DOMAIN DRIVEN SIMULATION MODELING FOR SOFTWARE DESIGN

by

Andrew Evan Ferayorni

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

December 2007

DOMAIN DRIVEN SIMULATION MODELING FOR SOFTWARE DESIGN

by

Andrew Evan Ferayorni

has been approved

November 2007

Graduate Supervisory Committee:

Hessam Sarjoughian, Chair
Paul Scowen
Joseph Urban

ACCEPTED BY THE GRADUATE COLLEGE

ABSTRACT

Complex software system designs are often plagued by defects that stem from misunderstood requirements and poor design decisions. The ability to execute and test high level design elements can help with decision making and defect detection early in the design process. Simulation is a tool that can be used to model high level software design elements and execute them for analysis purposes. More specifically, domain aware simulation environments can provide these benefits while reducing the time spent developing simulation models. System-theoretic modeling and simulation frameworks such as Object-Oriented Discrete-event System Specification (OO-DEVS) are commonly used for simulating complex systems, but they do not account for domain knowledge. In contrast, Model-Driven Design environments like Rhapsody support capturing domain-specific software design, but offer limited support for simulation. This thesis describes the use of domain knowledge in empowering simulation environments to support domain-specific modeling. Software design pattern abstractions are identified from the domain and used to extend domain-neutral simulation modeling. To demonstrate this the Façade, Observer, and Strategy patterns from the domain of astronomical observatory (AO) control systems are used to develop a domain-specific extension of DEVSJAVA, a realization of OO-DEVS, called DEVSJAVA-AO. This approach is exemplified with simulation experiments using models developed with DEVSJAVA-AO based on an actual AO control system.

ACKNOWLEDGMENTS

I would like to thank my committee chair, Dr. Hessam Sarjoughian, Department of Computer Science and Engineering, Arizona State University. He has given countless hours of his time towards guiding me in this research as well as mentoring me as a graduate student. From our very first meeting when I was a prospective graduate student, until now, his passion for his research and students has always been evident.

My gratitude also goes out to Dr. Paul Scowen, Department of Physics and Astronomy, Arizona State University, for sharing his domain expertise and providing the opportunity for me to work with the Braeside Observatory. Having full access to the observatory and its control system allowed me to understand the intricate details of how these systems work.

Finally, I would like to thank both Dr. Joseph Urban and Dr. Stephen Yau, Department of Computer Science and Engineering, Arizona State University, for their time reviewing this thesis. Their expertise in the areas of software engineering and thesis writing has been valuable in reviewing this work.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF FIGURES

1. Introduction

1.1. Motivation for Early Design Analysis

The engineering of complex software-intensive systems begins with a rigorous process of gathering technical and non-technical requirements. These requirements are then used to generate a design specification that serves as a blue print for the system being developed. As with many engineering disciplines, a blue print gives instructions on how the product is to be built. Errors in the blue prints can translate into defects in the final product, which in turn can be costly to fix. It is therefore important to validate designs early in the project lifecycle in order to help identify design defects and correct them before they are incorporated into the implementation of the product. The time and cost savings associated with correcting a design defect at this stage versus finding it during testing or even in production are significant.

Validation of a software system design against its requirements can ensure that the system provides the expected functionality and quality attributes. Traditional approaches to design validation include design reviews and requirements traceability. Although useful, such approaches do not explore the dynamic behavior of the design and therefore are limited in their ability to ensure functionality and Quality of Service (QoS) attributes will be satisfied. These approaches primarily serve as a checklist type validation at the end of the design phase. The need to obtain design feedback and validation earlier in the engineering lifecycle gave rise to another technique known as prototyping. With prototyping, certain aspects of an early design can be implemented and presented to the customer for validation while the requirements and design are still open for revision. Design reviews, requirement tracing, and prototyping are all useful tools for

validating design. For these techniques and many others the level of validation that can be performed and when it can be obtained is heavily dependent on the design representation.

Typical software system designs are represented by a document or series of documents. Such documents usually provide specifications and diagrams that capture the structure and behavior of system elements. However such static representations are limited in that they do not allow us the ability to simulate the behaviors of the design components. Simulation of software design components can allow us to evaluate how the system will behave early in the engineering process. The results of these evaluations can help to identify defects in the design that were not obvious during design verification.

1.2. Challenges with Complex System Design

Advances in computer technology have introduced new complexities to system design for solving computing problems. These systems are often built on architectures that are distributed and in some cases multi-processor. The complexity of these systems presents new challenges in designing the software to meet customer requirements. Traditionally systems were judged on how well they met their functional requirements. However, the large amount of time and money invested in development and support of these systems requires attention to the quality attributes the functionality is built upon. Attributes such as maintainability, portability, and scalability have a more lasting impact on the cost of the system over time. These attributes typically cannot be validated until the software is built, at which time it may be too costly to change. Thus there is a growing need to analyze the system design early in the engineering lifecycle to see how well it will meet these quality attributes.

1.3. Existing Design Tools

Due to the large number of components and interactions that comprise complex systems, there is a growing need for a more automated analysis of their software specifications. Traditional software design tools provide engineers with the ability to graphically represent the structure, interaction, and behavior of system components. Although useful for producing design documentation, many of these tools lack the ability to test and validate designs through execution. Commercial tools such as Rational Rose RT (Rational 2006) and Spin (Spin 2006) have provided the ability for software and system specifications to be executed, therefore allowing logical behavior to be tested and validated given the allotted resources and time. The ability to analyze software specifications at this stage of the development lifecycle does help to identify design issues before entering the implementation phase. However these tools rely on execution of detailed specifications and near complete implementations, therefore restricting their use to later points in the design phase. Other approaches such as Model Driven Engineering's (MDE) Domain Specific Modeling Languages (DSML) help to validate the semantics and constraints of models and their interactions in a domain, but still lack support for rigorous simulation of the system components' behaviors (Balasubramanian 2006).

1.4. Simulation for Design Evaluation

Simulation can be used as a unifying artifact in developing conceptual and architectural design of software-intensive systems. Rather than relying entirely on logical and physical system specifications before entering detailed design followed by implementation, simulation enables evaluation of a system architecture behavior. This

capability becomes indispensable since major flaws or shortcomings in a system's architectural specifications can be identified and thus resolved in the early stages of the detailed design and development process lifecycle (Ferayorni and Sarjoughian 2007). This produces benefits such as reduced time to market and lower project costs. Figure 1.1 gives a general view of the traditional approach to design evaluation (blue arrows only) and the proposed simulation based approach (blue and red arrows).

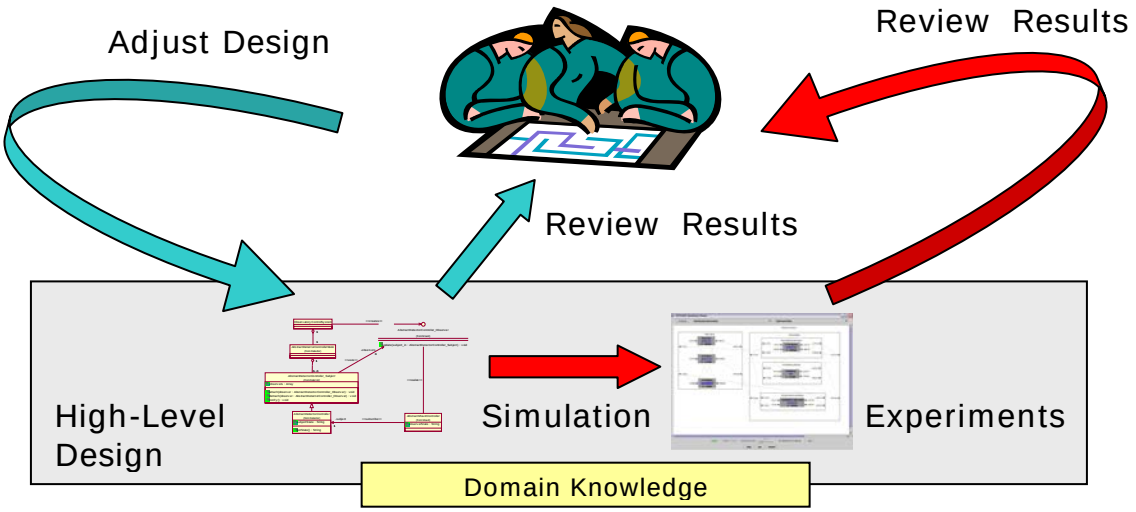

Figure 1.1 Traditional and Simulation Based approaches to design evaluation; the Traditional approach is depicted with the blue arrows; the simulation based approach is depicted with red arrows.

The benefits of simulation throughout various phases of the software development lifecycle have been recognized. Simulation tools such as ProSim have been successfully used to model business process flows and analyze them under varying conditions (Dalal 1997). Although process simulation tools might be used to study the impact of software design influences such as technology and standards choices on the software engineering process, they are not well suited for examination of the actual software architecture of the system being built. Simulation has played an important role in the development of

system/software architectural descriptions, the benefits of which lead to improved system design architecture (Cox 2002). Software product lines have also drawn the attention of simulation. Recent research has proposed the use of simulation for strategic management and long term forecasting of product line development and evolution (Chen 2004). Another example is simulation of an Intelligence, Surveillance, and Reconnaissance (ISR) system which requires synchronized processing of sensor data, prioritizing of resources, and communication among sensors (Hall 2005). This is an example of simulating architecture of a complex, large-scale ISR management system. Although beneficial, the use of the DEVS modeling and simulation framework and others, including HLA (IEEE 2000)(Sarjoughian and Zeigler 2000)(USDOD 2005), in the software development lifecycle remains ad-hoc. Similarly, the use of software modeling methodologies such as Unified Modeling Language – Real Time (UML-RT)(OMG 2006) and Model Driven Architecture (MDA) (OMG 2005) are not well suited for simulation (Huang 2004).

Systems theory (Wymore 1993)(Zeigler 2000) gives us design capabilities such as composition, component connectivity, and time dependent state transitions based on input and output interfaces. Furthermore, Discrete-event System Specification (DEVS) a class of systems theoretic models, provides additional design aspects such as state chart behavior mapping and concurrent execution (ACIMS 2003)(Zeigler 2000). Object-Oriented Discrete-event System Specification (OO-DEVS) incorporates object oriented concepts into its simulation modeling capabilities, but by itself lacks support for domain specific modeling. In order for OO-DEVS to support domain specific modeling it's modeling capabilities must be extended further. This thesis focuses on the use of design

patterns to extend upon the object oriented modeling capabilities of an OO-DEVS simulation environment. The aim is to detail the importance of software design patterns in developing simulation models in the context of a domain.

1.5. Thesis Statement

System-theoretic modeling and simulation frameworks such as Object-Oriented Discrete-event System Specification (OO-DEVS) are commonly used for simulating complex systems, but they do not account for domain knowledge (Zeigler and Sarjoughian 1997). In contrast, Model-Driven Design environments like Rhapsody support capturing domain-specific software design, but offer limited support for simulation. This thesis work describes the use of domain knowledge in empowering simulation environments to support domain-specific modeling. The research outcome shows how software design pattern abstractions extend the domain-neutral simulation modeling. This approach is demonstrated through application of Façade, Observer, and Strategy patterns (Gamma, *et al*. 1995) to an astronomical observatory (AO) command and control system (Braeside Observatory 2005) and development of domain-specific simulation models for the system using DEVSJAVA, a realization of OO-DEVS. This approach is exemplified with simulation models developed based on an actual AO system.

1.6. Thesis Contributions

This thesis defines a methodology of using domain design patterns to extend simulation modeling environments to systematically enable simulation modeling of high level software system designs. These simulation models are then used to study high level

design of the software system and thus expose design issues. These issues are therefore identified early in the design lifecycle, thus saving time and money.

The remainder of this thesis is organized as follows. Chapter 2 presents and discusses background information. This includes a comparison of software modeling and simulation modeling techniques, an introduction to the domain of astronomical observatory control systems, as well as information on the discrete event simulation modeling environment used in this work. Chapter 3 starts a detailed discussion on the approach used to develop domain specific design patterns and using them to extend the simulation environment. More specifically, it looks at how these patterns provide re-usable high level simulation modeling constructs that incorporate domain knowledge. Chapter 4 demonstrates this methodology using DEVSJAVA (ACIMS 2003), the domain of Astronomical Observatory (AO) control systems, and simulation experiments that evaluate the design of a simple AO system. Chapter 5 reviews other related work in the areas of software design analysis, simulation, and design patterns. Finally, Chapter 6 is a thesis summary and discussion of future work in this area.

2. Background

This chapter discusses background material in the area of software modeling and simulation modeling. In addition it will introduce the domain of astronomical observatory command and control systems, and specifically the Braeside Observatory system used at Arizona State University. This domain will be used in Section 4 to demonstrate the thesis contribution.

2.1. Software Lifecycle Challenges

With the increasing demand for complex computer systems comes the pressure to build these systems more quickly and more cost effectively. For a company producing a commercial software application, the time to market can make or break its success. In a corporate IT department on a tight budget, ensuring a project performs to time and resource estimates is critical. Therefore it is well known that good project management and software engineering processes are needed to deliver a product on schedule and on budget.

A host of tools and techniques have been introduced in an effort to reduce the time and cost of the software engineering process. The requirements gathering phase is one area that has improved significantly as a result of these advances. For example, the use of prototyping can allow the customer to test drive the look and feel of different user interfaces. Use cases are another popular tool that allow the customer to represent their functional requirements with a visual medium. Both prototyping and use cases help reduce time and cost in a project by improving the communication between the engineering team and the customer during requirements gathering. The result of this improved communication is a more accurate set of requirements that are well understood

by the customer and the engineering team. These requirements create a solid foundation for transition into the design phase of the project.

