DESIGN OF AN INTEGRATED ROBOT SIMULATOR FOR
LEARNING APPLICATIONS

by

Brendon J. Wilson

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
BACHELOR OF APPLIED SCIENCE
In The School

of

Engineering Science

© Brendon Wilson 1999
SIMON FRASER UNIVERSITY
January 1, 1999

All rights reserved. This work may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.
Approval

Name: Brendon Wilson
Degree: Bachelor of Applied Science
Title of Thesis: Design of an integrated robot simulator for learning applications

Dr. John Jones
Director
School of Engineering Science, SFU

Examiner Committee:
Technical and Academic Supervisor:

Dr. John Dill
Professor
School of Engineering Science, SFU

Committee Member:

Dr. Kamal Gupta
Associate Professor
School of Engineering Science, SFU

Committee Member:

Dr. Brian Fisher
Research Associate
Center for Systems Science, SFU

Date Approved
Abstract

In response to rising equipment costs and reduced funding, the REMOTE (Really Exciting Manipulator Object Tele-learning Experience) project is investigating new methods of providing hands-on experience to students using tele-learning. Tele-learning is a method of providing distance education using multimedia technologies via any combination of computer, telephone, video-conferencing, or Internet connections. The purpose of this project is to create a Java application that enables students to explore robotics programming via interactive experimentation.

This thesis extends work by Ron Racine and Scott Branden, and develops one half of the REMOTE project: a simulation application for a simple, open-chain robot. The completed simulator environment features a graphical user interface that displays visual simulation feedback and allows users to edit, program, and debug simulation files. The application’s underlying simulation engine controls the internal representation and simulation of the robot manipulator model. Future additions to the simulator will allow students to upload completed simulation programs to a remote robot, and view the results via the Internet.

This project is of particular importance to future ENSC 489 classes, which will use the simulator to provide students with more access to robot programming experience at a lower cost. To suit the purposes of ENSC 489, the software is required to simulate a Scorbot ER-III robot; however, by careful design and implementation, the completed application permits simulation of other similar robots, allowing the application to suit the present and future demands of ENSC 489.
Acknowledgements

I would like to thank Sun Microsystems, and James Gosling in particular, for creating the Java language and making its reference implementations freely available to developers. Without Sun’s enormous effort and generosity, I doubt the computing world would ever be able to realize the idea of “write once, run anywhere” software, unless of course we were all programming for Microsoft Windows, and only Microsoft Windows. I would also like to thank Ken Tallman of the Java3D group, and Amy Fowler of the JFC/Swing group for their quick responses to bug reports, and requests for information.

I would like to thank the members of the Advanced-Java mail list hosted by the Experimental Computing Facility at Berkeley, and the Java 3D User Group at Sun Microsystems for their valuable and insightful advice. These discussion forums not only offered advice on methods of overcoming various design problems, but also alerted me to the existence of bugs that could have prevented me from succeeding in my efforts.

Finally, I would like to the members of my thesis committee for the time they have taken to read, review, and comment this document, allowing me to hone it into a suitable form for presentation as a thesis document. I would especially like to thank my technical supervisor, Dr. John Dill, for his valuable input and guidance during the creation of the REMOTE simulator.
# Table of Contents

**APPROVAL** .......................................................................................................................................................... II  
**ABSTRACT** ........................................................................................................................................................... III  
**ACKNOWLEDGEMENTS** ........................................................................................................................................ IV  
**LIST OF FIGURES** ................................................................................................................................................ VI  
**INTRODUCTION** ..................................................................................................................................................... 1  
**DEVELOPMENT TOOLS AND TECHNIQUES** .................................................................................................. 4  
  - JAVA ....................................................................................................................................................................... 4  
  - DESIGN PATTERNS .............................................................................................................................................. 6  
  - UNIFIED MODELING LANGUAGE ........................................................................................................................ 7  
**THE SIMULATION ENGINE** ................................................................................................................................... 10  
  - CREATING INTERCHANGEABLE FILE PARSERS ............................................................................................... 12  
  - ENCAPSULATING SIMULATION COMMANDS ................................................................................................. 16  
  - ENCAPSULATING THE ROBOT MODEL .............................................................................................................. 19  
    - Encapsulating the Robot Inverse Kinematics ................................................................................................. 21  
    - Encapsulating the Movement Interpolation Algorithm ............................................................................... 22  
  - THE COMPLETED SIMULATION ENGINE ARCHITECTURE .......................................................................... 24  
**THE GRAPHICAL USER INTERFACE** ........................................................................................................................................ 25  
  - OVERVIEW OF SIMULATOR USER INTERFACE ............................................................................................ 25  
  - USER INTERFACE - SIMULATION ENGINE COMMUNICATION ................................................................ 27  
**THE FUTURE OF REMOTE** ..................................................................................................................................... 29  
  - GENERAL INVERSE KINEMATICS CAPABILITIES ........................................................................................... 29  
  - EXPANDED 3D MODEL SUPPORT ..................................................................................................................... 29  
  - IMPROVED USER INTERFACE LIBRARIES ...................................................................................................... 30  
  - CONTEXT-SENSITIVE HELP ............................................................................................................................ 30  
  - MULTIPLE ROBOT SIMULATION SUPPORT ................................................................................................. 31  
  - REMOTE ROBOT SERVER ............................................................................................................................... 31  
**SUMMARY** ............................................................................................................................................................. 32  
**APPENDIX A: REMOTE ROBOT CONFIGURATION SYNTAX** .................................................................................. 33  
**APPENDIX B: REMOTE SIMULATION LANGUAGE** .............................................................................................. 35  
**APPENDIX C: RUNNING THE EXAMPLE SIMULATION** ............................................................................................ 36  
**APPENDIX D: DENAVIT-HARTENBERG PARAMETERS** ............................................................................................. 37  
**REFERENCES** .......................................................................................................................................................... 38
List of Figures

