http:rossum.sourceforge.net

Rossum's Playhouse (RP1)
User's Guide for Version 0.60

12 January 2005

The Rossum Project gratefully acknowledges the assistance of Sonalysts, Inc. in the production of these documents. Learn more at http://www.sonalysts.com

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Document Scope

This document provides information on how to use the Rossum's Playhouse (RP1) robot simulation. It is intended to serve as a companion piece to the other two components of the RP1 documentation set: a Javadoc-generated HTML reference giving details on the Application Program Interface (see paragraph 1.6.4 below), and example code demonstrating methods for implementing simulator applications (paragraph 4.2, etc.).

The RP1 User's Guide is divided into 5 sections. The first three provide background information that will help you understand the architecture of the system. You may either skim these or read them in detail depending on your preference. The fourth and fifth sections discuss how to create your own virtual robots and robot software for the simulation environment. If you wish to use the simulator for that purpose, you should read these sections in detail.

As with everything else related to Rossum's Playhouse, this documentation is a work in progress. It will be improved in the future as user needs become apparent. And although the document is incomplete, the Users Guide does attempt to identify the important elements of the system and provide sufficient detail to get you started. Most of the more complicated issues of the simulator's implementation are encapsulated in the systems internals, so you will not have to deal with them when writing RP1 applications. So by combining this Guide with the Javadoc and example code, you should be able to glean a good understanding of RP1, its behavior, and good strategies for using this simulator.

RP1 is coded in Java. In writing this document, we have tried to avoid relying too much on the reader being familiar with the Java programming language. Some knowledge of Java would certainly be helpful, but it is not absolutely required. Paragraph 1.6 below gives a sketchy introduction to the language and its more important terminology.

1.2 System Overview

Rossum's Playhouse (RP1) is a two-dimensional mobile-robot simulator. It is intended to be a tool for developers who are building robot navigation and control logic. RP1 does not model any particular robot hardware, but provides components that can be configured and extended to represent the platform of choice. The simulator is a starting place for software development, not an end in itself. A key factor in the RP1 design is that code developed using the simulator can be moved to the target hardware more-or-less intact.

All source code for the RP1 system is provided in the standard software distribution.

1.3 RP1 as a Collaborative Effort

Rossum's Playhouse is intended to be a collaborative development effort. In the last 10 years, a number of successful systems have been developed through collaboration. Examples include the Linux operating system, the GNU C compiler, and the Apache web server (which runs more than half the web pages on the Internet). All of these systems grew in both features and sophistication

through the contributions of code, algorithms, and general comments of interested users and developers.

We hope that RP1 will follow the model established by these systems (albeit, on a *far* smaller scale). From the beginning, the software was implemented with this goal in mind. To support collaboration, the RP1 system design strives for openness. The architecture is modular and there are plenty of opportunities for improvement.

We are seeking code, algorithms, and recommendations for useful features. A "wish-list document" is included as part of the RP1 software distribution.

1.4 Use and License

The RP1 source code is freely available and may be copied and distributed according to the stipulations of the GNU General Public License (GPL). The full text of the GNU General Public License is provided in the text file "gpl.txt" which is included in the RP1 software distribution. Note that the GPL applies only to the RP1 code itself or code derived directly from RP1 source. The GPL license does not, in any way, assert claims over original code that merely interacts with the RP1 environment.

1.5 Programming Language and Run-Time Environment

All source code for Rossum's Playhouse is written in Java. All run-time code currently available is in the form of compiled Java class files.

Of course, Java is not the "language of choice" for many robot developers. Keeping the system language-independent was one of the most important element in its design. The techniques used to support this goal are described in the section on *System Architecture* below. And, in fact, some time into the project, a volunteer developer did contribute a C/C++ API for the simulator which is now available on our web site.

To run the Rossum's Playhouse simulation, you will require a system with the Java Run-Time Environment (JRE), Java Software Development Kit (SDK), or one of the many Integrated Development Environments (such as Borland's JBuilder or the Eclipse Foundation's free Eclipse IDE). Versions of the JRE or SDK for Sun Solaris or Windows architectures can be downloaded from the Internet from Sun's Java site http://java.sun.com/products. Other equipment manufactures have their own versions of the SDK. Rossum's Playhouse currently requires Java version 1.2 or later.

1.6 A Very Quick Introduction to Java

This section of the Users Guide provides a quick introduction to some of the terms and concepts used in Java. If you are unfamiliar with the language, these notes should help you navigate the remainder of the document. Syntactically, Java resembles C. It is a terse, but expressive and highly readable computer language. Developers with experience in other computer languages should be able to follow the code without undue effort.

If you are interested in learning more about Java, there are a vast number of books on the subject. Unfortunately, most of them are not very good. So you should exercise some caution before making a purchase. There are several good web sites offering Java tutorials and other introductory material. Some of the best are offered by Sun Microsystems, which originally developed Java, and the IBM Corporation, which has made a substantial investment in Java technology.