The design phase of the software engineering lifecycle continues to be a popular area for research and development of new methodologies and tools. These techniques look to improve many aspects of design such as productivity, representation, accuracy, and quality. The ability to verify the accuracy of a design against its requirements is one aspect that can have a significant impact on the time and cost of a project. For example, design defects identified in the testing phase of a project lifecycle are far more difficult to resolve than if they had been detected earlier. This difficulty in defect resolution is because changes late in the design phase can have an impact on several components in the system. If significant changes are required, it can potentially delay ongoing testing and cause a slip in the project timeline. Therefore, it is critical for the software design process to incorporate methods that will identify defects as early as possible.

2.2. Software System Modeling

The use of object oriented modeling methods and sound architectural principles (including design patterns) have been well utilized in the software design realm to ensure preciseness and quality. *Software modeling* emphasizes structural and behavioral specifications of executable software. Models describe conceptual and formal specification of software prior to implementation and testing activities. For example, Statecharts serve as a suitable basis to describe behavioral blueprint of a system. Statecharts are important for developing detailed software design specifications (Dias and Marlon 2007), and can also be used for simulation (Briand 2004). However, simulating

state space of a (hierarchical) Statechart relies on detailed specifications as they were to be implemented.

At the forefront of software modeling techniques is an approach known as Model Driven Engineering (MDE). A key feature of emerging MDE technologies is Domain Specific Modeling Languages (DSML) (Schmidt 2006). The idea behind DSMLs is their ability to define the relationships between concepts in a domain and specify key semantics and constraints associated with those domain concepts. The languages defined by these meta-models account for domain knowledge therefore supporting a declarative approach to modeling design intent. The second feature used in MDE technologies is model transformation. These are transformation engines and generators that analyze aspects of software models in order to support automated mappings to software implementation artifacts. These mappings help to ensure the design captured in the software models is applied appropriately during implementation (Balasubramanian 2006). The use of MDE technologies incorporating DSMLs and model transformation in software design is motivated from the standpoint of domain driven software modeling and transition to software implementation. In this regard, simulation is not a primary capability and thus the approach is limited to implementation level analysis such as logical design verification. For example, model to model transformations using tools such as C-Saw provide the ability for automated scalability analysis of models (Gray 2006). This type of analysis is focused on software model scalability impact on system constraints, and does not provide any results based on executed model behavior. In contrast, development of simulation models to represent the software design will enable

model execution. Results produced from simulation model execution will support behavioral evaluation of architectural choices against QoS attributes such as scalability.

2.3. Simulation Modeling

In contrast to software modeling, *simulation modeling* is concerned with describing simulations of a system. These model descriptions may range from conceptual foundations to logical operations of systems under varying settings. Therefore, these model descriptions need to offer a variety of ways to experiment with the external and internal workings of a system beyond what could eventually be developed for the real system. For example, alternative models of system architecture can be simulated by composing hierarchical, and/or specialized, model components. Simulation models can also be detailed – e.g., complete component-to-component communication protocols can be simulated as it were the actual software application (Gerla 1999). A central feature of simulation is its support for treating time in logical and/or physical scales (Fujimoto 2000). Logical time and physical time are complementary concepts with the former supporting artificially slow or fast passage of time. The importance of manipulating time in simulation is central to simulation models as compared with software models.

Systems theory provides us with the ability to define a system in terms of its structure and behavior. The structure of system components is modeled hierarchically, whereas the behavior of system components can be modeled in continuous or discrete time. System components can be configured with input and output ports, which when connected to the ports of other components, allowing interaction between them. Discrete-event System Specification (DEVS) is a class of system theoretic models which supports the modeling of hierarchical interacting components that can exhibit autonomous and

reactive based behaviors. System structure and behavior are captured with atomic and coupled models. Parallel atomic models allow for multiple ports that can accept bags of inputs and produce bags of outputs. Parallel coupled models can consist of any number of atomic and coupled models but must 1) consist of atomic models at the lowest level of any coupled model; 2) no coupled model can contain itself; and 3) output to input port coupling resulting in direct feedback is not allowed for atomic and coupled models.

The modeling capabilities of systems theory are well suited for simulation; however they are not intended for complex software design and development (Sarjoughian and Singh 2004). Systems theory gives us design capabilities such as composition and component connectivity. Furthermore, DEVS provides additional design aspects such as Statechart behavior mapping and concurrent execution. Although necessary, these features do not support some important software design techniques available to us in methods such as object oriented analysis and design. For example, the Unified Modeling Language (UML)(OMG 2005) can show the relationships between classes, interfaces, and sub systems in terms of inheritance, aggregation, composition, dependency, and realization. Modeling these relationships allows for the detailed characterization of components and their relationships with one another which are key to software design and implementation.

2.4. Executable Software Architecture

In recent years software architecture has emerged as a crucial step in the design process of complex software systems. The need for software architecture specifications has brought forth tools and standards for documenting and analyzing them. In recent years simulation has been used in conjunction with architecture specification to produce

executable architecture description languages (EADL). Rapide (PAVG 1998) is an event-based, concurrent, object oriented language specifically designed for prototyping architectures of distributed systems.

Architectures in Rapide consist of interfaces, connections, and constraints. *Interfaces* are used to specify the features of components and the behaviors they exhibit. The behaviors of a component's features can be modeled using reactive rules. *Connections* are used to define the communication between components in the system using the interfaces provided by those components. *Constraints* are what allow for restrictions on the behavior of interfaces and connections. By specifying components of an architecture using interfaces, connections, and constraints, Rapide can then perform checks against these requirements under various architecture component configurations.

Execution of an architecture specification using Rapide allows for testing and validation before making implementation decisions. The output produced by an execution is called a partially ordered set of events (POSET). A POSET represents the events that occurred in the execution of the system and their dependence on one another. The dependence of events can be analyzed in two ways, by causality or by time. Causally related events are most commonly the result of reactive rules in interface behaviors, connection rules, and mapping rules. The event generated by a reactive rule is said to be caused by the events that triggered the rule. Events can also be dependent based on the timing in which they occurred. Events that occur at time T+n are said to be dependent on events that occur at time T where $T < T+n$. Figure 2.1 shows an example POSET as viewed in Rapide. In the example below, behavior between a calling system (CIS) and resource (RSC) is modeled as synchronous. Therefore, the POSET result shows that the

CIS must wait for the result of a request to come back from the RSC before sending another request.

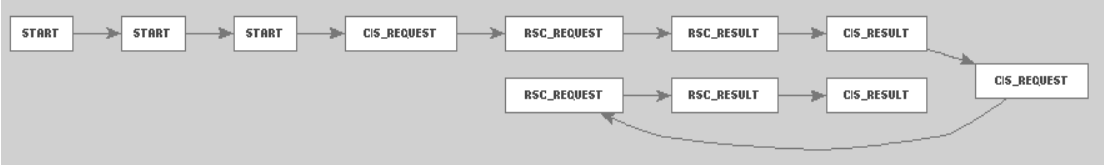

Figure 2.1 Sample POSET in Rapide

EADLs such as Rapide offer support for high level architecture analysis in terms of system components. However, they are limited in their ability to support modeling of high level software design concepts such as design patterns and object oriented concepts such as interfaces/realization. In addition, the behavioral modeling capabilities are limited to reactive events, and are not devised to support autonomous component behaviors. Therefore, the modeling capabilities of EADLs are limited for simulation of software design.

2.5. Astronomical Observatory Control Systems

Computer systems have played a crucial role in advancing research capabilities in astronomy. The science of astronomy is one that requires configurations of hardware and software components to support precise data measurements, accurate handling of timing of actions and events, and data collection. Computers are well suited for such tasks and in addition are not susceptible to human limitations such as fatigue or error when working late into the night. As a result, computers allow astronomers to spend less time worrying about controlling instrumentation and logging data, and more time devising experiments and analyzing the actual data gathered.

In a modern astronomical observatory one would find several components working closely together to carry out observations. Instruments that collect data, such as telescopes and imaging devices, are the most common components. Others might include motors and sensors for movement of the observatory dome, a humidity sensor, or something as simple as a room light switch. It is not uncommon for these and other components of an observatory to be controlled by a computer system. These systems are responsible for coordinating and synchronizing all these components so that observations can be conducted accurately and remotely by astronomers.

Designing software to control an observatory requires in depth knowledge about how these systems work. Understanding the domain of astronomical observatories requires knowledge from those who use them on a daily basis; astronomers. These domain experts can provide insight into the complexity of these systems and help identify important quality attributes. Dr. Paul Scowen, Arizona State University, and Dr. Marc Buie, Lowell Observatory, designed and developed a control system for ASU's Braeside Observatory (Braeside Observatory 2005). Working with these experts helps identify complexities and challenges faced when designing AO control systems.

The Braeside Observatory software system architecture consists of three layers: user interface, application, and data. Figure 2.2 shows each of these layers, their components, and the high level communication links between them.

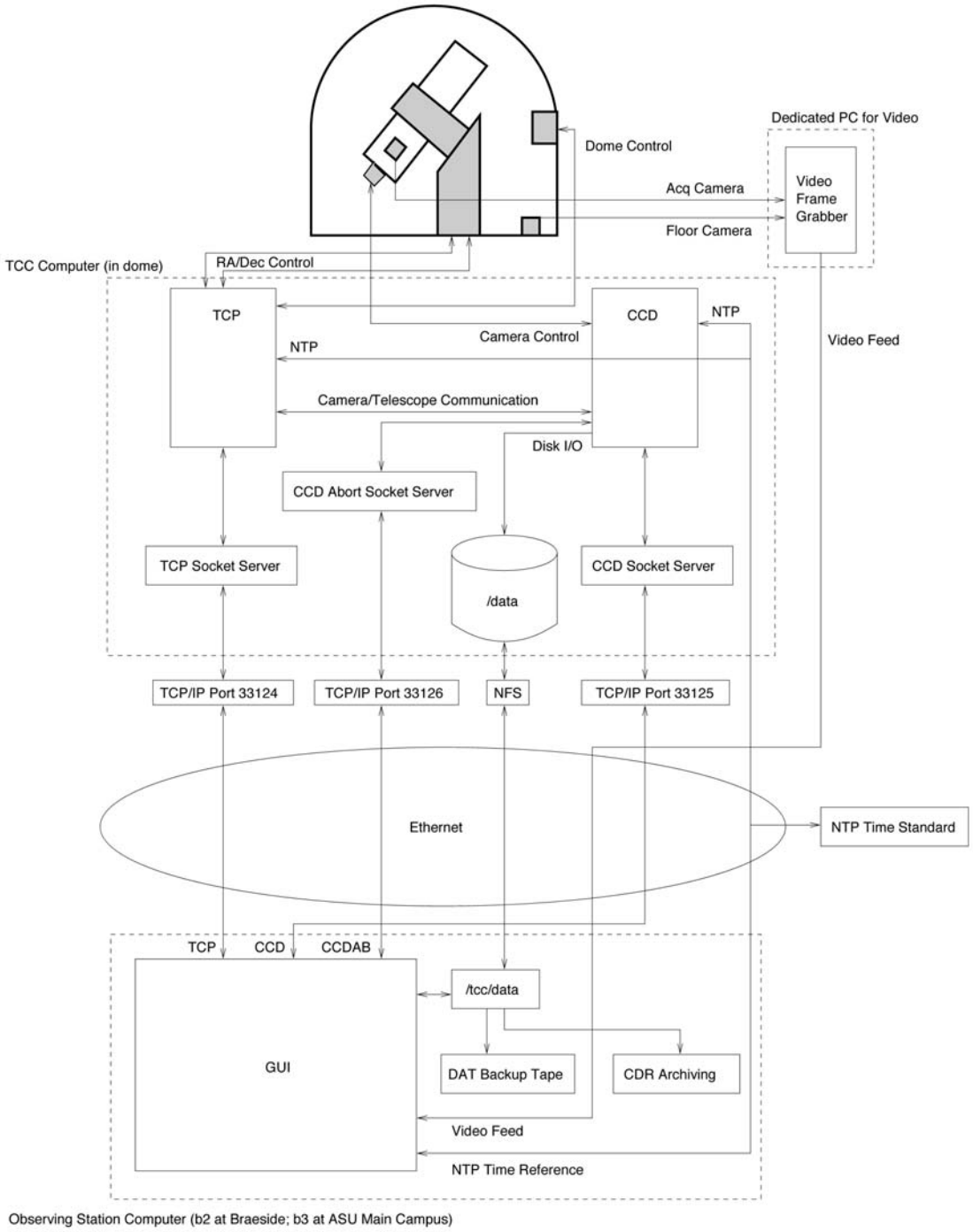

Figure 2.2 Braeside Observatory computer system architecture
Source: [Dr. Paul Scowen, Arizona State University]

The user interface (UI) layer enjoys the benefit of being independent from the application and data layers. The UI layer communicates with the application layer via TCP/IP messaging to dedicated ports on the application server. This layering allows the implementation of the UI to be totally independent of the application layer, and allows the UI to reside anywhere on the internet. The current implementation of the user interface was done with the Interactive Data Language (IDL), and can be run from any operating system with IDL and the Braeside IDL libraries installed. IDL is a programming language that is popular with scientists and researches due to its ease of use, widget based UIs, and image processing capabilities. In addition to the IDL based UI, the system also provides a UNIX command line based UI that can only be used on the application server itself.

The application layer consists of two major components, the Telescope Control Program (TCP) and the Camera Control Daemon (CCD). Each of these programs operates as a daemon process listening for command requests from the UI layer and communicating command responses and data back to the UI layer over dedicated TCP/IP ports. The TCP and CCD also communicate with each other to help with coordination during observation tasks.

The TCP daemon bears the responsibility of controlling all the components in the observatory except the camera, which is managed by the CCD daemon. The most critical of these components is the telescope mount which requires the utmost precision in positioning of the telescope and timing for tracking of an object with minimal error. In addition, the TCP must coordinate movement of the observatory dome so that the dome opening is aligned with the telescope during observations. Other components the TCP

must manage include the telescope focuser and room lighting. The TCP interacts with a PC48 stepper motor control card to facilitate many of the hardware level signals that must be generated to activate motors, switches, and sensors of these components.