FIGURE 1: REMOTE SIMULATOR USER INTERFACE .................................................................2
FIGURE 2: JAVA PROGRAM DEVELOPMENT AND EXECUTION CYCLE..................................5
FIGURE 3: UML INPUTS ..........................................................................................................8
FIGURE 4: UML NOTATION FOR CLASS RELATIONSHIPS ..................................................8
FIGURE 5: UML CLASS ANNOTATION ....................................................................................9
FIGURE 6: SIMULATION ENGINE SEQUENCE DIAGRAM ..................................................10
FIGURE 7: THE STRATEGY PATTERN (GAMMA ET AL., 1994, 87) ....................................13
FIGURE 8: DESIGN OF PARSER FOR 3D MODELS ..............................................................16
FIGURE 9: THE COMMAND PATTERN (GAMMA ET AL., 1994, 236) .................................17
FIGURE 10: THE MACROCOMMAND IMPLEMENTATION (GAMMA ET AL., 1994, 235) .......18
FIGURE 11: COMMAND QUEUE TRANSITIONS (GAMMA ET AL., 1994, 63) ....................19
FIGURE 12: THE SINGLETON PATTERN ...............................................................................20
FIGURE 13: ROBOT CLASS DESIGN AND RELATIONSHIPS ..............................................21
FIGURE 14: THE INVKin INTERFACE AND IMPLEMENTATION ........................................22
FIGURE 15: THE INTERPOLATOR INTERFACE .....................................................................23
FIGURE 16: COMPLETE SIMULATION ENGINE ARCHITECTURE ........................................24
FIGURE 17: THE REMOTE USER INTERFACE OVERVIEW ...............................................25
FIGURE 18: TOOL-TIP FOR THE PASTE TOOLBAR BUTTON .............................................26
FIGURE 19: THE OBSERVER PATTERN (GAMMA ET AL., 1994, 294) ...............................27
FIGURE 20: HEAVYWEIGHT-LIGHTWEIGHT COMPONENT COLLISION ...............................30
FIGURE 21: DENAVIT-HARTENBERG DETAIL (CRAIG, 1989, 67) .......................................37
Introduction

The process of educating students in new and existing technology is an expensive undertaking, requiring investments in machinery, maintenance, training, and facilities. Ultimately, the advance of technology renders this investment obsolete, demanding the replacement of complete systems in order to stay current; any prior investment is effectively wasted. In Engineering Science, the rapid advance in engineering technology combined with increasing class sizes make it difficult to provide adequate access to the latest technology; even if the latest technology is purchased, it may not be enough to allow equal access to all students.

One possible solution to this problem is to provide students with a mechanism to perform experiments “virtually”, through software simulations and/or remotely accessible experiments. Software simulation has the advantage that software can be designed to be configurable: one piece of well-designed simulation software could provide training for several different pieces of related equipment, such as different types of CAD/CAM machinery. Remote experimentation has the advantage that it provides students with greater access to available resources, and allows students to work with the actual equipment, rather than just a simulation.

The REMOTE project, headed by Dr. John Dill, is currently combining both software simulation and remote experimentation into a single solution to the problem of providing access to equipment for his ENSC 489 course. The purpose of the REMOTE project is to create an application which students can use to simulate programming a SCORBOT ER-III robot. Once students are satisfied with their complete programs, the application will allow them to upload the completed simulation program to the real manipulator via the Internet, and watch the execution of their program using streaming Internet video technology.

This thesis develops one half of the REMOTE project: a simulation application for a simple, open-chain robot. The completed application, shown in Figure 1, allows users to
create and edit files containing commands to control the simulated robot, and run a simulation. Running a simulation updates both the 3-dimensional representation of the robot’s motion, and the display of the robot’s joint positions in accordance with the user’s program.

![Remote Simulator User Interface](image)

**Figure 1: Remote Simulator User Interface**

Although the REMOTE project only requires a simulator for the SCORBOT ER-III manipulator, the final simulation application is flexible enough that it can be reconfigured to simulate other similar robots. The configuration details for the robot manipulator itself are contained in external configuration files; in the future, a new robot manipulator can be simulated by simply creating a new set of configuration files, without requiring changes to the simulator’s code.

The bulk of this thesis describes the design of a simple, flexible architecture for the simulation of an open-chain robot. This simulation architecture provides developers with
the opportunity to easily extend the current simulation engine’s abilities, without requiring modifications to the simulation engine code itself. In the event that developers need to add substantial new abilities to the simulator, the simple design of the simulation engine’s architecture will make it easy to modify the simulation engine to incorporate these new abilities.

Simpler features, such as new simulation language parsers or 3D model format parsers can be added to the simulation architecture without requiring alterations to the existing code. In the future, the flexibility of this architecture will allow developers to add more complex features, such as simulation of multiple robots, with minimal additions to the existing simulation engine. In addition to the task of robot simulation, the architecture will allow a future developer to tie the simulation to the control of a remote robot to enable interactive experimentation.

The creation of the simulator required consideration of not only implementation details, but also design details that would simplify the implementation. Thought had to be given to both the design techniques used to create the simulation architecture, as well as the development tools that would be used to implement the architecture. The following section introduces the tools and design techniques employed by the REMOTE simulator.
Development Tools and Techniques

In order to meet the design and implementation demands of the project, the REMOTE simulator makes use of several key technologies: design patterns, UML, human-computer interface theory, robotics theory, and Java. Design patterns and UML provided the basis for creating and documenting the simulation architecture, while robotics provided the modeling theory required to simulate a robot’s joints, position, and motion. Human-computer interface theory provided the background needed to create a usable interface for the simulation application. The final required component, Java, provided the implementation language for the simulation application.

Human-computer interface and robotics theory are both well-documented fields, and will not be addressed in this thesis; for further information on these fields, see *Tog on Interface* (Tognazzini, 1991) and *Introduction to Robotics* (Craig, 1989). The remainder of this section introduces each of the development technologies, outlines their history, and their impact on the REMOTE project.

**Java**

In the early 90’s, Sun Microsystems created the Oak programming language to provide a way to write consumer electronics applications for any device. One of the main goals of the project was to allow consumer electronics developers to write platform-independent applications, eliminating the cost of re-writing applications for new devices. The proof-of-concept for Oak, an interactive TV that allowed users to program their VCR using a simple touch screen, was deemed a failure; although the device acted as a convincing proof-of-concept, the project was disbanded.