New to Java (Sun Microsystems)
http://java.sun.com/learning/new2java/index.html

New to Java (IBM)
http://www.ibm.com/developerworks/java/newto/

Java SDK 5 Documentation
http://java.sun.com/j2se/1.5.0/docs/index.html

Java SDK Tools and Utilities
http://java.sun.com/j2se/1.5.0/docs/tooldocs/index.html

As with any other web resource, these are subject to change. If you cannot find them, try visiting Sun's main Java site http://java.sun.com or the IBM site at http://ibm.com. Beyond that there are many, many other introductions, special-topic discussions, and tutorials on the web.

Java is an object-oriented programming language. The key entity in an object-oriented language is, naturally, the object. In an object-oriented program, all operations are accomplished through the interaction and manipulation of objects. The concept of the object is related to that of structures in C or records in Pascal. Like structures, objects contain data *elements*. In addition, they also provide *methods* for accessing and manipulating that data. Java uses the term "method" in the much the same way as C++ and other languages use the term "function." A method is simply a reference to executable code that is associated with a particular object or object definition.

In Java, objects are defined using the *class* specification. A class is an abstract entity, much like the type definition for a C structure. The distinction between classes and objects is important. Classes are specified in source code, objects are created at run-time by the application. The class specification is used to create an object, which then may be used to invoke the methods that were specified in the class. An object has a specific location in memory and it is possible for a program to create multiple objects of the same class. Some documents refer to the act of creating an object using the rather fancy phrase "to instantiate an object." Objects are sometimes called "instances" or "instantiations" of a class.

When a class includes a method, the method can be invoked from any object of that class. For example, suppose the class "Robot" includes a method called "moveForward". Then the following fragment of code shows how the moveForward method might be invoked:

```
Robot robbie = new Robot();   // the "new" instantiates an object
robbie.moveForward(1.5);      // move forward 1.5 meters.
```

Classes can also contain data elements that can be accessed in a similar manner:

```
double massInKg = robbie.massInKg;
```

Note that in Java, all statements end in a semicolon.

The Java equivalent of a program is called an "application." An application consists of one or more classes which are compiled from Java source code. Java source code is stored in files with the extension ".java". Each Java file defines one class with the same name as the data file. For example, the Java class "RsProtocol", which is used for communication in the RP1 system, must be defined in a file called "RsProtocol.java". The Java compiler, javac, processes Java source code producing files with the extension ".class". "RsProtocol.java" would be compiled to "RsProtocol.class".

Java is an interpreted language. The Java class files do not contain native machine instructions, but rather a set of abstract binary codes which are interpreted by the Java Virtual Machine (JVM). The JVM is a program that runs on the target computer and acts according to the specifications in the class file. On most systems, the JVM program is called "java". A Java application called Foo is run using the command "java Foo." The JVM looks for a class file called "Foo.class" and uses it to run the application.

An application can consist of a great number of classes, often with associated data files or computer-graphics image files. As an application grows in complexity, it becomes convenient to collect its components into what Java calls "packages."

A package is, roughly, the Java equivalent of a "library." A package is a directory or an archive containing multiple classes and related data files. Java packages have a close relationship with the file directory structure. All the class files for a particular Java package must be stored in a directory with the same name as the package. For example, the RP1 software distribution provides a package called "rossum" which includes general tools for building RP1 client applications. All the class files for that package are stored in a directory that is also called "rossum."

When dealing with an application or a package that contains a large number of files, it is often convenient to store those components in an archive. Although Java code can be stored in conventional zip files, Java also provides its own archive format called the "jar" file. The Java jar format is similar to that of zip files, but jar files can be accessed at run time without being previously uncompressed. Jar files also have to virtual of being portable to non-Wintel systems.

Summarizing the terms introduced above:

.

.java	the extension used for Java source code files
.class	the extension used for compiled source code
.jar	the extension for Java archive files
application	the Java equivalent of a program
instantiate	to create a new object of a particular class

instance	an actual object of a class
method	the Java equivalent of a function or subroutine
package	a collection of Java classes (equivalent to a library), packages are always associated with a folder (directory)

1.6.1 Applications, Applets, and Browsers

As mentioned above, the Java equivalent of a program is an application. An application is run when the user launches the java executable (or other JVM) directly from the command-line or windowing environment. Java also supports a kind of psuedo-application, called the "applet" which can be launched from a Java-capable web browser.

The main difference between an applet and an application is that, for security reasons, an applet has limited access to a system's resources. Security is an important consideration when using applets because they are often downloaded from the Internet from unknown and potentially unreliable sources. When run from a web browser, applets will usually be unable to read and write files on the user's system.

All the RP1 software is provided as applications. They can be modified to function as applets, but the current revision does not provide an API to support such a use directly.

1.6.2 Important Considerations about CLASSPATH and Packages

The most common problem that users new to Java have when running an application is difficulty with class path settings. The text below gives some introductory material and, for more information, you can also visit the following web pages:

How Classes are Found
http://java.sun.com/j2se/1.5.0/docs/tooldocs/findingclasses.html

Setting the Classpath
http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/class path.html

When a Java application is executed, it needs to know where to find the various class definitions and other resources that are associated with it. Unlike programs, the binaries for a Java application (the class files) are not linked together before the application is executed. The class files that comprise the application may be scattered across several packages (directories or folders). And applications often require additional resources such as graphic elements (gif or jpeg images), properties files, session data, etc.