The CCD daemon is solely responsible for interfacing between the UI and the CCD camera. The CDD daemon translates user commands coming from the UI layer into commands for the CCD camera to execute, and streams data results back to the user when requested. One key feature of the CCD is its ability to suggest adjustments to the telescope pointing based on data gathered in a CCD exposure. These small corrections are crucial in ensuring the telescope coordinates and tracking are as accurate as possible, and improving long exposure image quality. Once the CCD had identified pointing corrections, the CCD daemon will transfer the corrected coordinates to the TCP through messages in the OS level message queue.

The data layer is simply a separate UNIX server and large hard disk used for data storage. Data is stored on a separate disk in an effort to keep the application server hard disk from filling up with data, and potentially crashing that server. Having the data layer on its own server hard disk also reduces the risk of losing data should the application server hard disk crash.

3. Approach for Domain Specific Simulation with Design Patterns

This chapter introduces a four step approach for extending a component based simulation environment with domain knowledge to support simulation modeling of high level OO software design. The first step explains how to obtain domain knowledge and use it to identify design challenges specific to the domain. The second step will discuss selection of suitable domain specific OO design patterns as solutions to these challenges. The third step will discuss how these patterns can be used to extend a component based simulation modeling environment, thus creating a domain specific modeling and simulation tool. The fourth step will explain how the domain specific modeling and simulation tool can be used to create customized simulation models that represent high level OO software design. Finally, the expected benefits of this approach as well as known limitations will be discussed.

3.1. Overview

This approach introduces design patterns into simulation modeling. Instead of solely using systems theory and object orientation for specifying simulation models, design patterns are used to extend the simulation environment. These patterns capture important traits of common solutions to design challenges in the domain. This kind of simulation modeling provides principled use of design patterns as applied to a domain. With a suitable choice of design patterns, *domain specific simulation model components* (see Figure 3.1) can be created. The result of the extended simulation environment is a collection of simulation model components where relationships among them include patterns of interaction and dependency beyond whole-part and is-a relationships. These components extend the domain-neutral simulation environment's structural and

behavioral modeling constructs with domain specific dynamics. The result is a domain specific simulation environment which can be used to develop specialized simulation models and evaluate alternative architectural or high-level design configurations.

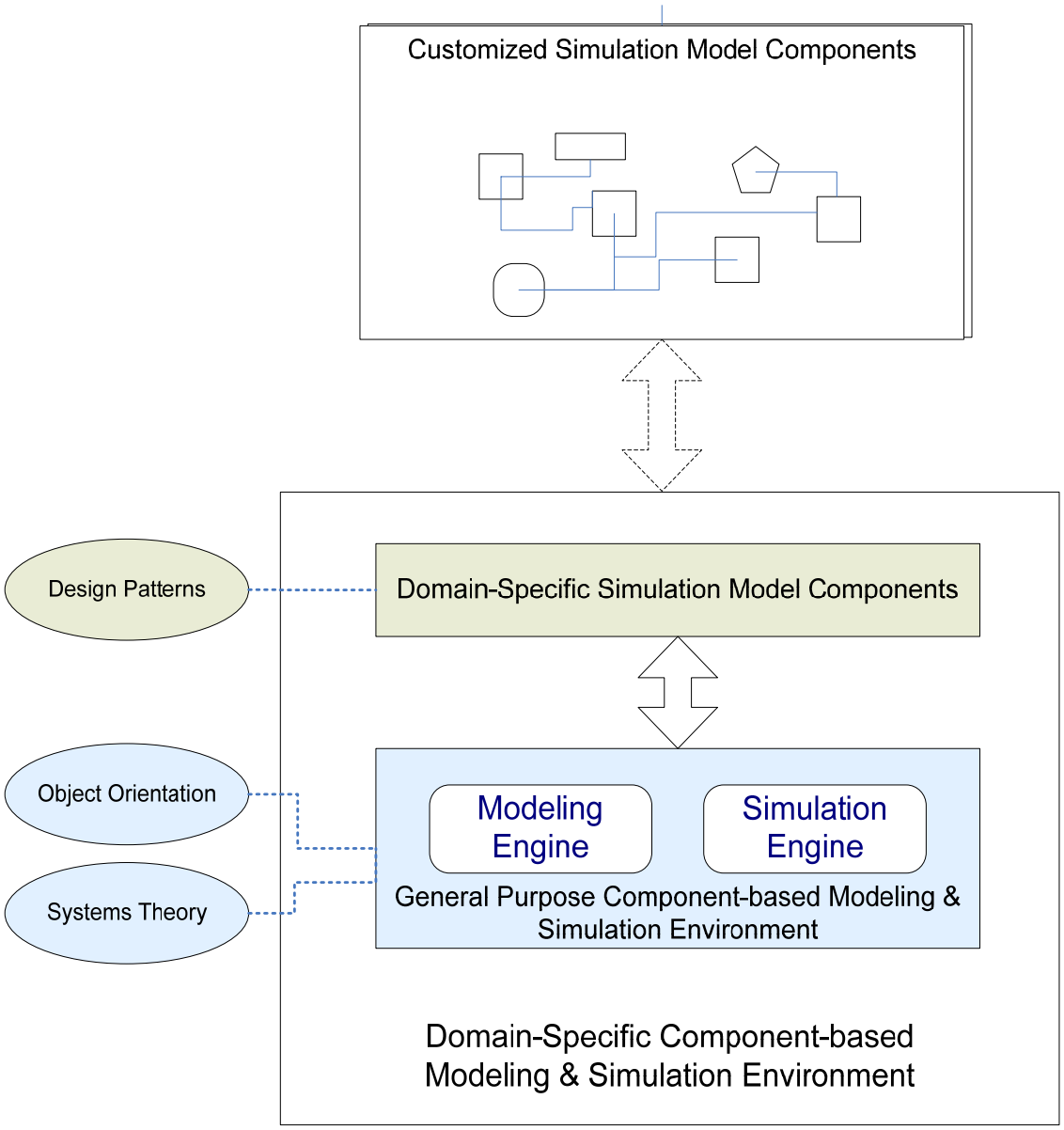

Figure 3.1 Simulation model design using domain specific environment

3.2. Step 1: Gather Domain Knowledge with Use Cases

Simulation supports evaluating the behavior of a system under varying conditions. In order to study high level software design, simulation models that represent the components and interactions of the design must be created. However, identifying these aspects of the design requires knowledge about the problem domain in which the system operates. Domain experts are an excellent source for obtaining this knowledge. They are users and software engineers that are experienced with working in the system domain. Gathering domain knowledge from these experts helps identify domain specific design challenges. Solutions to these challenges can then be incorporated in the high level software design. Simulation models representing these design elements are therefore domain aware.

Gathering domain knowledge can be achieved through generation of use cases (Jacobson, 1992). Use case diagrams support capturing the scenarios (use cases) under which the system will be used from the perspective of its users (actors). Each use case that is generated represents a goal that the user wants to achieve with the system. These goals help to describe the requirements of the system by specifying what is expected of it. However, these requirements should not specify the details of how the system will achieve these goals. Once a set of use cases have been developed for a system they will help expose the high level functions expected of it. Use cases can also expose dependencies between expected functions. Understanding these high level functions and their dependencies will help identify where complex system challenges exist, and enable engineers to start looking for ways to solve them.

3.3. Step 2: Select Design Patterns to Solve Design Challenges

Use cases provide valuable domain knowledge about what functions the system is expected to provide and dependencies between system functions. This information can be used to identify challenges that will be faced in the system design. Many of the challenges faced when designing software have been solved by engineers before on other projects. Solutions to these common challenges have similarities, and are often referred to as design patterns. Design patterns (Gamma, *et al*. 1995) are re-usable object oriented (OO) solutions to common software design challenges. These patterns are solutions that engineers have re-used many times, and that are known to have worked in the past. Because these patterns are based on experience and re-use, they are often flexible enough to be incorporated into many different systems. Every system will have its own design challenges, some of which may be solved by re-usable design patterns. A group of systems from the same domain will often have similar design challenges, and therefore call upon some of the same design patterns to help solve them.

Selection of one or more design patterns to solve a design challenge requires some research. There are many design patterns documented, from the original 20 published by the Gang of Four (*Gamma, et al. 1995*), to the many available on internet web sites. It is the discretion of the engineer to decide if a pattern is appropriate for use. When evaluating a design pattern it is important to think about its intent, and how it can be used to solve problems. Reviewing an example of how a pattern solves a problem can help clarify how the pattern is intended to be used.

Once a set of design patterns has been selected the high level design of the software system begins to take shape. This set of patterns represents solutions based on

domain knowledge and design experience. Evaluation of high level design can start at this point and progress as more concrete classes are defined. In order to enable simulation based evaluation of these high level design elements, simulation models representing these elements must be created.

3.4. Step 3: Extend Simulation Environment with Design Patterns

Incorporating domain knowledge into a simulation environment requires the access to extend its core modeling constructs. Environments that do not support extending the core modeling constructs are not well suited for capturing domain knowledge in the form of design patterns. Many commercial off-the-self (COTS) products do not support extending their proprietary core modeling constructs. It is also important for domain knowledge in a simulation environment to be modifiable so it can evolve with the domain. Many domain specific COTS simulation environments do not allow their domain specifics to be modified by the user. Therefore for this approach it is important to select a simulation environment that is extensible and domain-neutral.

Domain neutral object oriented (OO) modeling and simulation environments provide basic constructs for component based modeling. These core constructs can be extended with OO design patterns to incorporate domain knowledge into the simulation environment. The new extended modeling constructs allow the user to model at a higher level while ensuring domain knowledge is enforced. For example, Discrete-event System Specification provides the systems theory based component modeling constructs, and Statechart based behavior modeling constructs needed for simulation modeling of software design. An object oriented realization of DEVS (OO-DEVS) provides additional modeling capabilities needed to represent OO software design concepts such as

specialization and inheritance. These OO capabilities also support capturing the domain knowledge represented by OO design patterns.

Object oriented design patterns typically consist of a set of abstract classes and interfaces which define methods and behaviors to be implemented by a set of concrete classes. The abstract layer defines what abstract classes are involved in the pattern and what functionality is expected of them. However the modeler has the flexibility to create different concrete implementations of these abstract classes. These concrete classes are also where behavior modeling using the protocol defined by the simulation environment should be done. To enforce use of the simulation protocol, the abstract classes that these concrete classes implement should extend the core behavioral modeling constructs of the simulation environment.

3.5. Step 4: Create Customized Simulation Models

Software designs for systems in the same domain will typically solve some of the same design challenges, and thus incorporate some of the same design patterns to solve them. The domain specific simulation environment developed with this approach aims to capture these common patterns and enforce their use as more customized models are built on top of them. These customized models will be required to follow the structural and behavioral rules defined by the patterns they are built on. However each model still maintains a degree of freedom in how it’s behaviors, both expected and un-expected, are implemented.

The design patterns used to extend the simulation environment typically define abstract classes and interfaces that must be realized by concrete classes. It is the creation of these concrete classes by the modeler that provides the ability to customize the design

of the system. For example, an abstract class from a pattern may require some functionality be provided by the concrete class that implements it. However the specifics of how the concrete class behaves when providing that functionality are customizable by the modeler. Similarly, a pattern may not define a limit on how many objects of a certain type can exists. The modeler can therefore create as many instances of that object as needed to represent the specific object structure of the system being modeled.

Creating customized simulation models in a domain specific simulation environment provides the ability to capture the unique attributes of a specific system design while enforcing domain knowledge through use of design patterns. These customized models can now be simulated and the result evaluated to determine how well the specified design meets functional and quality of service attributes.

3.6. Expected Benefits

This approach allows for developing prominent features of an application domain on top of the general-purpose capabilities of a modeling and simulation environment. There are a number of benefits. First, modelers can take advantage of design patterns to develop domain specific simulation models. Second, since design patterns are incorporated into simulation model components, they can support simulating “software architecture” without first developing simulation models which are close to actual detailed designs. Third, basic differences between simulation and software models can be bridged in a logical fashion since high-impact architectural specifications can be evaluated via simulation instead of delaying them until detailed design, implementation, and testing phases. Consequently this can help with reuse of “solution” simulation models for developing software design models which in turn should lead to improved

time to market and increased quality of the end software/system product. The main benefit of design patterns, therefore, is the ability to create *simulatable software architectures*.

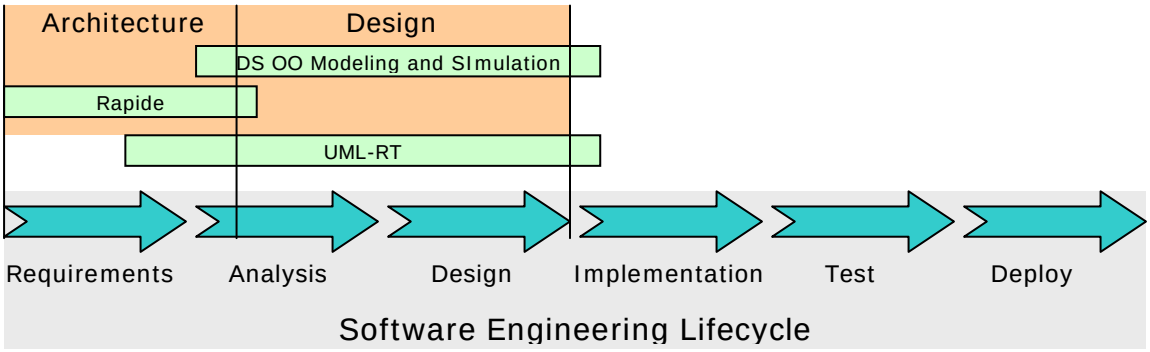

Figure 3.2 Simulation approaches in the software engineering lifecycle

3.7. Limitations

There are many different tools and approaches available for evaluating aspects of software design. Each of these looks to provide some value at different points in the software engineering lifecycle. The approach presented in this thesis of modeling object oriented (OO) software design using simulation environments extended with domain specific (DS) design patterns crosses several stages of the lifecycle. This DS OO Modeling and Simulation approach begins with the analysis phase of the lifecycle, during which use case generation occurs. It continues in the analysis phase with identifying design challenges and then enters the design phase as patterns are selected to solve these challenges. Simulation modeling of these high level patterns now begins, and evaluation of the design can start. Because the simulation models contain some of the same high level design elements as the software models they can continue to be used in parallel for design evaluation. Figure 3.2 shows the general start and end points for use of this approach in relation to the other simulation tools discussed earlier.