It was not until the explosion of the Internet in 1994 that Sun co-founder Bill Joy rescued the project, with the realization that the newly renamed Java language could be used to provide additional interactive duties within a web browser. You could build applications that could run anywhere, even from thin-clients that downloaded their applications from the network; this realization renewed Sun’s development efforts, and refocused the
company’s thinking, as reflected in Sun’s marketing motto “the network is the computer”.

Although most people associate Java with the Internet and web browsers, Java is much more than a means to produce interactive web content. It is a fully functional object-oriented language, described by Sun Microsystems as:

A simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multi-threaded, and dynamic language (Flanagan, 1997, 3).

Java is different from most programming languages in that the code produced by the compiler is not executable directly on a specific platform. As shown in Figure 2, a Java compiler translates source code into an intermediate format, called byte-code, which can be executed by a Java virtual machine; the Java virtual machine is an abstract computer that understands Java byte-code, and interprets it into platform-specific calls at run-time. The format of the byte-code is cross-platform, therefore the only thing that needs to be specific to a particular platform is the implementation of the Java virtual machine.

![Figure 2: Java Program Development and Execution Cycle](image)

The original version of the REMOTE simulator (Branden and Racine, 1995) used the Java language to take advantage of the language’s cross-platform “write once, run anywhere” capability. The portability of Java ensured that the effort expended in creating the REMOTE would not be wasted in the event that the application had to run on another platform.

By continuing to use Java to create a new version of REMOTE, this project leverages the existing code, and takes advantage of new libraries to extend the capabilities of the
application. In particular, the addition of 3-dimensional rendering libraries (Java3D) and improved user interface libraries (Java Foundation Classes) by Sun has contributed to a significantly improved user interface. The continued popularity of Java, the increasing quality and availability of Java development tools, and the recent standardization of the Java language under ISO ensure that the effort spent creating and improving REMOTE will provide a solid foundation to fulfill the future needs of ENSC 489.

However, in order to run the REMOTE application users are required to install the Java Runtime Engine, which allows a specific platform to run Java applications; this is a disadvantage due to the size of the runtime engine, and the potential installation difficulties a user may encounter installing the runtime. In the future, it is anticipated that operating systems will incorporate a native Java interpreter, removing this disadvantage; for example, the Macintosh OS has already incorporated Java technology, allowing Java programs to be run in the same fashion as native applications.

**Design Patterns**

Design patterns encapsulate a particular solution to a general programming problem, assigning a name to the solution, a description of the problem, a solution to the problem, and consequences of the design that should be considered before employing the design pattern. Design patterns act as high-level architectural descriptions, detailing objects and their interactions; you could think of them as the high-level object-oriented algorithms, except that their purpose is to detail architectures, rather than implementations.

A simple definition of design patterns is provided by the authoritative reference on design patterns, "Design Patterns: Elements of Reusable Object-Oriented Software": design patterns are "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" (Gamma et al., 1994, 3). Ironically, design patterns did not originate in computing at all; instead, design patterns began in the realm of architecture, addressing common issues of building design and construction.
Software design patterns can be placed into three categories:

Patterns can have either creational, structural, or behavioral purpose. Creational patterns deal with the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility (Gamma et al., 1994, 10).

Much of the flexibility of the REMOTE simulator is due to its ability to dynamically use different components to handle reading 3D model files, simulation files, and configuration files. These capabilities are built on top creational and behavioral design patterns that address the problem of separating implementation details from architecture details; by separating the simulation engine architecture from the implementation details, the simulation engine remains as general as possible.

By using design patterns to solve common object-oriented problems, REMOTE benefits from generations of iterative design by literally millions of programmers. Not only can the patterns provide solutions that are “ready to go”; design patterns document the advantages, disadvantages and consequences of a particular design. Less-than-obvious abstractions discovered and refined by other programmers into design patterns improve both the flexibility and maintainability of the application architecture.

**Unified Modeling Language**

In order to record the specific architectures presented by object-oriented designs, several notation standards have emerged over the years, including methods created by Rumbaugh, Booch, and Jacobson. Each of these standards had their specialities, advantages and disadvantages, and individual notation standards, all of which were succeeded by a new standard, the Unified Modeling Language, or UML, which combines the strengths of these and other notation standards. The influences on the formation of UML are given in Figure 3:
UML distills designs and implementations into a visual notation:

UML is a language used to specify, visualize, and document the artifacts of an object–oriented system under development. It represents the unification of the Booch, OMT, and Objectory notations, as well as the best ideas from a number of other methodologies as shown in [Figure 3]. By unifying the notations used by these object-oriented methods, the Unified Modeling Language provides the *de facto* standard in the domain of object-oriented analysis and design founded on a wide base of user experience (Quatrani, 1998, 6).

For the purpose of this thesis, UML will be used only for its ability to help the reader visualize the implementation of the design patterns used in the REMOTE application. The UML notation is capable of describing traditional object-oriented programming concepts, such as inheritance, and relationships between classes. Figure 4 shows the graphical notation used by UML to annotate class relationships.
The notation in Figure 4 shows the classic object-oriented relationships between classes and objects, including (from left to right) a subclass inheriting from a superclass, a class implementing an interface, association, aggregation, and instantiation or dependency. UML can also annotate additional class features, such as public and private operations and fields as shown in Figure 5.

![Figure 5: UML Class Annotation](image)

As shown in Figure 5, UML can annotate public attributes (variables) and operations (methods), available to any external classes, protected attributes and operations, available to only the class and its subclasses, and private attributes and operations, available only to the class itself.

Using UML to annotate the design of the simulation engine not only helps document the design for future developers, but also helps find additional opportunities for the application of design patterns.
The Simulation Engine

The simulation engine is the core of the REMOTE application, responsible for coordinating components used to create the internal model of the robot, parse simulation programs into sequences of commands to operate on the robot, and manipulate the model using the resulting commands. Note that the simulation engine is only responsible for coordinating these activities; the simulation engine is neither responsible for the specific implementation details required to achieve each of these tasks, nor should it be required to know these implementation specific details. To perform the simulation, the simulation engine will need to perform the steps shown in the sequence diagram in Figure 6, and distribute the responsibility for the various steps across several classes.