To provide this information, Java uses a concept called the class path. The class path is a string specifying paths to folders, directories, or zip and jar files where Java is supposed to search for class resources. A class path can define multiple folders or directories by separating the specification with either a semicolon (under Windows) or a colon (under Linux/Unix).

In windows:

```
CLASSPATH = c:\Program Files\Java\MyStuff;c:\Robotics\RossumsPlayhouse
```

in Linux/Unix:

```
CLASSPATH = /usr/local/bin/MyStuff:/home/users/smith/RossumsPlayhouse
```

By default, Java will look in the current directory or folder when you launch an application (earlier versions of Java did not). If the class path is defined, Java can also look in those paths specified as shown in the example above.

Typically, the class path is set as an environment variable. In Windows, environment variables are set using the System tool (Programs/Accessories/System) or in the Properties menu (under "Advanced" associated with the "My Computer" icon). In Unix and Linux systems, it is set in a resource file in the user's home space (.profile for the Korn Shell environment, .bash_profile for the BASH shell environment).

The specification of a class path is closely tied to the location of the packages (folders) containing the RP1 source code and classes. The main class for RP1 is called "Server" and is stored in a package called "rp1". Once Java is properly installed and the CLASSPATH environment variable is set, the RP1 simulator can be launched in a command window (on Unix, Linux, or Windows) or in the Windows Start/Run utility by typing the following command:

```
java rp1.Server
```

The java command launches the Java Virtual Machine (main Java interpreter). The rp1 specification tells it to look for a package called "rp1" within the various folders specified by the class path setting. The Server specification tells it to look for a class by that name once it has identified the package. From there, Java can find all the information it needs to run the RP1 simulator.

1.6.3 Package and Import Statements in Java Source Code

The Java language includes two important keywords related to package and class path. The first keyword, *package*, allows the source code to declare its membership in a particular package. Although Java does permit the package statement to be omitted, all RP1 source code uses explicit package statements. For example, the main container package for all the primary RP1 source code is called rp1. General utilities are under the package rp1.rossum and source code intended solely for simulation purposes is under the package rp1.simulator. Thus if you examine Java source code in the rp1.rossum package, you would see the following statement close to the beginning of each file:

```
package rp1.rossum;
```

When processing the RP1 source code, the Java compiler uses the class path to find the directory (folder) named rp1 and then follows the sub-specifications in the package statement.

Code that accesses the classes in rp1.rossum must include declarations that tell the Java compiler where to find those classes. This specification is accomplished through an import statement. Import statements can include a specific class or use a wildcard specification (an asterisk) to import all classes in a particular directory as in the following examples:

```
import rp1.rossum.RsProperties;

import rp1.rossum.*;
```

The rossum package itself is divided into two subdirectories, *event* and *request*. All of these include classes that are needed for RP1 applications. While it is possible to use a very selective syntax to specify what classes are needed, many application that use RP1 code will include the following statements

```
package myPackage;     // the package name for the application
import  rp1.rossum.*;
import  rp1.rossum.event.*;
import  rp1.rossum.request.*;
```

Note that event though it includes a wildcard, the statement "import rp1.rossum.*" will not enable the Java compiler to find classes in the subordinate packages event and request. The developers of Java have taken a very strict, constructionist approach on many of the implementation details in the language.

For example, the RP1 distribution includes a package called rp1demo that provides demonstration applications. The source code for those demonstration applications usually includes the use of the wildcard operator. In cases when only a single class was needed from one of the event or request sub-packages, the code sometimes includes that class using the highly specific form, such as:

```
import rp1.rossum.event.RsTimeoutEvent;
```

1.6.4 About Javadoc, an automatic API documentation generator, and RP1

Among the tools in the Java Software Development Kit (JDK) is a utility called Javadoc that can be used to automatically generate API-level documentation for Java code. Javadoc parses source code, extracting from it information about Java class structure, derivations, method signatures (calling arguments), and cross-references. This information is compiled into a HTML document that can be read using a web browser. Javadoc has the ability to read documentation from comments written in a special format and can include that text as part of the HTML.

As of release 0.60, the RP1 source code includes Javadoc-related comments and supporting information. The file RP1public.zip, which may be downloaded from the Rossum Project website, contains HTML documents generated for public classes and method calls in the rp1.rossum package and sub-packages. Most of these are elements needed for writing RP1 application software. If you download and unpack the zip file, you may examine Javadoc products using a web browser (in Windows, you can simply double click on the file "index.html" in the main RP1public folder).

The HTML in the standard distribution is restricted to the public elements in the rp1 packages. This restriction is applied to screen out information that may be distracting to an RP1 application developer who isn't interested in the simulator's internals. If you wish to see the additional information, you may run Javadoc yourself and generate HTML for the overall software distribution. A typical command would be

```
javadoc -d <your output directory>
  rp1.simulator rp1.rossum rp1.rossum.event rp1.rossum.request
```

At this time, Java comments have been added to most of the classes, but few of them are completely documented.

1.7 Directories, Files, and Hierarchy

The 0.6x distribution of RP1 includes a hierarchy of folders. In the top-level, there are four folders as listed in Table 1 and four files as listed in Table 2.