4. Demonstration

This chapter will demonstrate the proposed 4 step approach using DEVSJAVA, an OO realization of DEVS in Java, and the domain of Astronomical Observatory control systems. AO domain knowledge will be gathered and used to identify design patterns. These patterns will be used to extend the DEVSJAVA environment into the domain aware DEVSJAVA-AO. Simulation models for a simple AO control system design will be built using DEVSJAVA-AO, and simulation experiments will be executed to evaluate the design. Figure 4.1 shows the tools and models used in this demonstration.

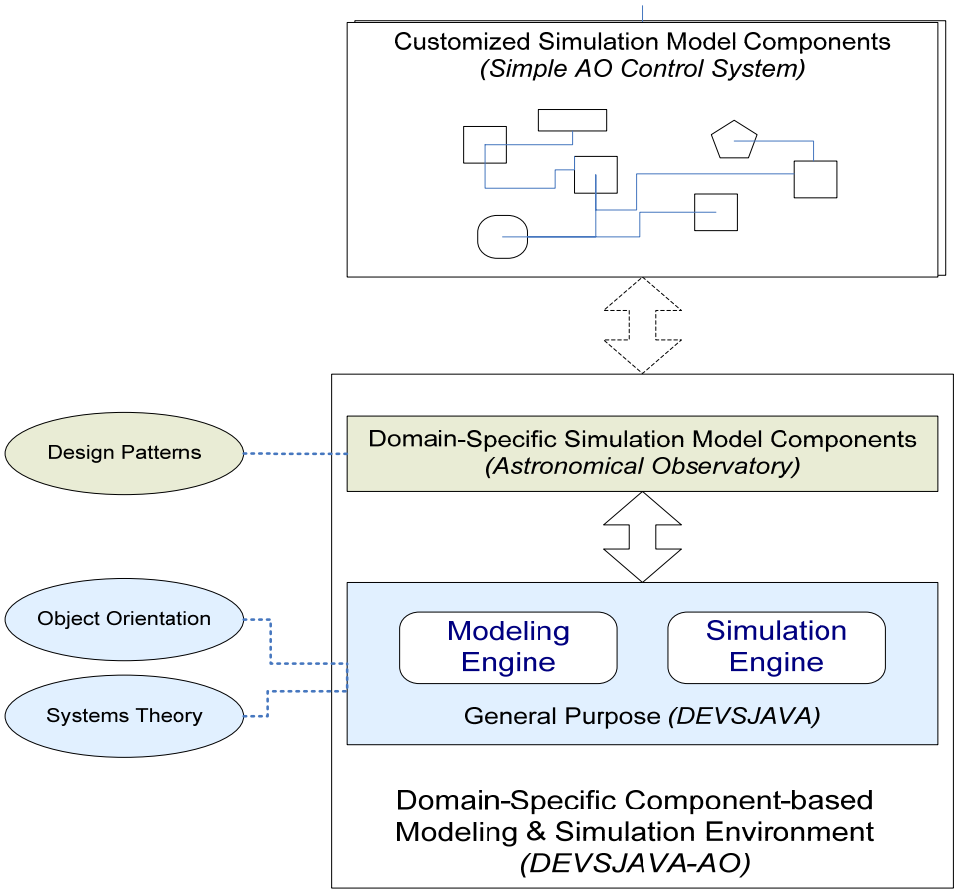

Figure 4.1 Extending DEVSJAVA for the AO domain

4.1. Step 1: Gather AO Domain Knowledge with Use Cases

To better understand AO control systems a group of domain experts must be consulted. Braeside Observatory in Flagstaff, Arizona is owned and operated by Arizona State University. Dr. Paul Scowen (ASU) and Dr. Marc Buie (Lowell Observatory) designed and developed the control software for Braeside. Working closely with these experts domain knowledge was obtained and used to create use cases that help identify the functionalities expected and their dependencies.

During analysis of the AO domain two types of users are focused on; the astronomer and the technician. The astronomer would primarily be concerned with the use of the system during an observation. The technician on the other hand would be interested in the configuration of the system. Three general categories are created to capture how these users would view their needs of the system. The *analysis* category would capture requirements for which data from the system is needed. The *observation* category would cover control of the system during an observation or test. This category would be common between the astronomer and the technician. Finally, the *configuration* category would include needs involving setup of the system components.

The next few sections will cover use cases for the CDD controller in more detail. For a collection of other use cases generated as part of this research please refer to Appendix A.

4.1.1. Identify expected functions

In Figure 4.2 a use case diagram for the CCD camera control module is shown. The actors in this use case diagram are the astronomer and the technician. The CCD system module is represented by a box, and the use cases are shown as ovals within the

system box. Use cases for the astronomer stem from the observation category and include tasks such as starting, stopping, and aborting an exposure. The astronomer also has use cases such as downloading an exposure, which fall into the analysis category. The technician, by nature of being in a support role, inherits all the same use cases as the astronomer, but adds abilities from the configuration category such as registering a new CCD camera. The use cases in this example will most likely map into functions the users can invoke from the user interface layer of the system. However, these use cases also expose a design challenge: the need for further layering of the system in order to prevent tight coupling between user interface and CCD camera control modules.

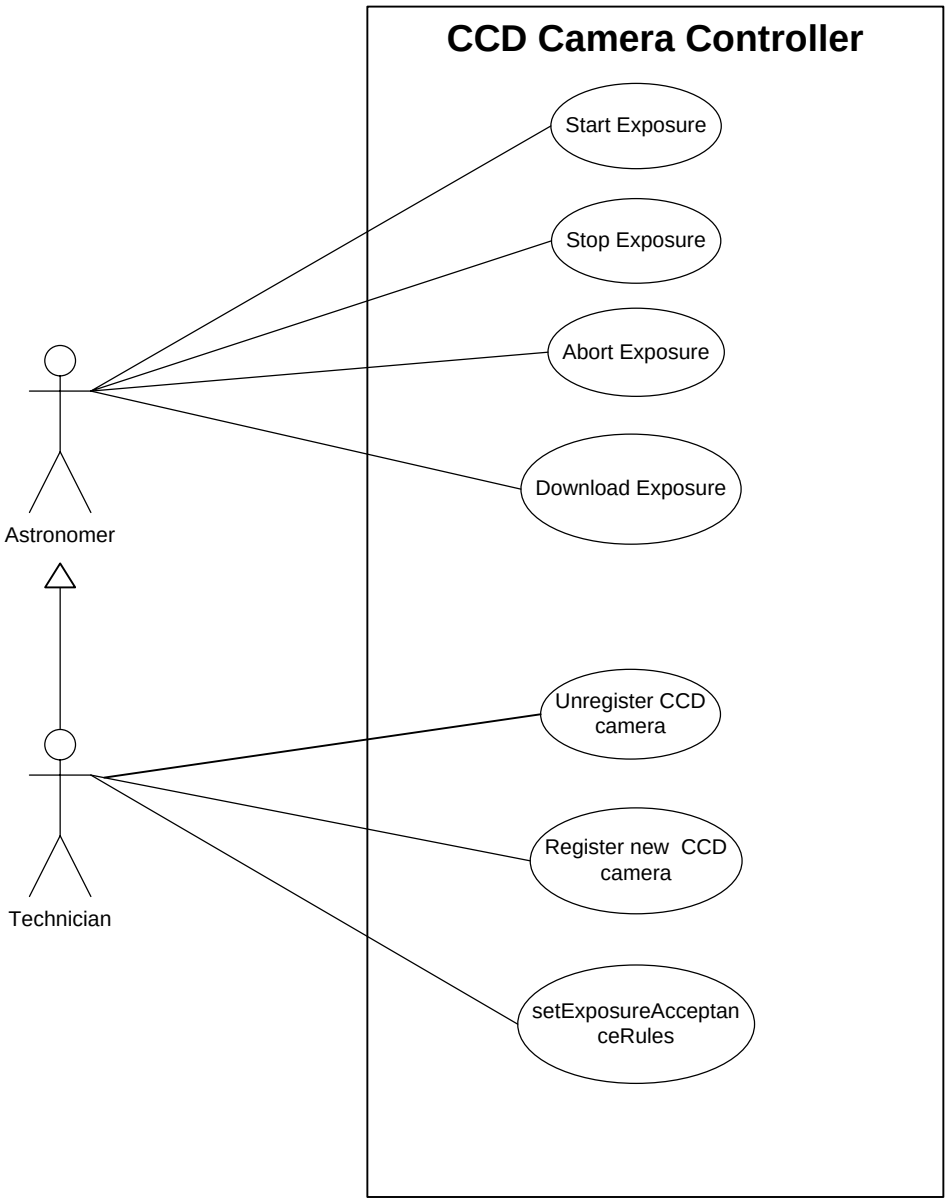

Figure 4.2 Use case diagram for a CCD camera control program

Actor / Use Case Descriptions:

Astronomer: Actor that interacts with the system to take observations and collect data.

Technician: Actor that interacts with the system to perform maintenance.

statExposure: Start taking an image

stopExposure: Stop taking an image

abortExposure: Abort process currently taking an image

startDownload: Begin process to download data from AO system to user's local system

registerCCDCamera: Register a CCD camera controller with the system.

unregisterCCDCamera: Un-register a CCD camera controller with the system.

setExposureAcceptanceRules: Set algorithm to use when deciding to accept an exposure request.

4.1.2. Identify functional dependencies

The previous example showed how use cases can identify the functions users will invoke in the system. In addition to these functions, use cases can also expose what other system functions are carried out as a result of the user invoked functions. These functional dependencies are often referred to as an "include" relationship between use cases. Figure 4.3 shows that when the user invokes the Start Exposure use case in the CCD control system, this includes invoking functionality in the Telescope control system that disables telescope slewing. This behavior of the system is required to prevent new slew requests from being executed during an exposure. Similar behavior can also be seen in this use case diagram such that when the exposure is stopped or aborted, the slewing is enabled again. This use case exposes a design challenge: a state change dependency relationship between two major components in the system. The next section will discuss how this relationship can be included in the design by allowing a component to observe the state changes of another component, and take appropriate actions.

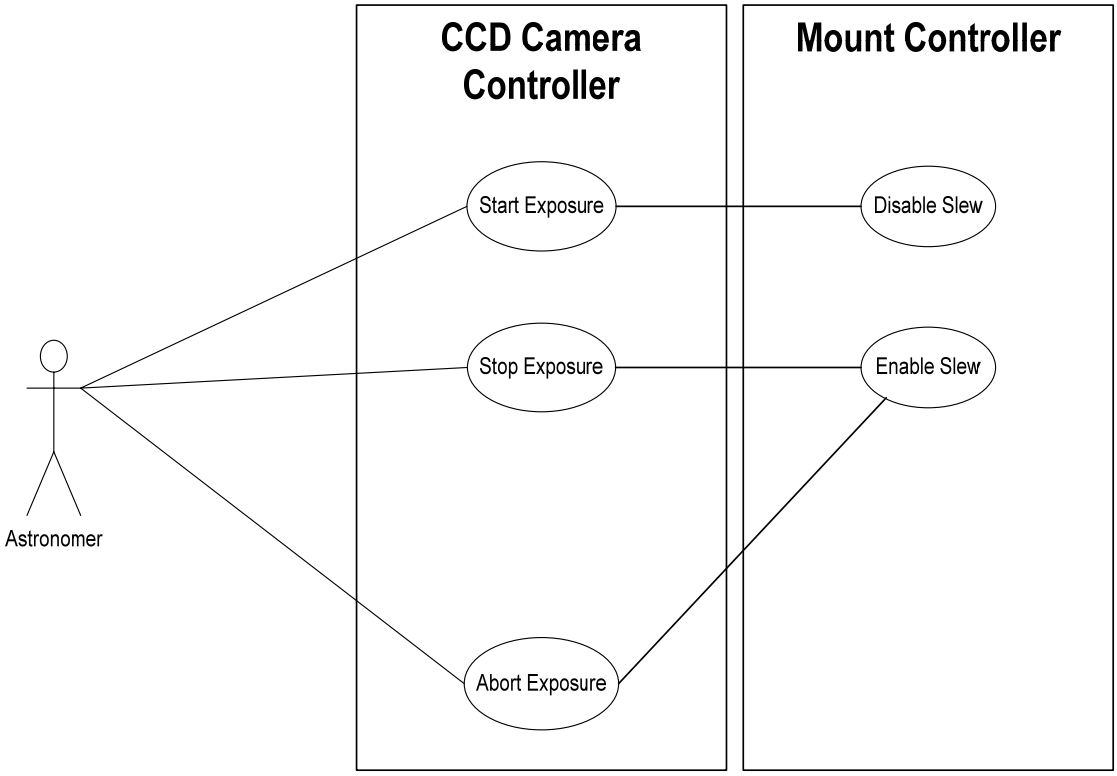

Figure 4.3 Use case extension from CCD controller to Mount controller

Actor / Use Case Descriptions:

Astronomer: Actor that interacts with the system to take observations and collect data.

Technician: Actor that interacts with the system to perform maintenance.

startExposure: Start taking an image

stopExposure: Stop taking an image

abortExposure: Abort process currently taking an image

disableSlew: Disable the ability to slew.

enableSlew: Enabe the ability to slew.