Figure 6: Simulation Engine Sequence Diagram
Each step roughly corresponds to delegating an activity to another object which encapsulates the details of how to perform the requested activity; these details are hidden from the simulation engine, so it only has to know how to request services from an object, not how the services are implemented. The ‘Remote’ class only needs to know how to tell the ‘SimulationEngine’ class to load the simulation commands, and start the simulation process; similarly, the ‘SimulationEngine’ class knows how to iterate through the queue of commands, and instruct each ‘Command’ to execute itself.

The extraction and encapsulation of implementation details outside of the simulation engine reduces the dependency of the simulation engine on the implementation of components of the simulation infrastructure, such as the parsers to load the robot model, or transform the user’s program into operations on the robot model. This guideline of “programming to an interface, not an implementation” (Gamma et al., 1994, 18) results in two benefits:

1. Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.
2. Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class(es) defining the interface (Gamma et al., 1994, 18).

In addition to addressing the problem of shielding the simulation engine from implementation details, the design of the simulation engine also addresses the creation of objects. For example, although programming to an interface allows the simulation engine to deal with parsers for simulation languages independent of the particular implementation of the interface, how does that particular implementation get instantiated? The simulation engine uses creational design patterns to abstract the process of object creation for components outside of the simulation engine’s implementation.

To perform the steps shown in Figure 6, the simulation engine requires the following components:

- File Parsers: to interpret robot simulation languages, robot configuration details, and physical robot models.
- Robot Model: to represent the robot’s current state (e.g. joint angles) and provide operations to manipulate the model. This includes modeling the robot’s physical geometry and kinematic behavior.
- A mechanism to coordinate the use of file parsers, in order to pull in configuration information, construct the robot model, and perform the simulation according to a user’s program.

The following sections outline each of the components outlined above, the design patterns used to create them, and how the components fit into the overall simulation architecture.

**Creating Interchangeable File Parsers**

Part of the flexibility of the REMOTE simulator includes the ability to provide alternate parsers to read a variety of 3D model formats, robot configuration formats, and robot simulation languages. These parsers handle creating and configuring the internal robot model, as well as manipulating the model according to the user’s simulation program. In order to enable developers to create additional parsers, without requiring re-compilation of the application, the simulation engine requires:

1. A mechanism to allow the simulation engine to handle a parser object independent of its concrete class.
2. A mechanism to allow the simulation engine to create an instance of an appropriate parser without knowing which parser implementation is being instantiated.

For example, the simulation engine requires a parser type that it can use to transform the simulation program into commands for the robot model. There could be several different implementations of this parser, required to deal with several different robot programming languages, but the simulation engine shouldn’t have to know which implementation it is using. In addition, the simulation engine shouldn’t even have to know which parser implementation to create, leaving the details of how to create the parser implementation in the hands of a mechanism external to the simulation engine.
Solving the first problem requires parser implementations to use a common interface, which means that parsers need to provide a set of common methods the simulation engine can use to parse simulation information. By providing a common interface to the simulation engine, the parsers implement the Strategy design pattern, which allows the architecture to:

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it (Gamma et al., 1994, 87).

What this means is that the simulation engine is capable of using a parser object to accomplish a desired task, without having to know which concrete implementation of the parser interface is being used, as shown in Figure 7.

![Figure 7: The Strategy Pattern (Gamma et al., 1994, 87)](image)

A particular client (Context) holds onto an object (ConcreteStrategyA, or B, or C), which provides a concrete implementation of the algorithm’s interface (Strategy). In order for the client to invoke the algorithm, it uses the methods defined by the Strategy interface to the specific algorithm implemented by the concrete class. By analogy, using the Strategy pattern is similar to the use of device independent graphic interfaces; an application only needs to know how to ‘talk’ to the graphics interface, not how to talk to a particular type of graphics device (such as a printer, or graphics card).

Providing a uniform method of accessing algorithms using the Strategy pattern solves the first problem: allowing the simulation engine to use objects without knowing which
concrete implementation is being used. However, using the Strategy pattern alone is not enough to allow the simulation engine to be completely ignorant of the specific concrete implementation of the Strategy interface. The simulation engine still needs to be able to create an instance of a specific concrete implementation; therefore, a mechanism to abstract the object creation process is required to de-couple the simulation engine completely from concrete parser implementations.

Allowing the simulation engine to create instances of the concrete parser classes without knowing the specific class to use required use of the dynamic instantiation feature of the Java language. The desire here is to allow an external configuration mechanism to control which parser the simulation engine will use for various operations, without requiring this configuration to be specified at the code level. Using this configuration mechanism, in this case a text file, a developer will be able to replace complete portions of the simulation engine’s peripheral functionality non-invasively.

Normally, a Java program creates a new object instance of a class directly using the `new` operator, for example:

```java
MyParser foo = new MyParser();
```

This piece of code specifies a variable of type `MyParser`, and gives it the name `foo`. However, the same effect could be achieved by using the dynamic instantiation capabilities of Java, using:

```java
Class aClass = Class.forName("MyParser");
Object foo = aClass.newInstance();
```

Now a string specifies the “MyParser” class should be instantiated, and an instance is created; a configuration text file read at initialization could provide the “MyParser” string.
The only problem here is that foo is now an Object, and the operations of the MyParser class are not exposed; casting the foo object to be an object of type MyParser would defeat the purpose in employing dynamic instantiation. However, by combining this solution with the use of the Strategy pattern:

```java
    Class aClass = Class.forName("MyParser");
    Parser foo = (Parser) aClass.newInstance();
```

If the Parser interface defines all the operations of a parser, the program can cast the dynamically loaded object to a Parser type; MyParser is just one concrete implementation of the Parser interface. This solution allows the use of the object independent of its concrete class, and also abstracts the creation process to the point that a developer can replace the functionality that the MyParser class provides. All a developer needs to do is create another class that implements the Parser interface, and change the name of the class in the configuration text file. Using the combined techniques of the Strategy pattern, and dynamic instantiation, three major sections of the simulation engine can be readily replaced without modifying the simulation engine code: the robot configuration parser, the 3D model parser, and the simulation language parser.