Table 1 – Folders in Top-Level Folder

Folder	Description
rp1	Source code and compiled class files for the RP1 simulator.
rp1demo	Source code and compiled class files for the RP1 demo applications.
FloorPlans	Text-based files providing the "floor plans" for simulated environments. Includes floor plans for various demo's.
Properties	Properties files (.ini extension) providing set-up data for various simulation runs. Includes properties for various demo's.

Table 2 – Files in Top-Level Folder

File	Description
readme.txt	Release notes
gpl.txt	Statement of GNU Public License (GPL)
FireFighter.bat	An example of a DOS ".bat" file that demonstrates how a user can launch the fire-fighting demo application by clicking on an icon in a file browser. This file is also intended to provide an example command-line argument and will work as a Unix/Linux shell script.
LineTracker.bat	An example of a DOS ".bat" file that demonstrates how a user can launch the line-tracker demo application by clicking on an icon in a file browser. This file is also intended to provide an example command-line argument and will work as a Unix/Linux shell script.

Recall that Java packages are associated with actual folders on disk. For example, the folder named "rp1", which was include in the list for elements in the top-level folder in Table 1 above, corresponds to the Java package of the same name. A subfolder of rp1 named "rossum" corresponds to the Java package rp1.rossum. From an application developer's point-of-view, the most important package in this set is rp1.rossum because it provides all the client-side classes. Table 3 lists the contents of the rp1 folder.

Table 3 – The Folder "rp1" and its Contents

Element	Type	Description
Server.java	Java source	The source code for a Java "main" method that can be used to launch the simulator.
Server.class	Java class file	The binary for the main simulator (compiled from source code Server.java)
RP1.ini	properties file	A Java-style properties file defining options for the RP1 simulator (the Server)
rossum	package	Java package *rp1.rossum* containing the source and compiled classes for the rossum package, a set of general tools used by both clients and servers. For an application developer, the most relevant classes include **RsClient,** and the classes associated with the **RsBody** class.
rossum/ event	package	Java package *rp1.rossum.event* containing the source and compiled class files for RP1 events (see section 5, *Events and Requests*).
rossum/ request	package	Java package rp1.rossum.request containing the source and compiled class files for RP1 requests (see section 5*, Events and Requests*).
simulator	package	A Java package containing the source and compiled classes for the simulator package
planparser	package	A Java package providing code for parsing in floor-plan files.

As mentioned above, the standard software distribution contains example code under the package "rp1demo" in the main folder. Each folder has a "main class" that contains the main method for the demo. For example, under rp1demo.ackermain, the main class is called "Main.java" and the application can be launched with the command

```
java rp1demo.ackerman.Main
```

This command will cause Java to launch the "ackerman" demo application which, in turn, will attempt to connect to the RP1 Server.

Table 4 – Packages Included Under rp1demo

Package	Main Class	Description
ackerman	Main	Provides the simplest demo application, showing a vehicle with an Ackerman steering mechanism navigating an empty floor plan. Run: `java rp1.Server -p ackerman.ini`
clientzero	ClnMain	A simple client that interacts with user mouse clicks. Run: `java rp1.Server -p clientzero.ini`
demozero	DemoMain	The fire-fighting robot demo launched from the FireFighter.bat script mentioned above. DemoMain extends the class ClnMain and used the robot body specifications from the clientzero package.
linetracker	LtMain	The line-tracking robot demo launched from the LineTracker.bat script mentioned above.

1.8 Running RP1

Now that we've described the general layout of directories, packages, and the class path for RP1, we can describe launching the application. The main class for RP1 is called Server. It is stored in the package called rp1. Therefore, once the class path is set, we can run the simulator using the following command:

```
java rp1.Server
```

Java will search the folders specified in class path looking for the rp1 package. Within that package it will look for a class called Server. On Unix or Linux, the command given above can simply be typed in at the command prompt in a terminal window. On Windows, you can type in this command using either the Run utility under the Start menu or by launching a Command Prompt (a text-based interface resembling the old DOS command interface... in Windows-XP, the command prompt is found in Start/Programs/Accessories).

The class path is often configured to point to a particular folder, or set of folders, but Java provides an additional capability. It is possible to include .zip archives and Java .jar files as part of the CLASSPATH. The class loader used by Java will search the archive files just as if they were a standard directory hierarchy.

For example of how useful this feature can be, suppose you downloaded the RP1 distribution for some revision 0.6 or higher. Suppose also that you stored the archive file on your C disk in a folder called myStuff/RossumProject. Even if you hadn't set your class path, you could run RP1 and supply the class path as a command-line argument:

```
java –classpath c:/myStuff/RossumProject/Sim060.zip rp1.Server
```

1.9 Terms and Abbreviations

The following is a partial list of terms and abbreviations are used in this document or in related materials.

