Reengineering A Legacy System Using Design Patterns and Ada-95 Object-Oriented Features

Shan Barkataki
California State University &
Litton Data Systems
Northridge, CA 91330-8182
818-597-5633
shan@csun.edu

Stu Harte
Litton Data Systems
29851 Agoura Rd.
Agoura, CA 91376-6008
818-597-5509
sharte@vines.littondsd.com

Tong Dinh
Litton Data Systems
29851 Agoura Rd.
Agoura, CA 91376-6008
818-597-5104
tdinh@vines.littondsd.com

1. ABSTRACT

This paper describes experience with using several design patterns for extensibility and reuse. Also described are techniques for implementing classes in Ada-95, using the tagged types and the child unit facilities.

1.1 Keywords

Design patterns, Ada-95, reuse, reengineering

2. INTRODUCTION

The context of this report is a project aimed at reengineering a large and high-entropy air defense system into a modern software architecture for achieving improved performance and better maintainability. A secondary objective of the project was to discover and create a set of reusable software components for use in future air defense projects[1,2]. This paper describes our experience in accomplishing this task. In implementing these components we used several design patterns [3] and took full advantage of the object-oriented features of Ada-95 [4]. All diagrams in this paper are in the Unified Modeling Language (UML) [5] notation.

3. THE SOFTWARE ARCHITECTURE

The top-level software architecture consists of a number of segments. Each segment represents a large concept in the application domain and is designed for high-reusability. In the air defense domain, examples of such large concepts are: Target, Weapon, Air tracks, and Communication links. Each segment encapsulates a number of core classes. A core class represents a smaller but cohesive application domain concept.

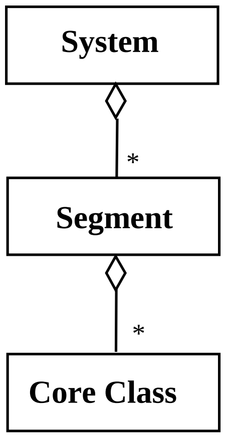

Figure 1.0 System Architecture

4. CORE CLASS PACKAGE

A core class is a UML package[1], implemented as an aggregation of several domain and implementation specific classes. The architecture of the UML package is illustrated in Figure 2. Central to this aggregation is an Ada package containing a private tagged type declaration representing the root class. Other parts in the aggregation consist of Ada packages representing helper classes, utility packages and primitive types packages.

Subclasses are implemented as Ada *child packages* of the parent Ada package. By using the Ada child package facility the core classes can be extended beyond what is possible with inheritance alone. For example, it allows the child classes to share exception identifiers and non-primitive subprograms (e.g. those used for handling exceptions,) that are declared within the Ada package for the root core-class. Such subprograms, are not public, rather they are meant to be shared only among the subclasses within the inheritance hierarchy. Therefore, we place all such declarations in the private part of the Ada package imple-

[1] We will use the terms *Ada package* and *UML package* to distinguish the two different concepts.