The configuration parser is responsible for reading a file which details the robot’s configuration details, such as the length of links, their relative orientation, and the location 3D model files for each of the robot’s links. The 3D model parser is responsible for parsing files for 3D geometry and material data used to represent the robot’s current physical state in the user interface. Both of these parser interfaces provide the simulation engine with a set of common operations it can use to gain information about the robot to be simulated; for example, the 3D model parser interface, ModelParser, and its implementation, DefaultModelParser, are shown in Figure 8.
The final parser, the simulation language parser, is responsible for parsing the user’s program and providing the simulation engine with a list of commands that it will use to run the simulation. In addition to the Strategy pattern, and dynamic instantiation techniques previously described, additional patterns will be required to de-couple the simulation engine from the representation of individual commands. In short, the simulation language parser will need to produce a set of command objects that the simulation engine can use to update the simulation, without requiring any knowledge about the type of any specific command object. The following section will detail the pattern and techniques used to encapsulate the concept of a command to the simulation engine.

**Encapsulating Simulation Commands**

It is conceivable that developers may want to create simulation language parsers for robot languages that support features not written into the current simulation framework. For example, a particular language may support the ability to define macros, which collect several simulation commands and associate them with a single call to a named macro. In order to allow developers to add these features in coordination with the creation of their new language parser, the simulation framework needs an extensible way to encapsulate commands, and allow developers to add new command classes without revising and re-compiling the simulation engine. This encapsulation not only maximizes the flexibility
of the simulation engine framework, it simplifies the responsibilities of the simulation engine.

To address the requirements outlined above, the simulation engine and simulation command parser use the Command pattern, whose purpose is to:

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support [“undo”-able] operations. (Gamma et al., 1994, 233)

The Command pattern, shown in Figure 9, creates a general structure to encapsulate simulation commands as objects, which the simulation engine can use to manipulate the internal model of the robot.

As Figure 9 shows, the Command pattern simplifies the Invoker object by extracting the knowledge required to execute a particular command, and representing it in an object; the Invoker only needs to know about the Command interface, not the implementation details required to perform a particular action. In order to execute a command, the Invoker only needs to call `Execute()`, and the concrete implementation of the command takes care of the implementation details associated with a particular command by sending messages to the Receiver object. For example, a command may instruct the robot to move a particular joint to a specified joint angle.

Should a developer wish to add a new command to the framework, no alterations need to be made to the Invoker class; only a concrete implementation of the Command interface
needs to be created. For example, if a developer wished to add a macro command, which would group several Command objects together, only a new MacroCommand class would need to be created; this new concrete implementation is shown in Figure 10.

![Diagram of Command and MacroCommand classes](image)

**Figure 10: The MacroCommand Implementation (Gamma et al., 1994, 235)**

In the case of the simulator, the simulation engine acts as the Invoker, and invokes concrete implementation via the methods defined by the Command interface; the Receiver is the object containing the internal model of the robot. To suit REMOTE, the application uses the following Command concrete classes:

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>HomeCommand</td>
<td>Move the robot to the ‘home’ position.</td>
</tr>
<tr>
<td>MoveXYZCommand</td>
<td>Moves the end effector to a specific endpoint in x, y, z.</td>
</tr>
<tr>
<td>MoveJointCommand</td>
<td>Moves a specific joint to a specified joint position.</td>
</tr>
<tr>
<td>DefineNamedXYZCommand</td>
<td>Defines a name for specific point in x, y, z space.</td>
</tr>
<tr>
<td>GotoNamedXYZCommand</td>
<td>Moves the end effector to the named x, y, z endpoint.</td>
</tr>
<tr>
<td>DefinePositionCommand</td>
<td>Defines a name for the current robot position.</td>
</tr>
<tr>
<td>GotoPositionCommand</td>
<td>Moves the robot to a defined position.</td>
</tr>
<tr>
<td>SpeedCommand</td>
<td>Changes the speed of the robot’s movements.</td>
</tr>
<tr>
<td>WaitCommand</td>
<td>Pauses the simulation for a specified time.</td>
</tr>
</tbody>
</table>

The details of the command’s implementation are hidden from the simulation engine, simplifying the process of simulation and allowing developers the freedom to create new concrete commands, without interfering with the simulation engine code.

Another advantage of implementing the Command pattern is the ability to store commands in an execution queue; by having the simulation command parser return a queue of command objects, commands can easily be executed in sequence by repeatedly
calling `Execute()` on the next command in the queue. Alternatively, the Command interface could be enhanced to include an ‘unexecute’ feature, which would provide the inverse operation for a particular command; the Invoker would gain the ability to “undo” commands by calling `Unexecute()` on the previous command in the queue.

![Figure 11: Command Queue Transistions (Gamma et al., 1994, 63)](image)

For the REMOTE simulation, only the execution functionality, and not the “undo” functionality has been implemented; it was not an appropriate feature to add in the current simulator implementation.

**Encapsulating the Robot Model**

The purpose of the Robot class is to encapsulate the current state of the robot model, and provide operations to manipulate the robot’s state; for example, the Robot class encapsulates the link information, such as link length, and operation to move the link to a specific joint angle. In addition, the Robot class provides global access to the robot model, allowing any object to be able to get the Robot object for two reasons. First, the user interface needs to be able to retrieve the 3D models and joint position information in order to render the current state of the robot. Second, the commands being executed by the simulation engine need to be able to access the Robot class in order to manipulate the robot model.

Although objects requiring the robot state information could retain a reference to the Robot object, initializing each such object with a Robot reference incurs a lot of code overhead; in addition, there remains the problem of ensuring that all references are
pointing to the same Robot object. If, for example, the application needed to be able to dynamically load a new Robot configuration, all components referring to the Robot object would need to be refreshed somehow, increasing the complexity of the architecture.