API	Application Program Interface
CAD	Computer-Aided Drafting
DLC*	Dynamically Loaded Client
GPL	GNU General Public License
ICD	Interface Control Document
IDE	Integrated Development Environment
J2SE	Java 2 Standard Edition
JRE	Java Run-Time Environment
JDK	Java Development Kit (old term for SDK)
JVM	Java Virtual Machine
RP1*	The simulator, "Rossum's Playhouse # 1"
SDK	Java Software Development Kit
TCP/IP	Transmission Control Protocol/Internet Protocol
jar	command to invoke the Java Archive utility
javac	command to invoke the Java compiler
java	command to invoke the Java interpreter (Java Virtual Machine)
javadoc	command to automatically create documentation of Java classes

*non-standard term, usage specific to RP1 project

1.10 Units of Measure

In the external user interfaces, the RP1 simulator and related applications are free to display distance and angular measures in whatever units the user pleases. In the floor plan specifications, the user may control which system of units are used to describe objects in the simulation environment (by default, metric units are used). In the floor plan, all angle and orientation values are given in degrees, measured anti-clockwise from the x-axis.

Internally, and in the classes and parameters that comprise the RP1 API, distance is always measured in meters. Internally, and in the RP1 API, angles are always measured in radians, anti-clockwise from the x-axis. Time is measured in seconds when using real-valued specifications. Linear velocities are in meters per second. Rotational velocities are in radians per second.

2 System Architecture

2.1 The Client-Server Architecture

RP1 is based on a client-server architecture.

Anyone who has ever operated a web browser has experienced a client-server architecture. The browser, a client program resident on one computer, connects to a server which is resident on another. The two programs exchange data via network-based communications. Often, more than one client may connect to the server simultaneously.

Though its implementation is quite different than that of a web browser/server, the relationship between RP1's client/server components is analogous. In RP1, the server is the simulator and the virtual world that it provides. The clients are the robots that occupy that world. The server provides the virtual hardware for those robots. It models their bodies and physical interactions with their simulated environment. It responds to their commands to change position or begin a movement and provides them with sensory feedback. But it does not control the robots. Control comes from the clients. In a sense, the clients provide the "brains" for the robot, while the server provides them with a "world."

You may develop and test robot logic by writing your own clients for the RP1 server. The standard Rossum's Playhouse software distribution includes code for example clients as well as a Java API to support their development.

2.1.1 Network/Local Connections versus Dynamically Loaded Clients

The RP1 client and server can run in two different modes:

1. as two individual programs, potentially written in different languages and running on different computers (if the programs run on different computers, the client establishes a *network* connection; if they run on the same computer, the client establishes a *local* connection).

2. using a *Dynamically Loaded Client (DLC)* which is loaded into the server at run time so that both client and server logic is executed within a single application.

Each of these approaches has its own advantages and disadvantages. If you implement the client and server as separate programs, it does permit you to run a client that is not written in Java.[1] It also provides better isolation between the client and the server for debugging purposes. If the client is crashing, it does not affect the server (and visa versa). And, again, it even allows you to run a client on a different machine (and even a different *kind* of machine) than the server.

[1] A C++ API is available for release 0.5 of the simulator. Unfortunately, changes made to release 0.6 rendered it incompatible. Updates are planned for the future.

On the other hand, the Dynamically Loaded Client (DLC) can be easier to use. It is easier to manage one program than two. And because a DLC can be incorporated into the same JVM session, it has the option of bypassing the network-communications mechanisms and using much more efficient data transfers than when separate programs are used.

At this time, dynamically loaded clients are restricted to Java applications.

2.2 Why a Client-Server Architecture?

2.2.1 Language Independence

The primary motivation for adopting a client-server architecture was language independence. Although the RP1 simulator is written in Java, few developers implementing robotic software are currently working in that language. The division of the simulation into separate entities, a client and a server, allows the two components to exist in different environments. Client programs are free of any dependency on Java. They may be written in a number of different languages and may even run on a different computer than the simulator/server. The only system requirement is that the platform of choice be able to handle elementary data types and network connectivity.

2.2.2 Quicker Development for User Implementations

The client-server architecture simplifies user implementations by separating their code from the simulator software. Code written for independent clients tends to be more modular and more easily extracted from the simulator. The overall size of client executables is reduced, thus permitting faster development, testing, and modification. And because the client and server are implemented as separate entities, it becomes easier to identify the source of problems. If the client crashes, the simulator continues to run and can accept new connections without delays for start-up times.

2.2.3 Extensibility

Even a full-featured simulator cannot meet every user need. Eventually, there will be some requirement that falls outside the range of the simulator. The client-server architecture used for RP1 allows users to easily develop interface layers in their applications to allow them to extend the capabilities of the simulation. Because the client-server architecture provides the user with more control over their own code than they would in an integrated simulation, the architecture allows them more freedom in building their applications. Users are free to import tools, adapt existing solutions, and code the algorithms without the restrictions that other architectures might impose.

2.3 Client-Server Communications

Of course, for a client-server architecture to work, there must be some mechanism for the client and server to talk to each other. In paragraph 2.1.1, we mentioned that there were two modes for running the client and server modules. If the modules are run as separate programs (or as Java applications running in separate Java Virtual Machine sessions), RP1 uses a network-based communication mechanism known as the "socket." If the client module is loaded dynamically, it can still use socket communications, but might benefit from a more efficient mechanism known "piped data streams" or "pipes." Pipes are a means of allowing one part of a program to

talk to another part as if they were separate entities[2]. Because pipes are restricted to a single program, they can bypass the overhead associated with socket communications.