4.2. Step 2: Select Design Patterns to Solve AO Design Challenges

Observatory control systems present several design challenges that must be analyzed. Meeting with domain experts and generating use case diagrams for the system allows these challenges to be identified. This section will discuss some of the key challenges of AO control systems, and how domain specific design patterns can help solve them. Additional design patterns can be found in Appendix B.

4.2.1. Decoupling layers with the Façade design pattern

In the Braeside Observatory command and control system there are several sub systems each providing control over different components. Over time these sub-systems may need to change as instruments and devices are upgraded. An example of this need for system re-configuration can be seen with the CCD camera used to capture images through the telescope. Earlier, a use case diagram was developed to capture common high level commands that a user would issue to interact with the CCD camera through the system. As CCD technology evolves and new techniques for optimizing CCD image capture arise, new functionality will be added to camera systems. For example, many modern systems now offer add on modules such as advanced cooling, external dew control, and expanded filter wheels. These changing features in CCD technology often prompt researchers to upgrade their old CCD camera to newer models, or camera manufacturers to change their APIs to accommodate new image capture methods. If the user interface layer or application layer interact directly with these sub-systems they will encounter maintenance issues as those sub-systems evolve. A façade can therefore be introduced to help de-couple the application layer from its client (user-interface). Use of a

façade can be incorporated at one or more levels in the design depending on how the system needs to be layered.

The use of the Interface classifier in design allows us to show what functionality is expected (syntheses, interaction, and collaborations) but not how that functionality will be achieved. The method signatures for each operation of an interface will specify what inputs are to be given and what outputs are expected. The details of how interface operations will be implemented are left to the model classes that realize them.

The concept of interfaces maps to the first design pattern for the AO domain. The façade design pattern provides a unified interface to a set of sub-systems. AO control systems are comprised of many sub-components that work together to complete a user request. Once interfaces that capture the services provided to the user have been identified, all or part of those interfaces can be combined to create a façade. This higher level interface will hide the details of how sub-system components are used to execute the request. In addition, changes to how the sub-system components carry out the request can be made without impacting the user of the façade. This layering through use of the façade is an important pattern for the AO domain because instrumentation is frequently upgraded to stay atop research needs and evolving technologies.

Figure 4.4 shows the use of an interface (*DetectorControllerInterface*) in support of the façade pattern between a client of the observatory (*ObservatoryClient*) and the detector controller (*DetectorController*). This controller may be implemented with one software component (as shown in this example), or with coordination of many sub-components. The example below also shows two types of specialized detector controllers (*CCDCameraController* and *SpectrometerController*) and how the façade can hide the

details of which controller is being used. Therefore applying the façade pattern allows us to hide the details of how the interface methods are actually carried out.

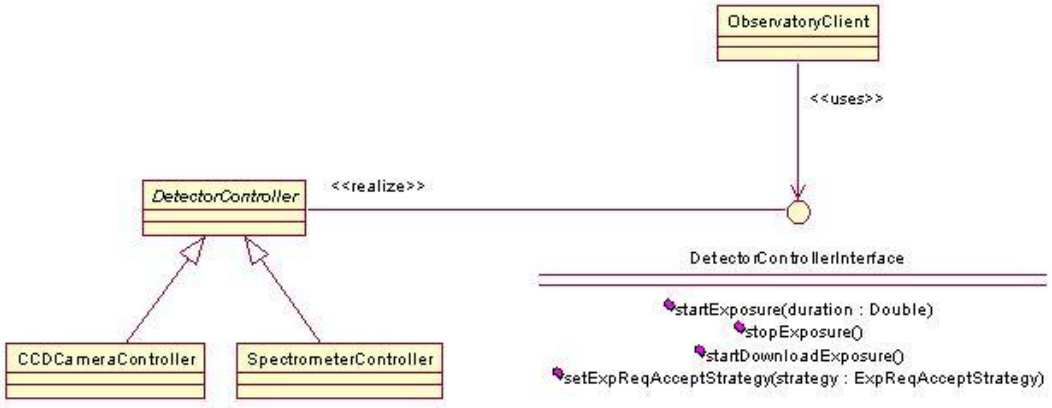

Figure 4.4 Facade design pattern for the AO domain detector controller

4.2.2. Component synchronization via the Observer design pattern

Another key design challenge in the AO domain is the communication between individual software components in the observatory control system. Several of the instruments and mechanics in the observatory must be managed by dedicated software modules. These modules can be separate processes running on the same server or in some cases on different servers. In many scenarios these software processes must share information with one another in order to complete a task. Earlier this type of interaction was seen in the use case diagram for starting and stopping exposures. An extension of the Start Exposure use case with the CCD control program was to notify the mount control program so that it could disable slewing. An example of this interaction was seen in the Braeside Observatory control system where a communication link exists between the TCP daemon and the CCD daemon. That implementation made use of OS level message

queues to share information between the daemon processes. Another approach to solving this problem is to have one process observe another process in order to look for state changes. This pattern of interaction between components is commonly referred to as the “observer” design pattern.

The *observer* pattern allows for components (the observers) to be notified when the state of another component (the subject) changes. It is also referred to as Publish-Subscribe or Subject-Observer. This pattern is important for the AO domain because subject component state changes are often shared with many observing components that may vary from one system configuration to another. For example, when the detector is taking images the mount will need to block any incoming slew requests from the user. Similarly when the detector is finished taking an image the mount will need to unblock. This common coordination between the control software of the mount and detector can be managed via the observer pattern. Furthermore, a new system configuration may introduce a second detector that also needs to be observed by the mount. In this case the pattern supports the client subscribing the new observer to the subject through well defined interfaces.

Figure 4.5 shows how the observer pattern can be used to allow a mount controller (observer) to subscribe to state change notifications by the CCD controller (subject). In this example the *DetectorController* class is the subject, and extends the *DetectorController_Subject* abstract class which defines the subject interface methods that must be implemented. The subject’s *attach* and *detach* methods are called by observers to subscribe and unsubscribe respectively from state change notifications. The *notify* method is used by the subject to send its current state to all subscribed observers.

The *MountController* class is the observer and implements the *DetectorController_Observer* interface which defines the observer methods that must be implemented. The observer's *update* method is called by the subject passing the state, and allows the observer to define what actions to take.

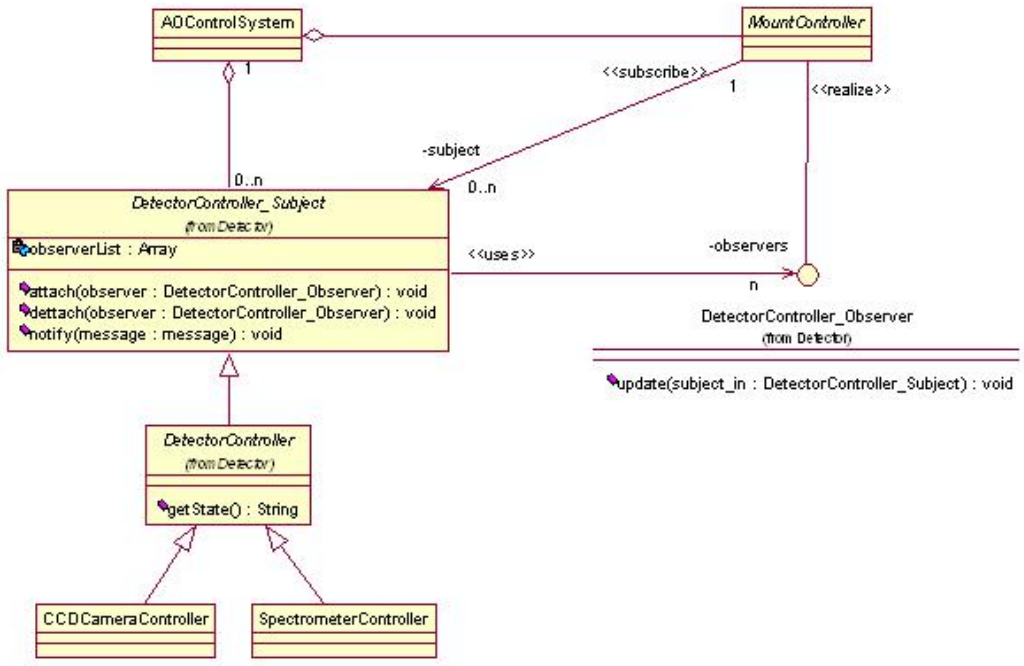

Figure 4.5 Observer design pattern used by AO detector and mount

4.2.3. Modifying control algorithms using Strategy design pattern

AO control systems consist of many algorithms that define the behavior of the system in different scenarios. In some cases, the algorithm used for controlling a certain aspect of the system needs to be changed frequently depending on the observing being done. In other cases an algorithm may need to be modified due to new requirements of the system. In many systems these algorithms are difficult to locate because they are buried in thousands of lines of code. Once found, these algorithms can be difficult to

change without impacting many other pieces of the system. Having a system design that allows simplified modification to these algorithms can save time and money.

The *strategy* design pattern allows a family of algorithms to be defined and provides access to them through a standard interface. Clients can then change between different algorithms in the family without having to change the way the algorithm is called. One of the use cases presented earlier showed that the technician may need to change algorithms in the CCD control program. The strategy pattern allows these changes to be made easily and with minimal impact to the rest of the system. For example, a technician may need to change the algorithm that determines whether the CCD control program will accept an exposure request or not. One algorithm, called "Always in View", might require that the coordinates the system is currently pointed to are above the horizon for the entire requested exposure length. Another algorithm, called "Now in View", may simply accept any request and not be concerned about the current coordinates falling below the horizon before the exposure time is over.

Figure 4.6 shows how the strategy pattern can be used to solve this design problem. The abstract class *ExpReqAcceptStrategy* defines the algorithm interface which is a method *checkExpReqForAcceptance* that accepts all the parameters that might be needed to make the decision. There are two concrete classes, *AlwaysInViewExpReqAcceptStrategy* and *NowInViewExpReqAcceptStrategy*. Each implements the *checkExpReqForAcceptance* interface method but with different decision logic. Also shown is how each client will have an object of type *ExpReqAcceptStrategy*, and must provide an attach method that takes an *ExpReqAcceptStrategy* object as a

parameter. The attach method can be used to assign a different algorithm from the *ExpReqAcceptStrategy* family to the client.

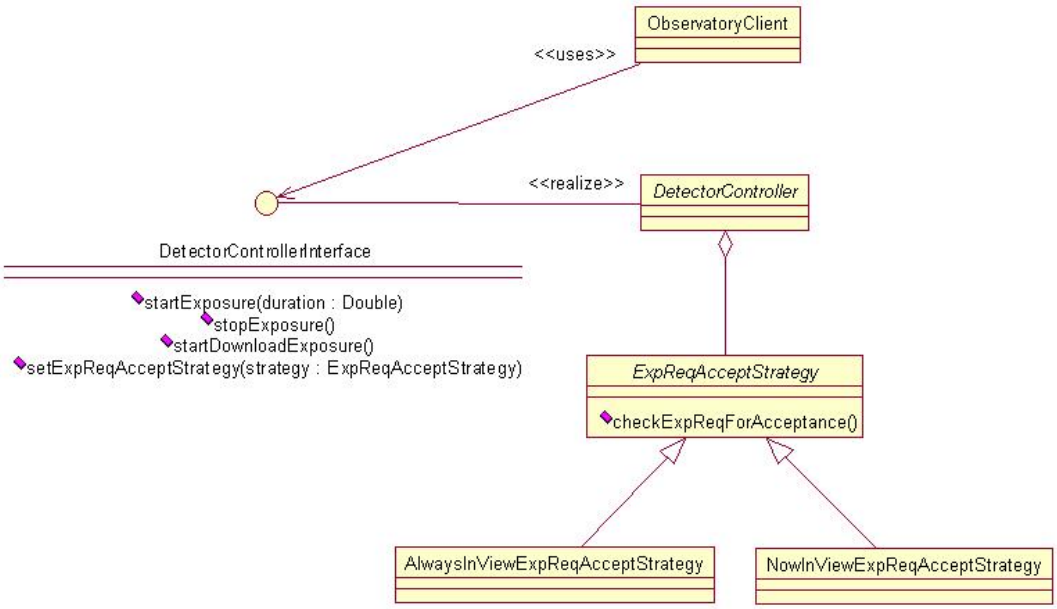

Figure 4.6 Strategy design pattern used by AO detector controller

4.3. Step 3: Extend DEVSJAVA with Selected AO Design Patterns

Implementation of the AO system simulation models will require an environment that supports object orientation and provides the ability for extension of the core components with design patterns. Commercial Off The Shelf (COTS) simulation packages generally do not allow access to core components of the environment, and therefore are not well suited for extension with design patterns. Simulation packages such as DEVSJAVA and SimPy (SimPy 2004) support modeling using object orientation and also allow for extension of core environment components. This research extended the DEVSJAVA environment with AO domain design patterns to create the DEVSJAVA-

AO simulation environment. This environment was then used to create simulation models representing the software components of a simple observatory control system. The following sections will look at the implementation details and discuss challenges faced.

4.3.1. Extending DEVSJAVA to DEVSJAVA-AO

At the foundation of the DEVSJAVA-AO environment is its extension of the DEVSJAVA classes. The two primary modeling constructs in DEVSJAVA are atomic and coupled models, which are made available with basic visualization through the *ViewableAtomic* and *ViewableDigraph* classes. These core modeling constructs provide the link to the simulation environment, as well as the part-of and is-a modeling relationships needed to develop simulation models. However with these components alone there are still many ways in which their ports can be defined. This variability in port definition presents a problem as more and more models are created because they will need to know the port names of those models they interact with. This limits the ability to easily re-configure the model to model couplings. Thus the first step taken in the AO specific modeling environment is to standardize the definition of port names for atomic and coupled models.

