor and control capabilities within the Crusader software architecture. The Monitor and

Control pattern manages the overall operation of the Crusader vehicle by: collecting inputs from other components to monitor their functional state; logically constructing, initializing, enabling, resetting, and/or disabling Crusader functional components and electronic devices; and, maintaining and displaying status and accepting new control instructions.

Capability objects are defined to maintain an understanding of a domain's operational capabilities; i.e., the necessary abilities and qualities to perform a requested action. Crusader vehicle capabilities are determined using information like equipment status, software fault status, and hardware fault status, all taken in context. Vehicle capabilities provide vehicle management software with the ability to determine which states and operations can and cannot be supported.

In our design, software capabilities are mapped to Crusader vehicle states. For a given state, the software can only perform capabilities that are available in the state. The Monitor and Control pattern ensures consistent management of a large state machine. State objects are defined to manage states within a specified functional domain. These objects are responsible for knowing what capabilities are required to support each state. They work with their associated supervisor object to verify that a requested state transition can be supported. Order objects are defined to enforce the state-capability mapping for each functional domain. An Order object is responsible for enabling and disabling a domain's capabilities based on state. This ensures that a functional domain can only service requests for capabilities that are enabled.

As well, our Order objects are used to implement a Facade pattern in each of the supervisory domains. This makes the task of accessing a large number of classes much simpler by providing an additional interface layer. This helps avoid excess coupling between the various supervisory domains. Using this pattern helps to simplify much of the interfacing that makes large amounts of coupling complex to use and difficult to understand. In Crusader, this is accomplished by having Order classes that are responsible for accessing a collection of classes within a domain. This pattern makes the interfacing between many classes more manageable and simplifies program maintenance. Figure 6 is a class diagram showing how one of our domains (armament) participates in the monitor and control pattern.

Safety was of primary concern as we designed these patterns. The Monitor and Control pattern provides a centralized coordinator for safety monitoring and control of system recovery from faults. Using the hierarchy of supervisor classes in the design, each supervisor is responsible for ensuring that their functional domain is operating safely, and the vehicle-level supervisor is responsible for ensuring that the vehicle is operating safely.

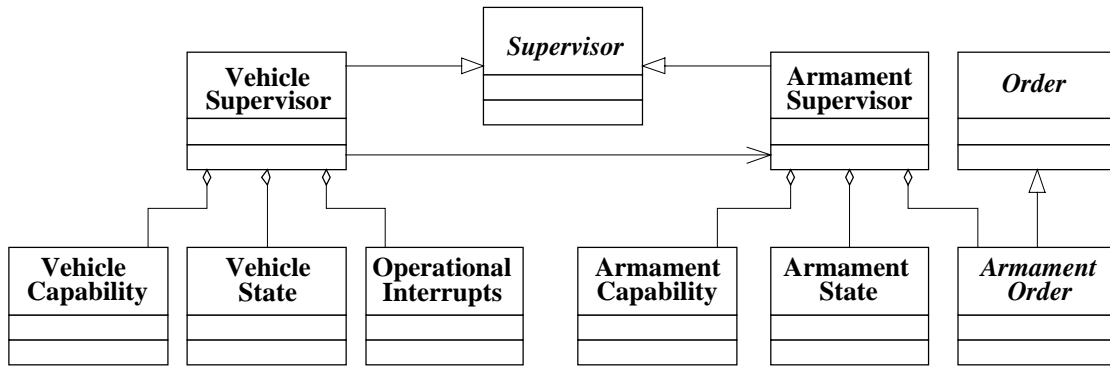


Figure 6. Example Monitor and Control Class Diagram

Through fault services provided in the RTCOE component, faults are continuously monitored and, when notified of a fault occurrence, the supervisor, capability, and state objects work together to ensure that each functional domain is configured to a safe state and that the overall vehicle is in a safe and consistent state.

5. COMMON STRUCTURE

Common patterns and mechanisms are an important element in obtaining design consistency, and ultimately play a major role in achieving a maintainable system. To support these common patterns and mechanisms, Crusader has provided system-level repositories in our O-O model and system level guidance, documented as architecture notes, to the development teams. Both the repository and guidance's have been effective communication tools. The following paragraphs describe some of the key repositories and guidance's that have been provided.