2.3.1 Network and Local Connection Issues

When running as separate programs, RP1 clients and servers communicate using the same protocol that is used in many internet applications, including net browsers and email applications. This protocol, which is called TCP/IP (or, in informal settings, Internet protocol), provides reliable connections between the client and server and permits them to communicate without loss of data. While TCP/IP is widely used for internet communications, it is also suitable for exchanges between programs running on the same computer. When a client connects to a program running on the same computer, it is said to be making a connection to the *localhost*, thus making a *local connection* rather than a *network connection*.

An excellent discussion of TCP/IP network communications can be found in "TCP/IP Network Administration" by Craig Hunt, published by O'Reilly and Associates.

When a TCP/IP client attempts to connect with a server, it does so by specifying a hostname or internet address. Hostnames are generally human-readable strings. Before they can be used by the computer, they need to be resolved into to an IP address. An IP address is a single integer value which gives a unique address for every computer on the associated network (and, perhaps, the Internet itself). For human-purposes, IP addresses are typically expressed as a set of four values such as 198.186.203.33.

Any particular host may be running several servers (or "services"). So in addition to the host address, a client wishing to connect to a particular server will require something to indicate which one it wants. This specification is accomplished through the use of a port number. A port number is an arbitrary two-byte integer value. Service are assigned to port numbers according to convention and accepted practice. For example, http (worldwide web) connections user port number 80. The POP3 post office protocol (used for email) uses port 110. The file transfer protocol (ftp) use 21, etc. Port values between 0 and 1023 are commonly reserved for system applications (such as ftp, telnet, timed, etc.) and should not be used for other applications.

In the current revision of Rossum's Playhouse, details such as port assignment and host are specified in the properties file "rossum.ini" which is described below. In the version supplied with the Version 0.6x distribution, the following values are supplied:

host: 127.0.0.1
port: 7758

The IP address 127.0.0.1 is called the "loopback address." A loopback address allows a machine to make a virtual connection to itself. It allows a client to look for the server on the same machine as the one on which it is running. If you wish for a client to communicate with a

[2] In Java, pipes are actually a method of allowing different *threads* to exchange data and, in fact, may result in deadlock if used to communicate within a single thread.

machine other than its own host, you will have to change this IP address. If your network is configured properly, the RP1 software will also accept hostnames as entries in this field. The port number is read from "rossum.ini" and reserved for client connections when the server starts up. There is nothing special about the value 7758, we simply chose a port number that we thought would not be used by another application. There is a small, but non-negligible, probability that there will be another process running on your system which allocates this port number. In that case, there will be a conflict and the simulator will be unable to run. The conflict will be noted in the RP1 run log. To resolve the problem, simply chose a different port number and restart the simulator.

2.3.2 Communication via Events and Requests

Once the server (simulator) is running, it may accept connections from clients at any time using the TCP/IP protocol. TCP/IP is a low-level protocol sufficient for establishing connections between clients and servers, but most applications require that a higher-level protocol be implemented "on top" of TCP/IP to support the exchange of meaningful data. Rossum's Playhouse implements a set of conventions known as the "RP1 protocol."

In the RP1 protocol, most data is exchanged between a client and server in the form of messages. Messages sent from the client to the server are described as "Requests." Messages from the server to the client are "Events." A typical request might be something like "set the wheel velocities for the robot drive system." A typical event might be "sensor 1 detected a light source 5 degrees off its central axis." The choice of these terms reflects the realities of controlling an actual robot. Although an operator might "want" a robot to move in a particular way, physical issues (such as wall collisions or power limitations) might make it impossible for the system to comply. Therefore, messages sent to the robot are described as "requests" rather than "instructions" or "commands." Similarly, the choice of the word "event" reflects the probability that conditions might occur in the simulated world that are unpredictable and require the client to respond without warning.

In general, you will not need to know the details of the RP1 protocol. It is handled by the rossum API which provides a simple interface for exchanging data.

2.3.3 Keeping the RP1 Protocol Language-Independent

The importance of permitting clients to be written in languages other than Java was noted above. In the existing RP1 code, messages are implemented (quite naturally) as Java classes. Java provides an elegant method for transmitting classes between different applications ("applications" are the Java equivalent to "programs"). But this method, called "object serialization," requires a special data encoding that is not easily implemented in other languages.

To avoid language-dependencies, RP1 implements its communications using primitive data elements such as floats and integers. These elements are defined in an architecture-independent format that allows communications across normally incompatible platforms.

2.3.4 C/C++ API for RP1 Clients

From the beginning of software development for RP1, it was hoped that other members of the open-source community would collaborate to provide support for client applications and programs written in languages other than Java. One such volunteer, author James Y. Wilson, wrote a C++ API that is currently available for RP1 revision 0.5 and older. Due to changes in the interface, this API is not compatible with revisions 0.6 but will be updated in the future.

2.3.5 Documentation for the RP1 Protocol

The RP1 protocol is not documented in this time. If there is sufficient interest, an Interface Control Document (ICD) may be developed in the future. Again, if your application uses the RP1 API, all details of the protocol should be encapsulated by the software.