The *AOControlEntity* and *AOControlNode* extend the *ViewableAtomic* and *ViewableDigraph* classes respectively (see Figure 4.7), adding standard port name definitions. Every component will have two in ports and two out ports. The inCmd and outCmd ports are used to move commands in and out of model, while the inData and outData ports will move data in and out of the model. Models created in DEVSJAVA-

AO can now easily be coupled together because they share the same simple interface port definitions.

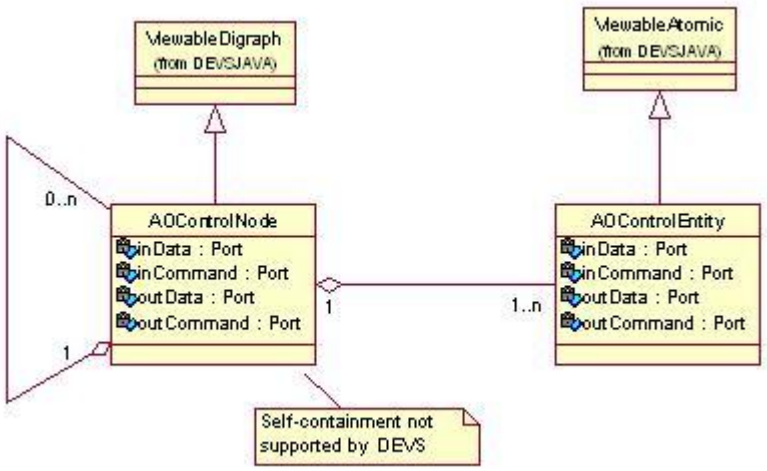

Figure 4.7 Extending DEVSJAVA core modeling constructs for AO domain

Three major software components of an AO control system are those controlling the mount, detector instruments, and telescope accessories. For this example AO system the mount, detector and focuser controllers are modeled. Because there can be different types of mounts, detectors, and focusers, each of these is first modeled with an abstract class capturing any general attributes and behaviors. These abstract classes are then specialized to capture attributes and behaviors specific to different types of controllers. This specialization is seen in Figure 4.8 with the *MountController*, *DetectorController*, and *FocuserController* forming the abstract class layer, and the *ForkMountController*, *CCDCameraController*, and *CatadioptricFocuserController* forming the specialized layer.

At this point object oriented concepts such as composition, abstraction, and specialization have been utilized to extend the DEVSJAVA environment into the DEVSJAVA-AO environment. These same concepts are commonly used in software modeling, and will be a part of the AO control system software design. The simulation models therefore align with software modeling goals. Another software modeling concept, design patterns, will also be used in the simulation modeling. The next few sections will discuss how these patterns were implemented and the challenges faced in doing so.

4.3.2. Implementation of the AO façade design pattern

The simple AO control system design shown in Figure 4.8 uses a separate controller for each of the mount, detector, and focuser. However, other AO system configurations may be different. To support configurability without impacting the users of the system the façade design pattern can be utilized. This pattern is utilized by having each controller abstract class realize the corresponding interface. Thus the *MountController* implements the *MountControllerInterface*, the *DetectorController* implements the *DetectorControllerInterface*, and the *FocuserController* implements the *FocuserControllerInterface*. These interfaces define the methods that must be implemented by the abstract class itself, or a specialization of it.

One interesting note with the use of interfaces in DEVSJAVA-AO is that the Java programming language requires that their methods be public. However it turns out that when these interface methods are implemented they are actually being called in a private sense. These methods are private because atomic models in DEVSJAVA do not communicate via direct method calls with one another, but instead through passing

messages to their DEVS port interfaces. The external transition function is where incoming messages are processed, and it is there that the model determines which façade interface method should be called. These methods are therefore private to the model, and do not benefit from being defined as public.

4.3.3. Implementation of the AO observer pattern

The observer pattern is utilized for state change notification between the mount and detector. In this example system the detector controller is represented with a single atomic model named *DetectorController*, this class can act as the subject by inheriting from *DetectorController_Subject* and providing access to its state. The mount controller is also represented with a single atomic model named *MountController*, which can be setup as an observer of the detector controller because it realizes the *DetectorController_Observer* interface.

There are some differences in how the observer pattern is used in simulation modeling versus how it is traditionally used in software modeling. For example, in software modeling the subject directly calls the *update* method of the observer, passing the subject's state. The only requirement is that the subject and observer objects be in the same scope or contain references to one another so they can call each others methods. However simulation modeling does not permit atomic models to directly call the methods of other models. Therefore the subject cannot directly call the *update* method of the observer. Instead it must pass a message from its output port to the input port of the observer, and the observer's external transition function must handle the message and know to call its own update method. Another issue is that in order to enable this port to port message passing, the coupled model that contains the subject and observer atomic

models must explicitly couple their ports. There is not a straight forward way to establish this connection from the *attach* method within the atomic subject model.

4.3.4. Implementation of the AO strategy pattern

In the AO control system the strategy pattern is used by the detector to allow easy configuration with different exposure request acceptance algorithms. The strategy interface is defined by the *ExpReqAcceptStrategy* abstract class. This abstract class defines the method signatures but does not provide any implementation for them. The implementation details are left to the specializations since they will differ for each algorithm. For exposure request acceptance there are two algorithm strategies: *NowInViewExpReqAcceptStrategy* and *AlwaysInViewExpReqAcceptStrategy*. The "NowInView" algorithm will accept a request as long as the current system coordinates are in view at the current time. The "AlwaysInView" algorithm will accept a request as long as the current system coordinates are in view now and will be above the horizon when the exposure completes. In this example the *DetectorController* provides the *ExpReqAcceptStrategy* member variable and a *setExpRequestAcceptStrategy* method for allowing configuration with a given *ExpReqAcceptStrategy* strategy.

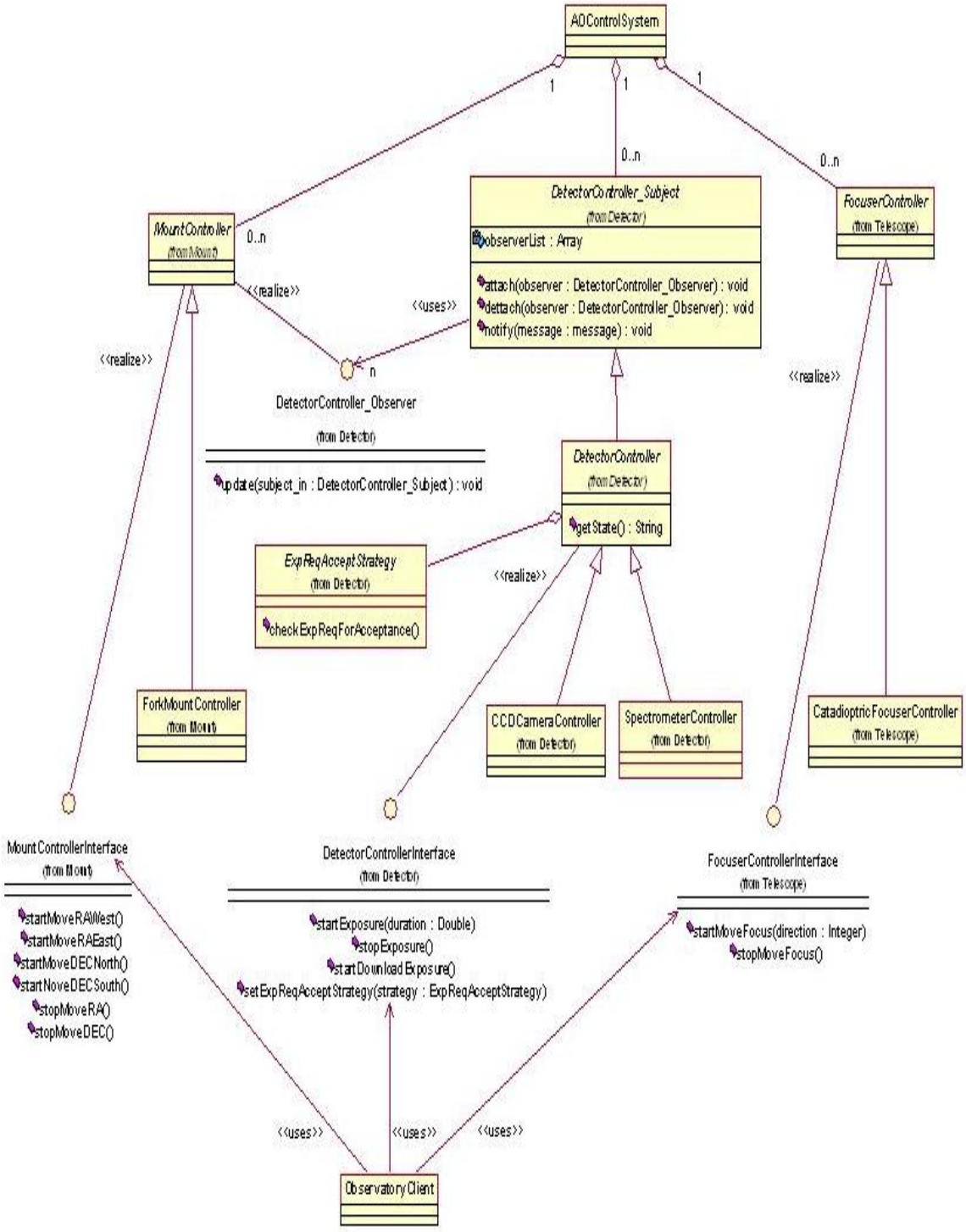

Figure 4.8 Simulation models for simple AO control system

The strategy pattern is well known and used in many software system designs. To make use of it in the AO system simulation environment is fairly straight forward. One reason implementation challenges are not as common here is because the algorithm is an internal method, and not a method that is called by another system object. Thus the algorithm is typically not called upon until a related state change requires it to be invoked.

4.4. Step 4: Create Customized Simulation Models for AO System Design

For this demonstration a simple set of customized models was created using the DEVSJAVA-AO environment. The observatory being modeled was similar to the Braeside Observatory and consisted of a catadioptic telescope on a fork mount with a CCD camera attached for imaging. The control system design was simple in that there was one controller for each major component.

The fork mount controller was modeled using a specialization of the *MountController* abstract class called *ForkMountController*. This class modeled all the behavior of a fork mount controller, including object tracking and slew capability along the right ascension and declination axes. Because it inherits from the *MountController* abstract class, two design patterns are enforced. The façade pattern is supported because the methods of the *MountControllerInterface* are realized. The observer pattern is also supported because the methods of the *DetectorController*_Observer are realized.

The CCD detector controller was modeled using a specialization of the *DetectorController* abstract class called *CCDDetectorController*. This class modeled all the behavior of the CCD detector controller including behaviors such as taking and downloading an exposure. Because it inherits from the *DetectorController* abstract class,

three design patterns are supported. The façade pattern is supported because the methods of the *DetectorControllerInterface* are realized. The observer pattern is supported because the methods of the *DetectorController_Subject* interface are realized. The strategy pattern is supported by inclusion of the *ExposureRequestStrategy* object, and realization of the setExposureRequestStrategy method.

The catadioptic focuser controller was modeled using a specialization of the *FocuserController* abstract class called *CatadiopticFocuserController*. This class implements behaviors of the focuser controller such as moving focus in and out. Because it inherits from the *FocuserController* abstract class it also supports one design pattern. The façade pattern is supported because the methods of the *FocuserControllerInterface* are realized.

4.5. Simulation Experiments using DEVJAVA-AO

The DEVSJAVA-AO environment provides the base classes necessary to build atomic and coupled models for a simple AO control system. In addition, it provides classes for basic block diagram visualization of simulation executions. Figure 4.9 shows the DEVSJAVA-AO simulation view for a system configuration that includes one detector controller for a CCD camera. The *AOControlSystem* coupled model contains all atomic models in the AO control system. The “AO Control System Client” is actually a group of models that generate data over time trajectories and collect results for analysis over time periods. These simulation results are evaluated to choose suitable command and control designs under a range of operational settings.

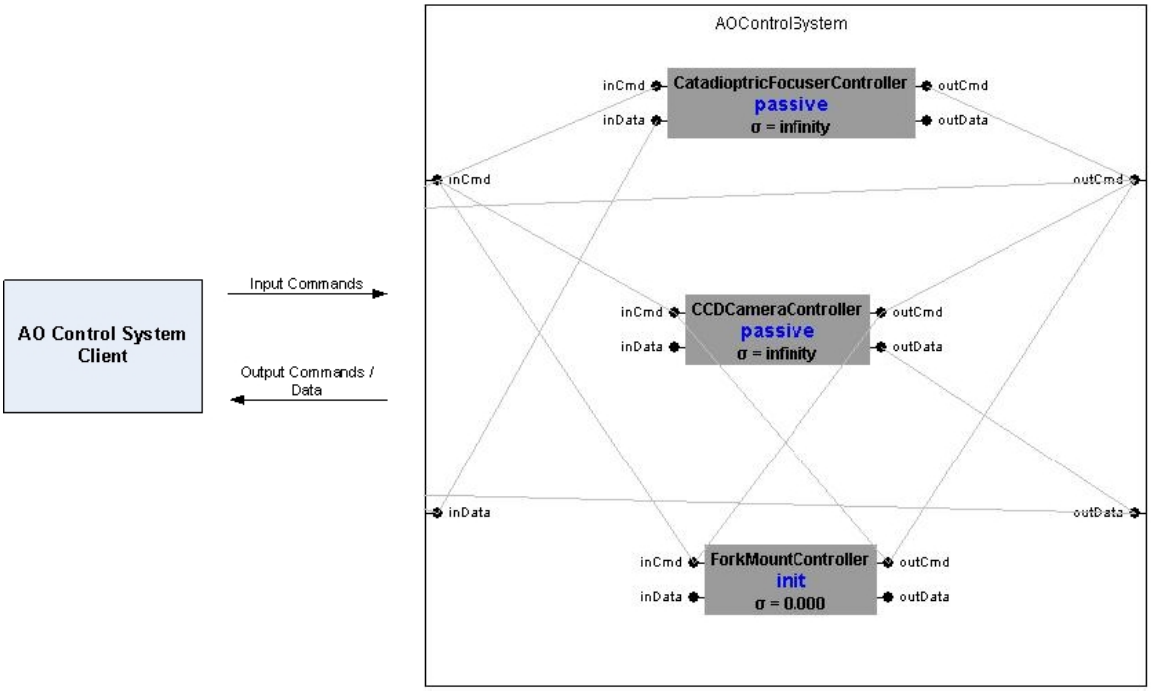

Figure 4.9 Simulation view for AO system configured with one detector controller

Simulation provides the ability to run experiments on a system and learn about its behavior under certain conditions. The models described here of a simple AO control system can be simulated in various experiments to measure many aspects of the system such as correctness, performance, and "what if" scenarios.