5.1 Common Abstracts and Types

In our O-O model, the Common Abstracts and Types subsystem represents the system-level repository for abstract reusable code (refer to Figure 7). Most of the aforementioned patterns have abstract classes in this subsystem—these abstract classes contain all of the common attributes and operations related to their associated pattern. Application level classes inherit from these abstracts, ensuring a consistent application of the pattern across the system. Because we are reusing the code of the abstract classes, we have found that our O-O development tends to go faster than conventional programming once a library of commonly reusable classes has been accumulated.

Common non-abstract types that are shared by multiple development teams or between several subsystems of a single development team that has no common import subsystem are placed into a global subsystem. These

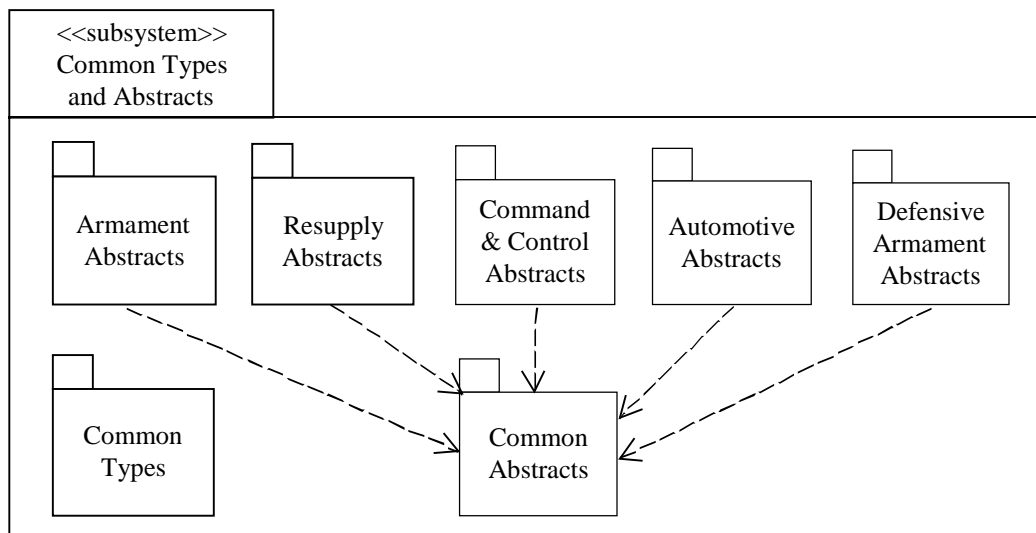


Figure 7. Common Abstracts and Types

common types are then additionally partitioned according to the areas which they support, such as munitions or units of measure.

Having the maximum amount of localized type declarations greatly reduces the recompilation time during development, test, and integration. It also reduces the number of subsystems affected by software changes later in the life cycle. Each time a type needs to change, only the dependent packages will be affected.

5.2 System Management

Our software architecture is divided into a series of application programs running on multiple processors connected by a local area network. Each application is responsible for handling some subset of the system events. Within each application are multiple threads (Ada95 tasks), allowing each application to prioritize different types of work and to support different levels of concurrency. These separate Ada tasks are collected together to form Ada programs.

The System Management subsystem provides services to release and monitor all application processes and threads. This subsystem contains Network Manager, Node Manager and Process Manager classes. The Network Manager is responsible for a collection of computing nodes on the Crusader LAN. The Node Manager is responsible for the release and monitoring of processes within a single network node. Lastly, each process has a Process Manager that releases and monitors all of its threads. When Crusader transitions from one major Vehicle State to another, some processes need to be terminated and others started. This network topology must also allow for reallocation of processes to processors during runtime. Figure 8 depicts the UML packages used in our system management design.

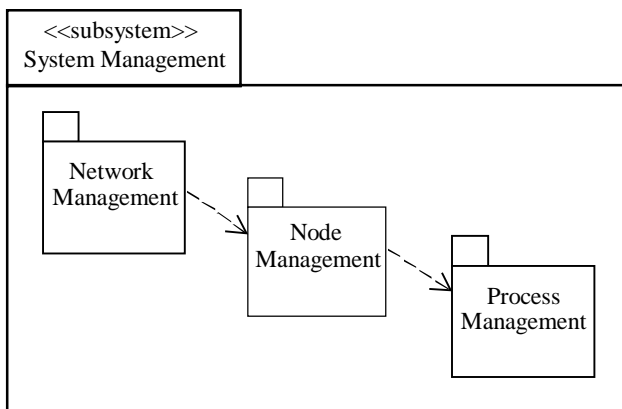


Figure 8. System Management UML Packages

This design was a special challenge in that we needed to establish a consistent implementation that resolved three issues. First, each process and task must be released and initialized in a specific sequence such that all dependencies

are resolved. Secondly all processes and tasks need a centralized reporting mechanism that allows for both monitoring and reporting of anomalies. Lastly, this design needs to allow for a dynamically configurable network topology. When Crusader changes major vehicle state, some processes need to be terminated and others started.