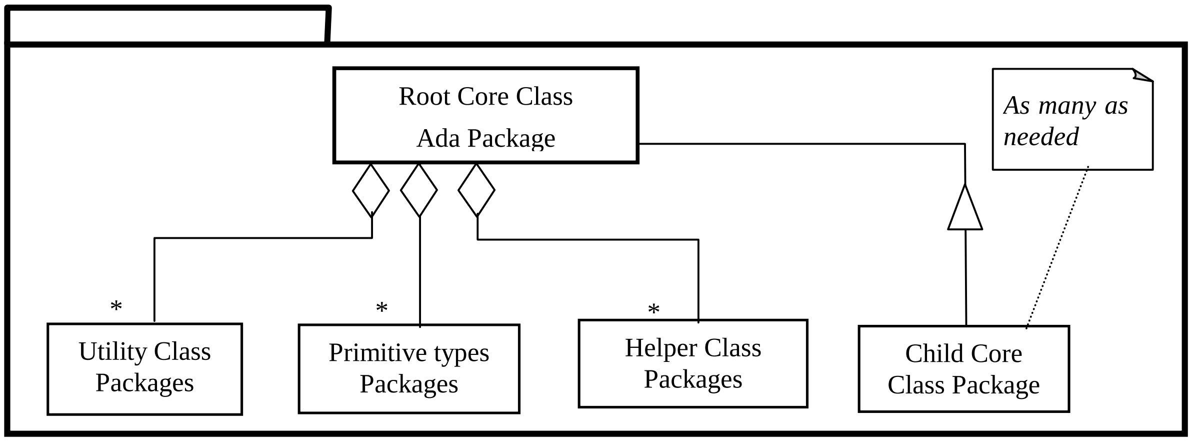

Figure 2: Core Class Architecture as a UML Package

menting the Root Class. Another advantage of this scheme is that we can write a set of primitive operations for each subclass, and enable the client to invoke any of these operations polymorphically. To do this, the client Ada package has to *with* the appropriate Ada child packages and make the root class Ada package directly visible by applying a *use clause.* It should be mentioned that we did not use this feature, favoring instead, the façade-controller design pattern with centralized dispatching (see Core class Façade-controller below.)

5. USE OF DESIGN PATTERNS

We used two fundamental design patterns in the reengineered software.

(a) Façade-controllers in the design of both segments and core classes.

(b) Brokers in each core class.

The primary motivation for using these design patterns was to increase reusability of the software components through elimination of unnecessary coupling.

Segment façade-controllers act as use case controllers. They receive inter-segment service requests and execute each request by invoking a sequence of operations (event trace) provided by their core classes (see Figure 3a).

A core class façade-controller acts as the API for the core class assembly and provides a flat interface for the services provided by the root class and all its subclasses. Thus, the façade-controller hides the class's internal organizational complexities from its clients.

When a client makes its first contact with a server, the client is given the identity of a class instance (server-id) that is set up to handle the current and all future service requests. The server-id is implemented as a class wide access variable pointing to the server class instance. In all subsequent interactions, the client passes this server-id as one of the parameters of the service request. When the façade-controller receives a service request, it uses the server-id to switch the request to the appropriate subclass. We use the Ada-95 dispatching mechanism (polymorphism) for implementing the switching operation. The basic concept is illustrated in Figure 3.b. The Ada package implementing the root class contains a class wide operation (*dispatcher*) for each dispatchable primitive subprogram. Whenever the façade-controller receives a dispatchable service request, it forwards the request to the appropriate message dispatcher along with the server-id. The dispatcher gathers any necessary data and then dispatches the request to the appropriate subclass. The dispatchers are designed to operate at class level, and use only class wide information. This makes the dispatchers completely extensible, i.e., addition of new subclasses does not affect the dispatcher's logic. We also found the dispatching mechanism very useful for handling cases of object mutation. For example, we may have a situation where an object reported by a radar is initially classified as an *unknown flying object,* and later the object mutates to a *ballistic missile*. In this case, if a client asks for the predicted location of the flying object, then depending upon the current classification of the flying object (unknown or ballistic), the dispatcher will invoke the *Location Predictor Subprogram* in the appropriate subclass.

We place the dispatchers in the private part of the Ada packages and implement the façade-controller as a child unit of the Ada package for the root class. This scheme enforces our design abstraction by ensuring that clients do not have visibility to the dispatchers; all client service requests have to pass through the façade-controller.

Façade-controller benefits: This design pattern provides two major benefits. The façade part provides a stable Application Program Interface (API) for the component it represents. The controller part separates the system specific control behavior from the domain specific behavior. The domain specific behavior is relatively stable, whereas system specific behavior changes with different systems. Therefore when a segment, or a core class is reused in another system, we expect most of the changes to be in the controller, leaving the underlying domain specific classes relatively unchanged.

Broker: We used the broker design pattern for implementing persistent associations. The clients register with their servers' brokers to receive automatic change notification. The core classes notify their respective brokers whenever a significant event occurs. Each broker broadcasts the event notice to all clients registered to receive notification for that particular event. The brokers isolate the core classes from the impact of changes in their clients. Brokers also provide the core class designers with a uniform method of notifying the clients whenever a significant event occurs.