A simpler alternative is to make the Robot object globally available, so that all components can access the correct instance of the Robot object, without having to hold onto a reference to the object. This is the purpose of the Singleton pattern, which:

Ensure[s] a class only has one instance, and provides a global point of access to it (Gamma et al., 1994, 127).

The Singleton pattern restricts the creation of an object via its constructor, and instead provides a static method which clients use to access the instance of the Singleton class. The Singleton class retains a sole static instance of itself, as shown in Figure 12.

![Figure 12: The Singleton Pattern](image)

As an added bonus, the Singleton patterns can be modified to retain references to a number of instances, rather than a single instance, allowing clients access to a particular instance. In the future, this small change may become useful to allow future versions of REMOTE to simulate multiple robots at the same time, with minimal changes to the architecture. Using the Singleton pattern, the Robot class takes shape, as shown in Figure 13, relying on other parsers to provide the data required to build the internal model of the robot.
The Robot class uses an instance of the JointVariables class, which groups the physical joints of the robot, and the degrees of freedom for each joint. In addition to encapsulating joint and degree information, the Robot class also provides operations to manipulate the state of the robot model, through movements of joints and degrees, and movements to specific points in x, y, z space.

The Robot class also relies on two auxiliary classes, specified in the robot’s configuration file, to provide additional capabilities: the InvKin and Interpolator interfaces. Concrete implementations of the InvKin interface provide the Robot class with the ability to perform inverse kinematics calculations for the specific robot being simulated. Similarly, concrete implementations of the Interpolator interface allow the Robot to increment the robot’s joints between two positions using a specific interpolation algorithm. The following sections expand on the InvKin and Interpolator interfaces.

**Encapsulating the Robot Inverse Kinematics**

In order to allow the simulation to move the robot effector to a specific x, y, z endpoint, the Robot class must be able to calculate the corresponding joint positions using inverse kinematics. Inverse kinematics calculations vary according to the specific layout of the
robot’s linkages; although these calculations can be solved numerically, this task is beyond the current scope of the current simulator.

To simplify the design of the robot, a class to encapsulate these calculations needs to be created. In the same way that the file parsers for the simulation engine used the Strategy pattern to de-couple the implementation of file parsers from the simulation engine, the Strategy pattern is used again to allow the Robot class to use a different inverse kinematics algorithm for different robots without modifying the Robot class. As shown in Figure 14, the Scorpion-specific implementation of the InvKin interface provides a method to get the required set of joint positions given a point in 3D space.

![Figure 14: The InvKin Interface and Implementation](image)

The disadvantage of using a concrete class to provide inverse kinematics capabilities to the simulation engine is that a new class must be developed for each robot with a different link structure (i.e.: 4 link vs. 3 link, prismatic vs. revolute joints). For the short term, this shortcoming is acceptable given that the primary purpose here is to develop a Scorpion ER-III simulator. In the future, there’s nothing to prevent the replacement of the robot-specific inverse kinematics classes with a general inverse kinematics engine; in fact, putting a general inverse kinematics engine in place would require no changes to the existing simulation engine code.

**Encapsulating the Movement Interpolation Algorithm**

In addition to requiring inverse kinematics capabilities, the Robot class also needs an algorithm to move the robot’s joints between two sets of joint positions: one associated
with the current position, the other associated with the destination position. In order to achieve this interpolation movement between positions, the Robot could use one of several interpolation algorithms; however, to maintain flexibility, it should ideally be able to use any algorithm. Again, the Strategy pattern is used to separate the algorithm’s interface from its implementation.

The Interpolator interface, shown in Figure 15, defines the method required by the Robot to allow it to move the model between two positions.

![Figure 15: The Interpolator Interface](image)

Note that in order to allow the Interpolator concrete classes to run in a similar fashion to Threads, the Interpolator implements the java.lang.Runnable interface, which defines the operations for the Thread class in Java. The implementation of an Interpolator concrete class is now quite simple: when the Robot is required to execute a particular movement, it generates a set of destination joint positions and provides them to an Interpolator, which then moves the Robot from its current joint positions to the destination positions incrementally.

Although currently the interpolation algorithm is set in the robot’s configuration file, this feature could be made accessible via the user interface to allow users to change the interpolation method “on-the-fly”.

The Completed Simulation Engine Architecture

In total, the simulation engine only requires the key classes and interfaces shown in Figure 16 to define the process of performing a simulation; concrete implementations of the interfaces make up the remainder of the simulation code. Hence, the simulation engine could conceivably be changed to use a completely different simulation language, configuration language, 3D-model format, and interpolation algorithm by providing new implementations for these classes and changing the robot configuration file.

Figure 16: Complete Simulation Engine Architecture

In addition to being able add and replace features of the simulation engine, developers can also create a simulation for an entirely new robot, without changing the application code. A developer only needs to create a set of 3D model files, a configuration file, and an implementation of the InvKin interface in order to simulate a new robot.
The Graphical User Interface

The graphical user interface (GUI) provides users with a mechanism to interact with simulation engine; users have the ability to input robot command programs, control the simulation, and view the current state of the simulation. In addition to control elements, the GUI addresses usability concerns, ensuring that a novice user can quickly and easily learn to interact with the application, while a more advanced user can access features in a faster, more direct way.

Overview of Simulator User Interface

The main application window for the simulator, shown in Figure 17, consists of several important interface components: the Simulation Viewport, the Simulation Variables, the Program Editor, menus, and an application toolbar.

Figure 17: The REMOTE User Interface Overview
The Simulation Viewport is responsible for rendering the current state of the robot during the simulation using a 3D model of the robot’s links; users can adjust the simulation display to view the simulation using a variety of camera manipulation tools from the toolbar or menus. More exact information on the exact position of the robot’s joints can be obtained from the Simulation Variables display, which displays joint information synchronized with the 3D display during the simulation. The Program Editor allows users to assemble a set of simulation commands, before sending them to the simulation.