If you do need to investigate the RP1 protocol, you may do so by studying the source code. Because we lacked the resources to develop an ICD for the initial releases of RP1, we tried to design the Java classes related to communications so that they would be largely self-explanatory. Users and developers may obtain the information about the RP1 protocol by studying the following classes found in the "rossum" package (subdirectory):

RsProtocol	base class for RP1 protocol;
RsClient	client-side communications;
RsConnection	server-side communications.

2.4 Configuration Elements and Properties Files

All RP1 client applications depend on at least one properties file, rossum.ini, which specifies the network communications port that clients can use to connect to the Server. The Server, of course, also needs this information so that it knows on which port to establish its service. The rossum.ini file is treated as a Java "resource" and is stored as part of the rossum package (recall that Java packages are equivalent to directories). Because it is treated as a resource, you do not have to worry about file path. As long as Java can find the rossum package, it will be able to find the rossum.ini resource (and if it can't find the package, you won't be able to run the application anyway).

The information in the rossum.ini file is used to populate the elements of a Java class known as RsProperties. The RsProperties class is derived from Java's standard Properties class. In Java, Properties can be used to read a file which specifies a set of values using a simple syntax which resembles a traditional assignment statement:

```
# rossum.ini -- fundamental specifications for all RP1 applications.

port=7758
hostName=127.0.0.1
logFileName=rossum.log
logToFile=false
logToSystemOut=false
logVerbose=false
```

Figure 1 – Properties specifications from rossum.ini

Java's Properties class provides methods for getting the strings associated with each properties name. Because RsProperties extends Properties, it inherits all of Properties methods. For example

```
RsProperties rsp  = new RsProperties();
String       name = rsp.getProperty("hostName");
```

2.4.1 The "port" and "hostName" Properties

RsProperties also extends Properties by adding fields that are relevant to RP1. The most prominent of these is the "port" value mentioned above. It also supplies the host specification for RP1 clients. The Server will always need the port specification. The clients will always need the host specification (to find the Server). All RP1 applications that use RsProperties read the rossum.ini file and, except where you override the specifications (discussed below), all RP1 applications will use the port value assigned in rossum.ini. This feature makes it possible to reassign the port value for the entire family of RP1 applications by modifying a single file.

You may modify the rossum.ini file as you see fit, but exercise caution in doing so. The Java Properties syntax is very fussy. It is case-sensitive and does not tolerate embedded white space characters. All specifications must be completed in a single line.

2.4.2 Overriding Properties

The rossum.ini file also includes specifications for logging. Typically, these are overridden by loading data from additional .ini files. For example, suppose a client application wishes to use the specifications from a data file. It could do so by invoking either the load() method from the Java Properties class, or the loadFromFile() method defined by RsProperties:

```
RsProperties rsp = new RsProperties();
rsp.loadFromFile("client.ini");
```

The loadFromFile method looks for files from either the current working directory or, if specified, from a fully qualified file pathname. The example above was simplified a bit for purposes of clarity. Consider what would happen if the "client.ini" properties file was not found or if it contained syntax errors. Java would detect the error and *throw an exception*. In order for the example code to compile, Java requires that it *catch* the exception from the loadFromFile() method:

```
RsProperties rsp = new RsProperties();

try{
    rsp.loadFromFile("client.ini");
}catch(RsPropertiesException e){
    System.err.println("Error reading client.ini "+e.toString());
}
```

2.4.3 Loading RsProperties Files as a Resource

The loadFromFile() example above reads properties from a file found in the "file path." The default file path is whatever directory you happen to be in when you launch Java. An alternate file path may be specified through the argument to loadFromFile().

You may find it convenient to keep your properties files bundled up with your class files as a *resource* in the same manner as the rossum.ini file. You can do so by using the RsProperties loadFromResource() method. Pass the method an object belonging to a class defined in the same directory as your properties file. RsProperties will use that object to obtain the path to the properties file.

```
Example object   = new Example();       // an object is created
RsProperties rsp = new RsProperties();

rsp.loadFromResource(object, "client.ini");  // throws an RsPropertiesException
```

In another variation, the example below creates extends RsProperties and load the specifications as part of the constructor:

```
public class ExampleProperties extends RsProperties {

    public ExampleProperties()   // the constructor
    {
           super();               // RsProperties is the "super class"
                                  // super(); invokes its constructor
           try {
                  loadFromResource(this, "client.ini");
           }catch(RsPropertiesException e){
                  // handle this according to your own needs
           }
    }
}
```

2.4.4 Adding Custom Specifications

Because the RsProperties class inherits all the functionality of the Java Properties class, client applications may add custom specifications to their own .ini files.

3 The Server

The main Java application for Rossum's Playhouse is named Server. The RP1 Server is a multi-threaded application that manages both the simulation functions and client communications. The server has an optional GUI that depicts the robot simulation as it navigates its virtual landscape. In Revision 0.6x, the GUI is quite primitive and has potential for considerable refinement in future versions.

Information about how to run the main server is provided in the release-notes file "readme.txt" which is included in the top-level directory of the software distribution.

3.1 Server Properties Files

Upon start up, the server reads two Java-style property files. The first is the "rossum.ini" file that was discussed above. From this file, it obtains the port specification for accepting TCP/IP connections (the host specification is irrelevant to the server). It then reads the file "server.ini" which supplies server-specific settings as shown below.