A broker's view is limited to the capabilities provided by its associated class. Whenever a core class is extended with a new subclass, we also create a new broker subclass that is compatible with the capabilities of the newly created core subclass. Thus, the broker classes are organized in an inheritance hierarchy identical in structure to their associated classes (see Figure 4).

6. CONCLUSIONS

We have successfully implemented and delivered several segments containing over 40 classes, thereby proving the viability of the design patterns described in this report. Improvements in the maintainability by using the façade-controller have been validated by the ease of making changes during the testing and integration phase. The broker design pattern has been effective in decoupling the core classes from the volatility of their clients. One major benefit has been the uniformity in the design in this large and varied project. The use of uniform architectural concepts helps engineers to comprehend the design (and the code) quickly, without getting bogged down by varied and complicated control logic. We are also pleased with the object-oriented programming features in Ada-95 that have enabled us to implement the object-oriented design concepts directly into code. As a result, there is good traceability between design and code.

7. REFERENCES

[1] Barkataki, S.; Harte, S.; Dousette, P.; Johnson, G. Strategies for Developing Reusable Components in Ada-95; Proc. ACM Symposium on Applied Computing; Atlanta GA; Feb 98.

[2] Barkataki, S.; Dousette, P.; Reengineering for High Reusability- A Process and a Method; Proc. 10th. Software Technology Conference, Saltlake City, Apr 1998.

[3] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. Design Patterns, Addison-Wesley, Reading, MA, 1995

[4] ANSI/ISO/IEC-8652:1995 Ada-95 Reference Manual.

[5] Booch G; Jacobson, I.; Rumbaugh, J,. The UML Specification Documents. Rational Software Corp.; Santa Clara, CA. www.rational.com 1997