To manipulate the execution of the simulation and the 3D display, a user can use any one of three possible methods: menus, accelerator keys, and toolbar buttons. Menus are useful in that they allow novice users to browse the available options and familiarize themselves with the application. For the more advanced user, accelerator keys provide a shortcut to common menu options, some of which are two layers deep in the menu hierarchy; this functionality allows more advanced users to perform operations more quickly than using the mouse. Finally, toolbar buttons provide fast access to common features; toolbar buttons act as visual shortcuts, allowing users to quickly execute commands without searching menus, or recalling an accelerator key combination.

Additional information about specific features is provided in several ways: on-line help, tool-tips, and information dialogs. On-line help provides tutorials and a catalog of the simulator features to guide the user into using the application; all on-line help is HTML-based, allowing developers to add further documentation as the simulator is enhanced. Tool-tips and information dialogs provide more immediate feedback to the user on their actions, errors, or intended actions; for example, the tool-tip shown in Figure 18 further clarifies the purpose of the ‘Paste’ button if the user’s cursor pauses over the button.

![Figure 18: Tool-tip for the Paste Toolbar Button](image)
Almost all of the features of the user interface are configurable, in order to allow for future customization by developers. In addition to allowing customization of the on-line help, the text for all error messages, information messages, menus, accelerator keys and buttons images are configured by external property files. Property files are loaded automatically for the particular language and dialect set by the user’s operating system; if a property file for the particular language is not available, a default file is used instead.

**User Interface-Simulation Engine Communication**

In order to synchronize the user interface with the current state of the internal model of the robot, a mechanism is required to:

Define a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically (Gamma et al., 1994, 293).

In the case of the simulator, the dependents that need to be updated are the 3D model of the robot, and the user interface components responsible for displaying joint data; the Observer pattern shown in Figure 19 fulfills these requirements.

![Observer Pattern](image)

Figure 19: The Observer Pattern (Gamma et al., 1994, 294)

The standard Java libraries define a Subject class, and an Observer interface, meaning the simulator user interface and 3D display only need to implement the Observer interface in
In order to provide synchronization between the GUI and the internal robot model, whenever an Interpolator increments the position of the Robot, it calls `Notify()` on the Subject class, which calls `Update()` on each Observer; this notifies the Observers to retrieve the latest joint information from the Robot class, and update themselves accordingly.
The Future of REMOTE

The current application, although complete, could be extended to provide additional capabilities, and allow it to be used in several other areas of ENSC 489. As REMOTE develops, several new features could be added to enhance both the usability and the features of the application. The following sections provide a “wish-list” of the desired features to be incorporated into the simulator in the future.

General Inverse Kinematics Capabilities

Currently, a developer wishing to simulate a different type of robot needs to create a class which implements the InvKin interface, and specify the class name in the robot’s configuration file. This class is required to provide the simulation engine with the ability to perform inverse kinematics calculations for the particular robot; however, a more general inverse kinematics engine could be created to replace this class, eliminating the need for any programming in order to simulate a new robot. The new inverse kinematics engine could “plug in” to the existing framework, and use the Robot singleton to create the inverse kinematics calculations. A developer would only need to specify the inverse kinematics engine in the configuration file, instead of specifying an InvKin implementation class.; therefore, no modifications to the existing code would be required.

Expanded 3D Model Support

At present REMOTE does not support any of the more sophisticated features of many 3D formats, including material properties, and lighting. The current capabilities provided to REMOTE by Sun’s ObjectFile 3D file parser class are too limited; however, the possibility of third-party parsers will end this limitation. These third party parsers could be used by using the ModelParser interface as a bridge between the simulator and the third party loaders, with minimal effort. However, recent additions to the Java3D API have specified a consistent interface for classes that provide 3D file parsing capabilities; it would be easier to replace the ModelParser interface with this new interface, thereby eliminating any additional work required to use third-party parsers that conform to this interface.
**Improved User Interface Libraries**

The use of Swing (the new user interface libraries for Java) in the application is limited by problems relating to mixing heavyweight and lightweight components; heavyweight components are user interface elements which use the underlying windowing system's native components, whereas lightweight components are written in 100% Java. At present (April 1999), the Java3D library uses a heavyweight component to display 3D models, which prevents REMOTE from using some Swing components; use of both lightweight and heavyweight user interface components can result in “collisions”, as demonstrated in Figure 20.

![Figure 20: Heavyweight-lightweight Component Collision](image)

Sun is currently rumored to be working on a solution to this problem, which would provide a lightweight Java3D canvas component to display 3D models. It is anticipated that this solution would be interface-compatible with the current component, meaning minimal work would be required to incorporate Sun’s solution into the simulator.

**Context-Sensitive Help**

In an effort to improve the usability of the REMOTE application, it would useful to incorporate the JavaHelp system, currently under development at Sun, into the application. This addition would allow REMOTE to provide users with context-sensitive help, which applies to the user’s current problems directly; for example, the context-sensitive help system could explain the meaning of an error message to a user in greater detail, and provide a possible solution. An easy-to-use help database of these topics would allow users to search for instructions on how to use the features of the simulator, and a troubleshooting guide.
Multiple Robot Simulation Support

The next logical step in the efforts to provide a meaningful robot simulator to students would be to enable simulation of multiple robots at the same time; more complex systems could be simulated, such as CAD/CAM manufacturing stations, or dexterous manipulators with multiple independent link chains.

Several modifications would be required to allow multiple robots to be simulated:

1. The Robot class would no longer consist of a single static instance of itself; instead, it would hold onto an array of Robot instances, accessible via an index number.
2. The configuration parser would have to be modified to configure multiple robots.
3. The command parser would have to be modified to associate commands with a specific robot.
4. The Command interface would need to be updated to associate its operations with a specific robot.

The most difficult part of this addition would be the modification of the robot configuration parser, in order to allow for each robot to be positioned relative to the other robots. This would allow for complex manipulators, such as dexterous manipulators, to appear as a complete robot.