Our solution was to capture the common implementation features in a 'common' task structure. This uniform task structure is common to many event-driven systems and conceptually has the following structure:

```

loop
  wait for event
  process event
end loop
  
```

There are two types of events in our system: periodic (such as the expiration of a timer) and non-periodic (such as the arrival of a message via Proxy/Publisher service or other services). Each of these types utilize the same common template.

This common task template contains a supplemental form that developers use to indicate all dependency rules. These dependency rules are used to establish the overall initialization sequence. All 'common' application tasks are statically declared at elaboration time and block and wait prior to initialization for release by the process management service. Each of these tasks are released for initialization according to all dependency rules. Upon release for initialization, each task performs all necessary initialization activities and reports status back to process management. This provides the ability to report task and process health and also allows process management to build a list of active tasks for monitoring purposes. All tasks that are in an 'active' state are periodically monitored to detect if they have transitioned to an abnormal, completed, or terminated state.

The Node Manager is responsible for spawning the correct processes according to the selected mode of operation. Process status is determined from the aggregate task status of the tasks within the process. The Node Manager periodically monitors each process to detect if it has transitioned to an abnormal, completed, or terminated state. Likewise, a node status is reported to the Network Manager. Network status is determined from the aggregate node status of the nodes within the network.

5.3 Exception Handling

The requirements for Crusader call for a high availability, fault resilient system to be built using Ada95. The Crusader software programs must be able to respond in a reasonable way to unexpected situations. They must ensure a

continuous service despite unexpected events, providing a response that includes:

- Reaching a well-defined state
- Reporting the problem
- Taking a recovery action
- Continuing execution, perhaps in a degraded mode.

Ada95 offers an *exception* mechanism to signal and handle some categories of errors at run-time. Several language defined exceptions are raised when constraint violations on types, program logic errors, or storage exhaustion occur. However, in large systems that use exceptions extensively, the code to handle exceptions sometimes grows to the point of dwarfing the “normal” code, thereby reducing the legibility and maintainability of the code. The problem is fundamentally one of scale: large projects require a well-defined approach to exception handling so that exceptions are all reported in the same consistent manner. Figure 9 shows some of the key exception handling guidelines developed for the Crusader program.

In our design, the standard exception block is included as part of the aforementioned tasking template. The template contains two placeholders, one for recoverable errors and the other for fatal errors. When an exception occurs, specific context information is saved using Ada95’s exception occurrence facility. Ada95 provides this service to capture any information the compiler has associated with the occurrence of an exception. Although the type of information saved may vary from compiler to compiler, the implementation of this service is consistent such that a pointer to the information that is saved is always returned. This record of occurrence contains contextual information such as the place in the program where the exception was raised and the reason that it was raised.

The common manner in which Crusader utilizes exceptions lends itself to maintainability. These exceptions are routinely found on task bodies and therefore are all encompassing of their respective thread. During maintenance, most new or modified code will fall under the umbrella of an exception block.

5.4 Fault Services

Our approach for handling faults is both distributed and centralized. Although this may sound contradictory, we determined that the most maintainable design would capture all of the commonality using a centralized approach and at the same time keep all domain specific knowledge at the domain level.

Our Real-Time Common Operating Environment provides a centralized fault service that allows clients to create fault objects. These objects contain the basic attributes of the fault including the status of the fault. Any interested

Key Exception Handling Guidelines

- Exceptions should only be used to indicate that something undesired or abnormal occurred. As long as normal conditions of operation exist, the code does not take advantage of exceptions as a programming technique. A Crusader program will not raise an exception to indicate normal successful operation.
- Most exceptions that are raised during normal execution are system defects that must be reported to the development organization and corrected. Some errors may be only configuration errors that do not require code modification; e.g., size of a table, duration of a time-out.
- If at all possible, an error must be reported only once (“latched”), as close as possible to the point of discovery, and with all information useful to understand the error condition and its cause. To remain effective, the error reporting mechanism must not be flooded by numerous reports all pertaining to the same error.
- If a subprogram is likely to receive ill-formed or inconsistent input, then a status out parameter will be used to signal this to the invoking code in lieu of an exception being raised. This applies in particular to any code that receives data from outside of the Ada95 program.
- Exceptions can be used to signal an incorrect sequence of calls when this sequence is not supposed to happen in a normal functional program.
- A Crusader program will only explicitly raise Crusader defined exceptions. It will not raise any of the predefined exceptions declared in `Ada.Exceptions`. These exceptions should only be raised via the built-in checking mechanisms of Ada95. For example, a Crusader program should not have the Ada95 statement: `"raise Ada.Exceptions.Constraint_Error."`

Figure 9. Exception Handling Guidelines

application can subscribe to a fault object and be notified by an event each time the fault status changes.