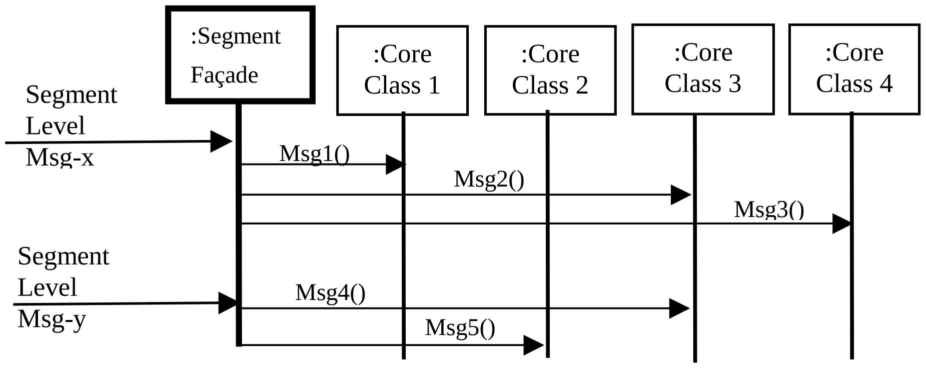

Figure 3a Sequence Diagram Showing the Role of the Segment façade

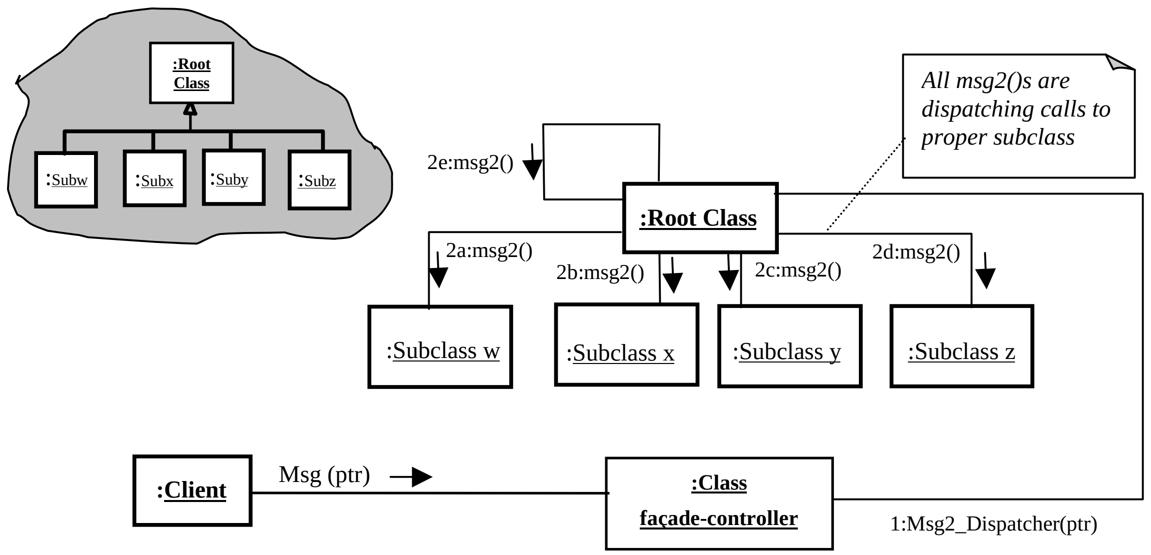

Figure 3b. Collaboration Diagram showing the role of the Core class façade-controller

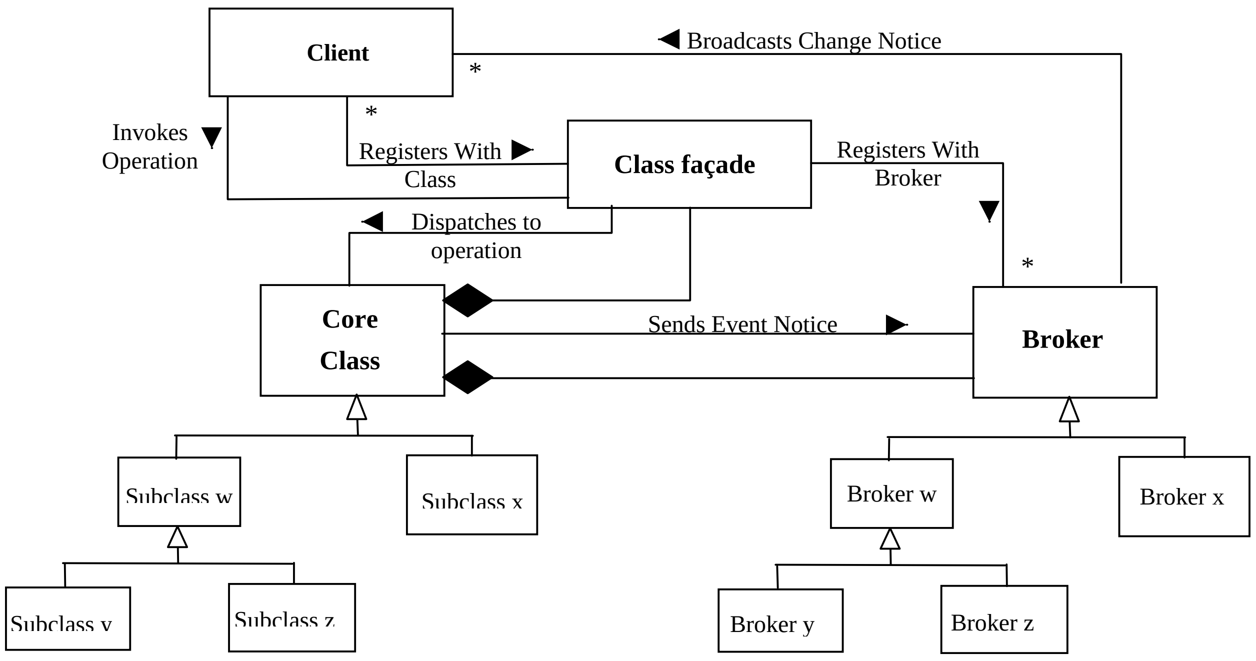

Figure 4. Core class architecture with broker