Remote Robot Server

The long term goal of REMOTE is to provide remote-control capabilities, allowing students to program and debug a simulation, before sending their program on to a remote robot server. Currently, Sean Lavin has created an application that allows a user to remotely control the Scorbot ER-III robot using a Java applet in a web browser, while viewing the results over the Internet using CU-See-Me video (Lavin, 1999). In the future, the REMOTE project will attempt to integrate these solutions into a single tele-learning application for students to use to perform remote experiments.
Summary

This thesis has outlined a flexible simulation architecture based on design patterns, which permits future developers to replace and extend key pieces of functionality without altering the central simulation engine implementation. Using this architecture, developers can provide REMOTE with new simulation languages, new configuration parsers, 3D-model interpreters, and simulation commands. Default parser implementations provided with REMOTE allow a developer to implement a simulator for a completely new robot with a minimum of difficulty and programming.

In the future, minimal extension of the core simulation architecture will be able to combine the current solution with work being completed by other students, in order to enable the simulator to command a remote robot. Once completed, this new simulator will provide students with additional access to valuable equipment, and allow remote experimentation.
Appendix A: REMOTE Robot Configuration Syntax

The robot configuration file provides the simulator with the location of the physical and mathematical models of the robot, and the Denavit-Hartenberg parameters for the robot. A robot’s links are defined using the following flowchart:
INVKIN (classname)
Description: provides the name of the class providing the robot’s inverse kinematics.

SRC {file}
Description: provides the location of the 3D file for the link, and begins the block defining a joint’s parameters.

MODELPARSER {modelparser}
Description: specifies which parser should be used to read the model file.

LENGTH (link length)
Description: provides the integer link length.

DOF (number of degrees)
Description: specifies the number of degrees of freedom in the current joint, and begins the block defining a degree of freedom’s parameters.

NAME {name}
Description: specifies a name for the current degree of freedom.

TYPE {REVOLUTE, PRISMATIC, or INANIMATE}
Description: specifies if the current degree of freedom is a revolute or prismatic joint. For ‘placeholder’ type links with no motion, the ‘inanimate’ type is used.

OFFSET (link offset)
Description: specifies the integer link offset distance.

TWIST (link twist)
Description: specifies the integer link twist (in degrees).

MIN {minimum}
Description: specifies the minimum value for the degree of freedom.

MAX {maximum}
Description: specifies the maximum value for the degree of freedom.

HOME {home}
Description: specifies the home value for the degree of freedom.

DOF-END
Description: ends the block defining of a degree of freedom.

JOINT-END
Description: ends the block defining a joint.

END
Description: ends the definition of the robot’s configuration.
Appendix B: REMOTE Simulation Language

REMOTE uses a set of commands based on the SCORBASE language to direct the simulation; this appendix details the purpose and syntax of these commands.

**HOME**
Description: moves the robot to the home position.
Example: HOME

**MOVE (joint) (value)**
Description: moves the joint named (joint) to the given (value).
Example: MOVE TORSO 30

**MOVE XYZ (x) (y) (z)**
Description: moves the robot’s tool to (x), (y), (z).
Example: MOVE XYZ 200 200 0

**DEFINE XYZ (name) (x) (y) (z)**
Description: defines an endpoint called (name) given by (x), (y), (z).
Example: DEFINE XYZ waystation 200 200 0

**GOTO XYZ (name)**
Description: moves the tool to the XYZ position defined by (name).
Example: GOTO XYZ waystation

**DEFINE POS (name)**
Description: gives the robot’s current position the mnemonic (name).
Example: DEFINE POS waystation

**GOTO POS (name)**
Description: moves the robot to a position defined by (name).
Example: GOTO POS waystation

**SET SPEED (speed)**
Description: sets the robot’s motor speeds.
Example: SET SPEED 5

**WAIT (time (in milliseconds))**
Description: Waits for (time) before issuing next command.
Example: WAIT 20
Appendix C: Running the Example Simulation

An example simulation file accompanies the current installation of the REMOTE simulator; users can use this simulation file to demonstrate the capabilities of the simulator. In order to run the example:

1. Start the REMOTE simulator (see the documentation accompanying the installation).
2. Choose File->Open, and an “Open File” dialog will appear.
3. Go to the “examples” directory directly below the REMOTE installation directory. Double-click on the file “example.sim”. The contents of this file will be loaded into the Program Editor.
4. Choose Simulation->Run to run the simulation. The robot will execute the movements described by the simulation file (see Appendix B for a description of the simulation commands currently implemented by REMOTE). Comments within the simulation file also describe the purpose of each set of commands.
Appendix D: Denavit-Hartenberg Parameters

The Denavit-Hartenberg parameters are used to define the relationship between one link of the robot and the next link in the chain. The diagram in Figure 21 shows the layout for the definition of the Denavit-Hartenberg parameters.

To change the frame of reference from the first link, $i-1$, to that of the second link, $i$, involves four parameters:

1. Link length, $a_i$, defined as the distance between $Z_i$ and $Z_{i+1}$, as measured along $X_i$.
2. Link twist, $\alpha_i$, defined as the angle between $Z_i$ and $Z_{i+1}$, as measured about $X_i$.
3. Link offset, $d_i$, defined as the distance between $X_{i-1}$ and $X_i$, as measured along $Z_i$.
4. Joint angle, $\theta_i$, defined as the angle between $X_{i-1}$ and $X_i$, as measured about $Z_i$.

Note that rules define the direction of the $Z$ and $X$-axes for a particular link:

1. The $Z$-axis coincides with the axis of revolution for a revolute joint, or the direction of offset for a prismatic joint.
2. The $X$-axis coincides with the mutual perpendicular between the $Z$-axes of $i$ and $i+1$ joint, defining the link length.
References

Branden, Scott and Ron Racine. December 15, 1995. WWW Scorbot Simulator with
Inverse Kinematics in Java: Final Project Report for ENSC 439(95-3). Vancouver:
Simon Fraser University.


Cambridge, Massachusetts: O'Reilly & Associates.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1994. Design
Patterns: Elements of Reusable Object-Oriented Software. Don Mills, Ontario:
Addison-Wesley Publishing Company.


Quaterni, Terry. 1998. Visual Modeling with Rational Rose and UML. Don Mills,

Publishing Company.