4.5.1. Analysis of system configurations

One interesting aspect of software architecture simulation is its ability to test how addition of a new module will impact the rest of the system. The first AO control system configuration introduced earlier contained only one CCD Camera detector, and thus one controller. However, another AO control system configuration might have a second detector, such as a spectrometer for measuring properties of the light. Therefore a second controller may be needed for this new Spectrometer detector. Addition of a new spectrometer controller to the system provides an opportunity to test how well the façade,

observer, and strategy design patterns support QoS attributes such as configurability and modifiability. In addition, simulation tests can allow execution results to be captured and analyzed to validate the systems behavior under this new configuration.

Introduction of the *SpectrometerController* class to the system is easily done as a specialization of the *DetectorController_Subject* abstract class. This specialization allows it to inherit the subject methods that are part of the detector-mount observer design pattern and the *DetectorController_Interface* methods that are part of the façade design pattern. Finally, when implementing the *SpectrometerController* class, the *setExpReqAcceptStrategy* method can be provided from the strategy design pattern to allow different acceptance algorithms to be set. The design patterns therefore allow the system to be easily modified. In addition, simulation runs can be performed to verify that the new *SpectrometerController* functions correctly and that the other system components are not negatively impacted. Figure 4.10 shows the DESVJAVA-AO simulation view of this new configuration. The ability to execute the system architecture in this manner provides information that would not be available if the design was only on paper.

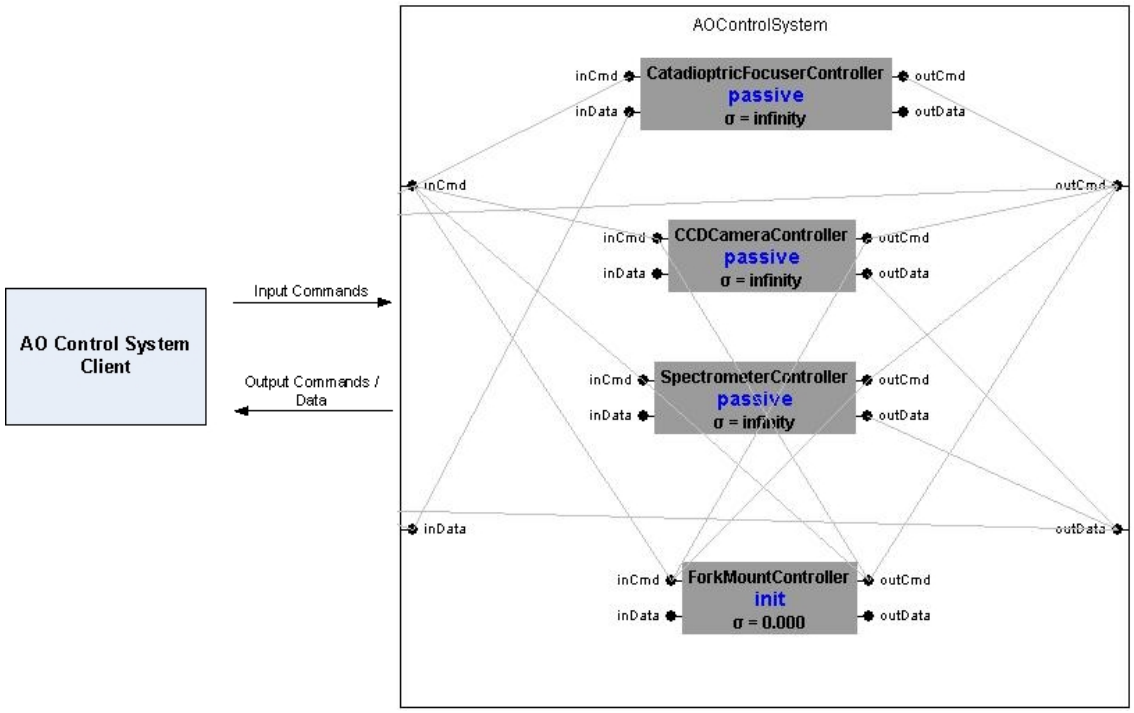

Figure 4.10 Simulation view for AO System configured with two detector controllers

4.5.2. Analysis of system behavior

Another aspect of software that can be tested is the behavior of the system when different algorithms are used to perform certain system functions. For example, when an exposure request is received the system must determine if the request is valid. Suppose there are two algorithms to choose from for implementing the CCD exposure request acceptance logic. The first algorithm will check only if the current coordinates are above the horizon. With this algorithm the detector software will simply check that the current coordinates are above the horizon. If they are, the request is accepted and the exposure begins. If they are not, then the request is rejected. The second algorithm will check if the current coordinates are above the horizon now and that they will be above the horizon for the length of the exposure. To check this rule the algorithm takes the current time plus the

length of the exposure to get the time the exposure will end. The algorithm then checks if the current coordinates will be above the horizon at that time. If they are, the request is accepted and the exposure begins. If they are not then the request is rejected.

To conduct the experiment the Strategy design pattern is utilized, and two specializations of the *ExpReqAcceptStrategy* abstract class are created: *AlwaysInViewExpReqAcceptStrategy* and *NowInViewExpReqAcceptStrategy*. The use of this pattern allows configuration of a detector controller with the desired strategy by simply passing an instance of it to the controller's *setExpReqAcceptStrategy* method. Changing to the other strategy requires a one line code change to pass an instance of the other strategy to the method.

To see the impact this algorithm change has on the system, simulations are conducted in which observation requests generated by the experimental frame are carried out by the AO control system. These requests are created by selecting random coordinates and a random exposure time. The exposure time is limited by a maximum value that is increased by two hours with each simulation run. A successful observation request is one in which the coordinates of the object being imaged can be tracked by the telescope from start to finish of the exposure. This scenario is considered to return the desired image. A failed observation request results if the exposure is cut short because the object goes below the horizon. This scenario is considered to return an erroneous image.

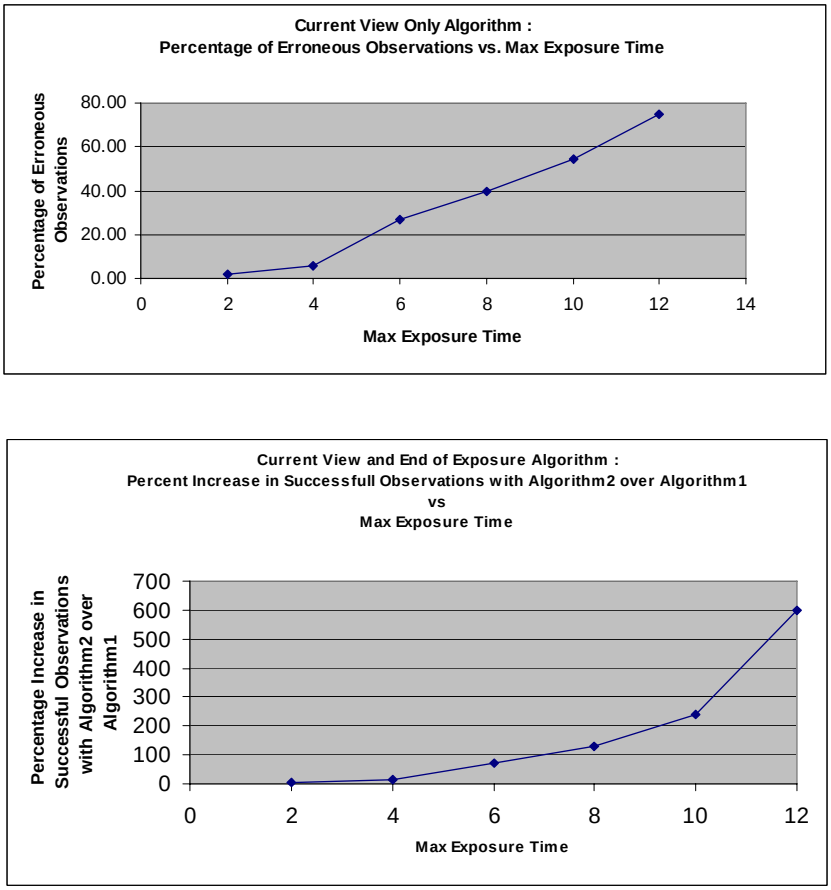

Figure 4.11 Simulation results using two exposure request acceptance algorithms

The same set of observation requests are fed as input to the models for each max exposure time, once for algorithm 1 and a second time for algorithm 2. This will show how many successful observation requests occur given the chosen decision algorithm and allowed maximum exposure time. The first chart in Figure 4.11 shows the percentage of observations that were erroneous as the max exposure time allowed was increased. Since algorithm 1 only checks that the object is currently in view when the request is received, the results start to see more erroneous exposures as the max allowed exposure time increases. The second chart in Figure 4.11 shows the percentage increase in the number

of observations that were successfully carried out using Algorithm 2 versus Algorithm 1. This chart shows that as the allowed exposure time increases there is more benefit in using Algorithm 2. These benefits are due to an increase in the number of long exposure requests that are received as the max exposure time is increased. Overall the results show that the second algorithm is a better choice because it eliminates requests that will be erroneous because the object is going out of view before the end of the exposure.

5. Related Work

5.1. Software modeling of real-time systems

Software modeling primarily deals with specification and implementation of software. Object-oriented modeling techniques allow for characterization of software components and their composition. Approaches such as UML allow specification of the relationships between components in terms of inheritance, aggregation, and realization. In addition, the UML sequence diagram can capture the timing of component interactions in non-real time. However these modeling approaches do not allow for execution of the models under logical time, and thus have limited capabilities for testing and validation.

The design of real time software applications is unique in that the need to include formal timing and concurrency in the software modeling is crucial. Methods such as UML-RT have been introduced to extend the OO modeling to formally account for time. In UML-RT the time, schedule, and performance related properties of the software models can be captured using UML stereotypes, tagged values, and constraints. Models are outfitted with ports that when connected to other models, allow events to be communicated during statechart execution. During this execution the timing related properties can be validated (Huang 2004).

Although UML-RT model execution is useful in testing for defects in timing constraints of software models, it cannot be used until the software modeling reaches a detailed stage. One benefit of the approach presented in this thesis versus UML-RT is that simulations are performed on high level software design concepts, such as design patterns. These patterns are identified early in the design phase, and therefore benefits of the simulation experiments are realized early.

5.2. Software design techniques in simulation

The benefits of using of modern software engineering techniques to build simulation models are now being realized by the simulation community. Researchers that use simulation to study science are utilizing the object oriented capabilities of environments such as DEVSJAVA to design more maintainable and re-usable models. In addition, software design concepts such as design patterns are being adapted for simulation model design. The result is models that are more easily configured and easy to maintain (Innocenti 2004). The use of design patterns in these applications is driven from the desire to make the models more maintainable and re-usable. The research presented in this thesis will also enjoy these benefits, but differs in that the use of patterns is driven more from the domain. This thesis lays out a principled approach for identifying design patterns in the AO domain. Use of the design patterns to develop the simulation models is driven from the desire to simulate those patterns, and thus simulate the high level design of the AO control system. In addition, the design patterns identified with this approach are used beyond simulation modeling, and are also incorporated in the detailed software modeling design.

6. Conclusion

The motivation behind this work was Simulation Based Acquisition (SBA 1998) which promotes systematic use of simulation across lifecycle of systems from conception to retirement. In this respect, the presented approach focuses on supporting simulation-based software design. This work demonstrated the use of design patterns in support of the command and control paradigm for the software development of astronomical observatory control systems using DEVSJAVA-AO. Thus the inclusion of design patterns in a modeling and simulation environment for specific domains plays a significant role in creating software design that can be simulated prior to detailed software design specification, with a key benefit being reduction in the overall software development effort. A future direction for this research is applying the simulation models for developing software controlling an astronomical observatory. Another future research opportunity involves forward engineering from simulation models to software models. A related area of interest is the inclusion of design patterns in real-time simulation modeling.

REFERENCES

ACIMS, *DEVSJAVA*, http://www.acims.arizona.edu, 2003.

Balasubramanian, K., A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema. 2006. "Developing Applications Using Model-Driven Design Environments". *IEEE Computer* (February): 33-40.

Braeside Observatory. 2005. Arizona State University. http://braeside.la.asu.edu.

Briand, L. C., Y. Labiche, and Y. Wang. 2004. "Using Simulation to Empirically Investigate Test Coverage Criteria Based on Statechart". *ICSE*: 86-95.

Chen, Y., G.C. Gannod, J.S. Collofello, and H.S. Sarjoughian. 2004. "Using Simulation to Facilitate the Study of Software Product Line Evolution". *Seventh International Workshop on Principles of Software Evolution* (September): 103-112.

Cox, J.E. 2002. "Simulation as an Integral Component of the Software Architecture Design Process". Masters Thesis. Arizona State University.

Dalal, M.A., M. Erraguntla, and P. Benjamin. 1997. "An Introduction to Using ProSim for Business Process Simulation and Analysis". *Proceedings of the 1997 Winter Simulation Conference*: 718-724.