All faults must be reported and logged in a priority fashion with our Diagnostics and Prognostics (DAP) component.

This centralized service is kept cognizant of all application level faults through a dynamic registry service. All applications create and destroy fault objects during runtime

through this registry service. This allows DAP to keep an accurate list of all application level faults in the system. Upon completion of initialization, DAP uses this list of faults to subscribe for notification of any fault status changes. The dynamic registry service eliminates the need to maintain a hard-coded list of all system faults that would require recompilation of the system each time a fault is added or removed.

All tasking anomalies are reported to fault services through our previously described Process Manager classes. Process management is the focal point in our system for reporting all software faults. This ensures that all software errors are reported and logged in a consistent manner.

The remainder of Crusader's fault handling mechanism is kept decoupled from the domain level applications. That is, the application that sets and clears a fault has no knowledge of who is notified of its fault status change. This keeps domain specific knowledge of the faults and their operational impact at the domain level. Providing system-wide notification through the use of events ensures that any interested process across our distributed architecture is notified in a timely manner. Our fault processing approach ensures that any software changes will only involve areas that have a specific interest in the fault from a domain perspective. This keeps the software maintenance impact to a minimum.

6. CONCLUSION

As stated in [1], it is the mark of a well-engineered object-oriented system that making post development changes does not rend the existing architecture, but rather, reuses and then augments its existing mechanisms. It has been our experience that unintegrated (stovepipe) software-intensive systems do not provide the necessary architectural infrastructure to support long term system maintenance. As a consequence, their useful life is shorter and they are not cost effective to maintain. On the other hand, systems developed using modern software architecture methods and object technology have attributes that address complexity, improve maintainability, promote reuse, and reduce software life-cycle costs.

The following points summarize the major system maintainability observations we have made as we designed and implemented our architecture:

- Modern software architecture methods and the use of object technology have helped us ensure that the software is designed with maintainability in mind. Considerable leverage can be obtained through reuse at the architectural coarse-grained level of system subsystems. Because of encapsulation and information hiding, object technology increases maintainability at all

levels—from individual objects to high level architectural components (subsystems).

- Documenting the software architecture in a rigorous way has many advantages for maintenance. Software maintainers must possess a good understanding of the existing code and a thorough understanding of the architect's intentions about software organization. Substantial maintenance effort is expended trying to understand the software in preparation for making changes. This maintenance effort can be reduced significantly if the software architecture is documented clearly and explicitly.
- Our use of design patterns helps maintainers become familiar with the overall system design through an understanding of the major control constructs, common mechanisms, and communication paradigms. As well, these design patterns provide a common vocabulary for maintainers as they evolve the system.
- Development of automatic code generation capabilities from our O-O model helps to ensure that our object model maintains consistency with the coded implementation. Having an up-to-date O-O model is a tremendous benefit for downstream system maintainers.

The importance of having a strong software architecture cannot be overemphasized when developing large, complex systems. For Crusader, we believe the software architecture has laid the groundwork for significant life-cycle cost savings.

7. ACKNOWLEDGEMENTS

The authors wish to thank the other members of the software architecture team for their significant contributions: Devesh Bhatt, Zack Hashemi, Gordy Keller, Matt Kulek, Robert Rolfe, and Roy Threadgill.

8. REFERENCES

- [1] Booch, G. Object-Oriented Analysis and Design with Applications, 2nd ed. Benjamin/Cummings, Redwood City, Calif., 1994.
- [2] Douglas, B. Real-Time UML: Developing Efficient Objects for Embedded Systems. Addison-Wesley, Reading, Mass., 1998.
- [3] Gamma, E., Helm, R., Johnson, R., and Vissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Mass., 1995.
- [4] Krutchen, Philippe, Architectural Blueprints—The '4+1' View Model of Software Architecture, IEEE Software, November, 1995.
- [5] Unified Modeling Language Specification. Object Management Group, Framingham, Mass., 1998