Dias, M. S. and E. Marlon. 2000. "Software Architecture Analysis Based on Statechart Semantics". *Tenth International Workshop on Software Specification and Design*: 133-137.

Ferayorni, A., Sarjoughian, H., 2007, "Domain Driven Simulation Modeling for Software Design", Summer Computer Simulation Conference, pp. 297-304, San Diego, CA.

Fujimoto, R.M. 2000. *Parallel and Distributed Simulation Systems*. John Wiley and Sons, Inc.

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

Gerla, M., R. Bagrodia, L. Zhang, K. Tang, and L. Wang. 1999. "TCP over Wireless Multihop Protocols: Simulation and Experiments". *Proceedings of IEEE International Conference on Communications* 2: 1089-1094.

Gray, J., Y. Lin, and J. Zhang. 2006. “Automating Change Evolution in Model-Driven Engineering”. *IEEE Computer* (February): 51-58.

Hall, S.B. 2005. “Learning in a Complex Adaptive System for ISR Resource Management”. *Winter Simulation Conference*: 149-158.

Huang, D. and H.S. Sarjoughian. 2004. “Software and Simulation Modeling for Real-time Software-intensive System”. *The 8th IEEE International Symposium on Distributed Simulation and Real Time Applications* (October): 196-203.

IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and rules. 2000. IEEE Std 1516-2000: i-22.

Innocenti, E., A. Muzy, A. Aiello, J. Santucci,, and D. Hill. 2004. “Active-DEVS: a computational model for the simulation of forest fire propagation”. *IEEE International Conference on Systems, Man and Cybernetics* (October): 1857 – 1863.

Jacobson, I. 1992. *Object-Oriented Software Engineering.* Addison Wesley Professional.

OMG, *MDA: Model Driven Architecture*. 2005. http://www.omg.org/mda/.

OMG, *UML: Unified Modeling Language*. 2005. http://www.uml.org/.

OMG, *UML-RT: Unified Modeling Language – Real Time*. 2006. http://www.omg.org/cgi-bin/doc?formal/05-07-04/.

PAVG, *Rapide*. 1998. http://pavg.stanford.edu/rapide/.

Rational Rose RT. 2006. http://www-306.ibm.com/software/awdtools/developer/technical/.

Sarjoughian, H. S. and R.K., Singh. 2004. “Building Simulation Modeling Environments Using Systems Theory and Software Architecture Principles”. *Advanced Simulation Technology Conference* (April): 99-104.

Sarjoughian, H.S. and B.P. Zeigler. 2000. “DEVS and HLA: Complementary Paradigms for Modeling and Simulation”. *Transactions of the Society of Modeling and Simulation International* 17(4): 187-197.

SBA. 1998. “Simulation Based Acquisition: A New Approach”. *Defense Systems Management College, Report of the Military Research Fellows*.

Schmidt, D. 2006. Model-Driven Engineering. *IEEE Computer* (February): 25-31.

SimPy. 2004. http://simpy.sourceforge.net/.

Spin. 2006. http://spinroot.com/spin/whatispin.html.

United States Department of Defense, *HLA*,. 2005. https://www.dmso.mil/public/transition/hla/.

Wymore, W.A. 1993. *Model-based Systems Engineering: An Introduction to the Mathematical Theory of Discrete Systems and to the Tricotyledon Theory of System Design*. Boca Raton: CRC.

Zeigler, B.P., H. Praehofer, and T.G. Kim. 2000. *Theory of Modeling and Simulation.* 2nd ed. Academic Press.

Zeigler, B.P., H.S. Sarjoughian, W. Au, 1997, “Object-Oriented DEVS”, 11th SPIE, pp. 100-111, Apr., Orlando, FL

APPENDIX A

UML USE CASE DIAGRAMS

This appendix provides additional use case diagrams generated during research in the domain of astronomical observatory (AO) control systems. These use cases were gathered through information obtained while studying the Braeside Observatory control system. The approach used to obtain and analyze these use cases is described in Chapter 3 of this thesis.

One of the key components of the observatory control system is the software that controls the telescope focuser. This software is responsible for carrying out requests from the user to adjust the telescope focus. In Figure A.1 the use cases for interaction between the client and the focuser controller are shown.

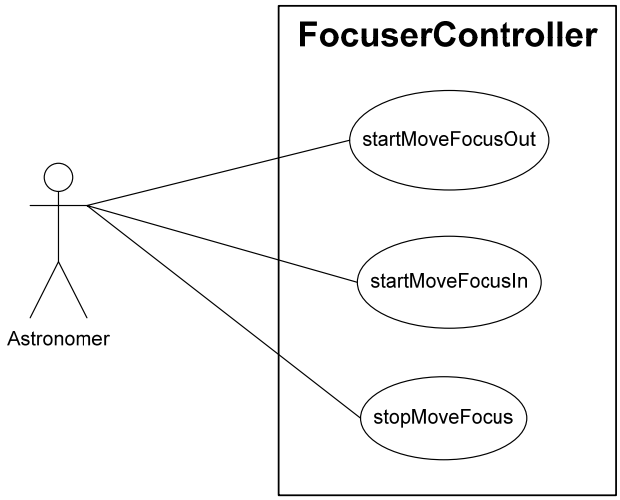

Figure A.1 Use case diagram for focuser controller

Actor / Use Case Descriptions:

Astronomer: Actor that interacts with the system to take observations and collect data.

startMoveFocusOut: Start movement of the focuser outward.

startMoveFocusIn: Start movement of the focuser inward.

stopMoveFocus: Stop movement of the focuser.

The mount controller is responsible for carrying out user requests to move the telescope mount. It also supports the ability to link with detector controllers in order to allow data to be shared, which is useful for applications such as auto-guiding. In Figure A.2 the use cases are shown as well as the actors that typically invoke them.

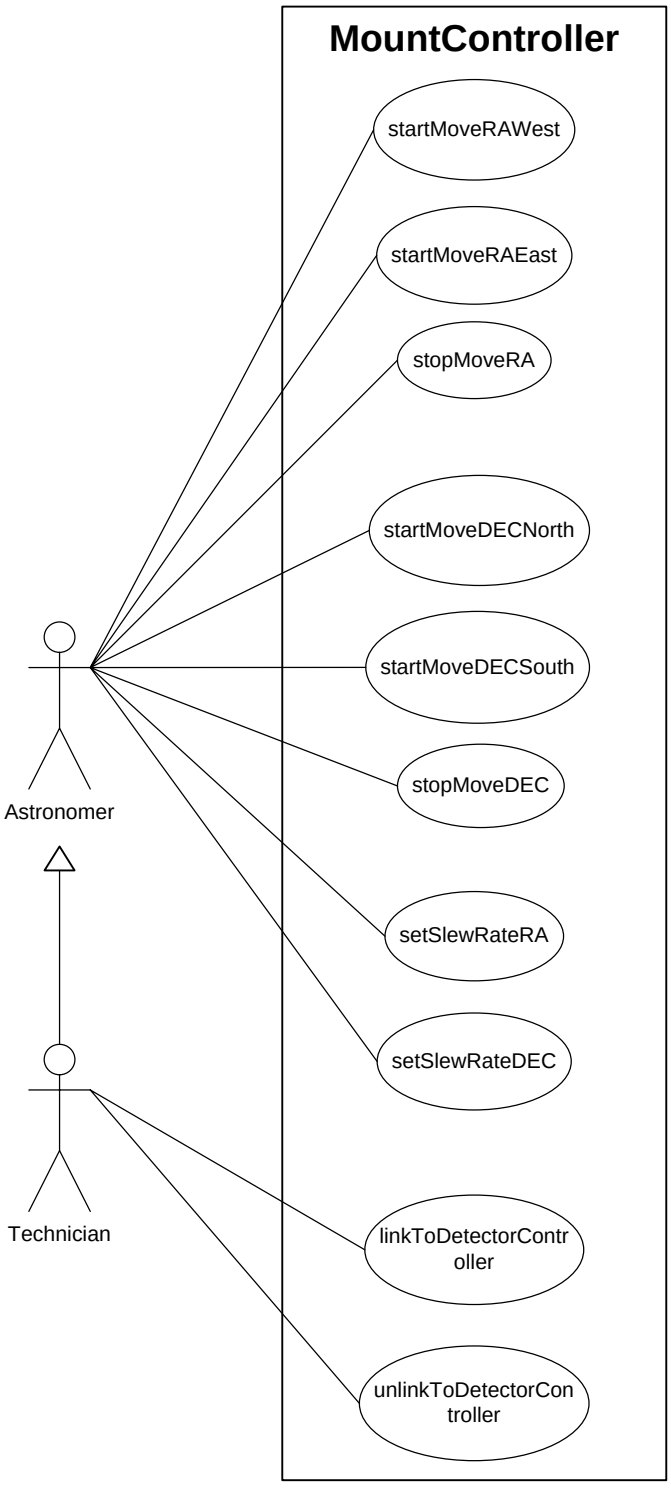

Figure A.2 Use case diagram for mount controller

Actor / Use Case Descriptions:

Astronomer: Actor that interacts with the system to take observations and collect data.

Technitian: Actor that interacts with the system to perform maintenance.

startMoveRAWest: Start movement of the right ascension axis in the West direction.

startMoveRAEast:Start movemenet of the right ascension axis in the East direction.

stopMoveRA: Stop movement of the right ascension axis.

startMoveDECNorth: Start movement of the declination axis in the North direction.

startMoveDECSouth: Start movemenet of the declination axis in the South direction.

stopMoveDEC: Stop movemenet of the declination axis.

setSlewRateRA: Set the slew rate for the right ascension axis to a given value.

setSlewRateDEC: Set the slew rate for the declination axis to a given value.

linkToDetectorController: Link the mount controller to a detector controller.

unlinkToDetectorController: Un-link the mount controller from a detector controller.

APPENDIX B

UML CLASS DIAGRAMS

This appendix provides additional UML class diagrams that capture design patterns used in the DEVSJAVA-AO framework. These patterns were identified using the same approach discussed in Chapter 3 of this thesis.

The mount controller component of an AO control system may contain many sub-components that provide the functionalities shown in the mount controller use case diagram (see Figure A.2). These use cases helped identify the need to create a façade layer that hides the details of how the mount controller functions are carried out. Figure B.1 shows the use of the façade design pattern in the design of a simple AO control system. The *MountControllerInterface* is used to specify which functions will be provided, what parameters they require, but not which software components actually implement them. The *MountController* class realizes this interface, thus de-coupling the client from the implementation details of the façade functions.

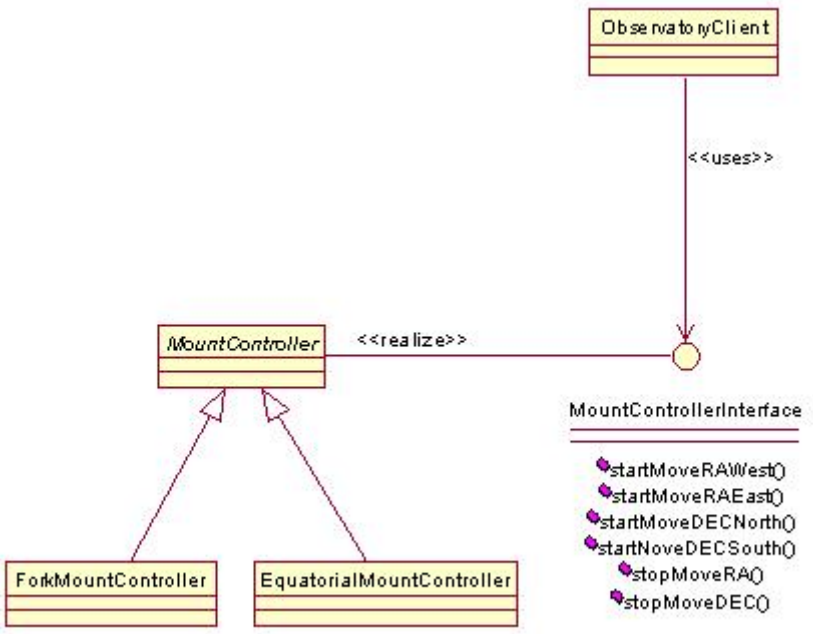

Figure B.1 Facade design pattern for mount controller

The focuser controller is responsible for carrying out all client requests to adjust the focus of the telescope. The use cases in Figure A.1 helped identify the need for a façade layer between the client and the focuser controller. The façade design pattern, shown in Figure B.2, is implemented using the *FocuserControllerInterface*, which must be realized by the sub-components of the *FocuserController*. This façade layer helps to de-couple the client from the implementation details of the focuser controller functions.

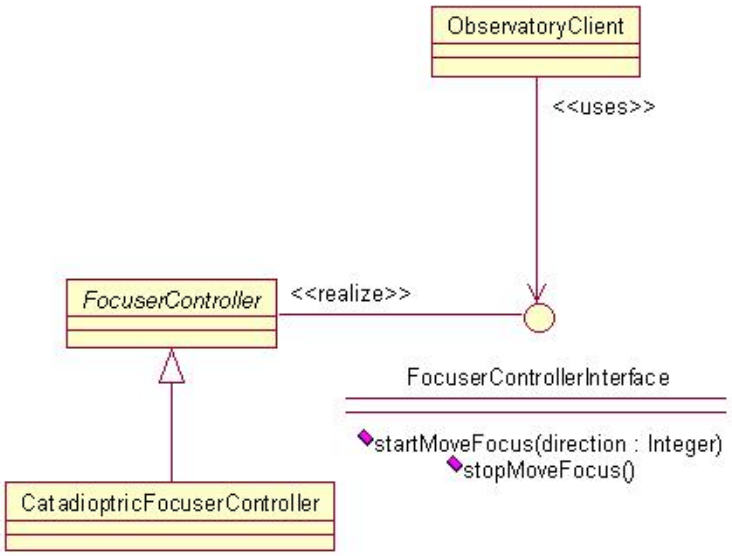

Figure B.2 Facade design patterns for focuser controller