Table 5 – Properties Used by the Main Server

Property	Description
floorPlanFileName	Supplies the name of the floor-plan file (see below) which is to be used to specify the simulated environment.
enableGUI	Boolean (true or false) value indicating whether the GUI is to be presented. Default value is true.
enableNetworkClients	By default, the simulator is configured to accept network connections by clients (or clients connecting from the local host using the systems network facilities). If you are running dynamically loaded clients (see below), and do not need this capability you may suppress it. Doing so is useful on systems where network drivers are not installed.
animationFrameRate (specified in frames-per-second)	The number of frames per second used for animation when the GUI is activated. If you make this value too large, the motion will appear jerky. If you make it too fine, the simulation will run slow due to the overhead of rendering the image. Obviously, different values work on architectures with different Java graphics capabilities. Default is 20 frames per second.
modelingFrameRate (specified in frames-per-second)	Many robot motion problems are modeled in terms of discrete intervals. The modelingFrameRate provides a default value for the number of intervals per second which are used for modeling purposes. Note that the period derived from the modelingFrameRate is a *maximum* value. Under certain algorithmically determined conditions, the robot position and motion will be evaluated more often than specified by the modelingFrameRate
simulationSpeed	The simulationSpeed specification allows the simulation to be run in fast-forward or slow motion. A value of 1.0 means that the

Property	Description
	simulation will run in real time, 0.5 at half speed, 2.0 at twice normal speed, etc. You may set this value as large as you please, but at a certain point performance is limited by the cost of the algorithms employed and the speed of the processor. For faster-than-real time evaluation, it is recommended that the GUI be turned off. Default value is 1.0.
logFileName logVerbose logToSystemOut	The name of a file (if any) to be used for logging the event data from the simulator. Default value is RP1.log. The logVerbose setting controls whether verbose logging is turned on. Verbosity is useful for debugging both client and simulation processes, but does degrade performance. Default is false. And logToSystemOut setting controls whether the output is sent to system error. Note that simultaneous logging to both a file and system console is allowed. Default is false.
dlcEnabled dlcName dlcSetIO dlcSetLogger	The RP1 simulator has the ability to load the class (binary) files for a client at run-time (dynamically) based on the name of the client class. If the boolean value dlcEnabled is set to true, it will search the CLASSPATH for a class with the name specified as the dlcName. By default, dlcEnabled is set to false. When specified as true, the client name is mandatory.
enableInterlock	Enables the use of the "interlock" features. Often useful when timing issues are confused by system scheduling and communications overhead, especially when running at a faster-than-real-time simulation speed.

3.2 Accepting Clients

The simulator accepts clients via TCP/IP connections as described above. By design, the system should be able to accept multiple clients. At present, it can accept multiple client connections, but does not correctly model the interactions between robots. A complete implementation of this feature is planned.

3.3 Dynamically Loading Clients

When server is configured to dynamically load clients, it will do so shortly after startup. In future releases, the RP1 GUI will provide a file-browser interface that allows you to find and load clients dynamically. For now, the only way to specify clients is through the RP1.ini file (only a single client is supported at this time).

You may optionally configure a dynamically loaded client to connect to the server through the network TCP/IP connections as described above. If effect, this results in a program connecting

to itself. A more efficient means of communication is to allow the server to establish an internal I/O connection for the client. This option is controlled by the "dlcSetIO" specification in the RP1.ini file. Letting the server set the I/O for a dynamically loaded client is recommended whenever possible.

You may also configure a dynamically loaded client to write its logging information to the main server log using the dlcSetLogger option.

3.4 Interlock

The "interlock" option in RP1 provides a way of ensuring rigid synchronization between the simulator clock and its client applications. This option is mainly intended for applications that are sensitive to timing issues. The interlock feature especially useful in cases where users the simulator is running at an accelerated clock rate (in which the simulator time moves more quickly than real time). It will also be useful in cases where some of the client's operations require a very large, and time-consuming, amount of processing and occasionally prevent the client from keeping up with the simulator.

To understand the rationale for the interlock option, consider a case where the simulator is set to run very quickly, perhaps 100 times faster than the real time clock. At these speeds, the system overhead related to task management and inter-process communications has a substantial effect on the simulator. For example, suppose the server issues a "sensor/detection event" to which the client is supposed to respond with a "stop motion request." If the simulator is free to run ahead, the real time delay due to system and communications overhead may result in several minutes of simulation time passing before the client ever responds... even though from the client's perspective it replied almost instantly.

In the interlock mode, the simulation clock is suspended whenever the server sends an event to the client. The clock is not re-started until the client process replies with an "interlock acknowledgement." Both the RP1 Java and C/C++ client API's implement code so that the client does not respond with an acknowledgement until all event handlers related to the simulator-issued event have been invoked and have completed their operations. Thus, we can ensure that all processing related to an event will be completed before the simulator is enabled to continue.

3.4.1 Overhead Related to the Interlock Option

Clearly, the selection of the interlock option means that more messages have to be sent between the client and the server. This extra overhead is the main cost of using the interlock option. Recall that, because RP1 messages are generally quite small (less than 200 bytes), the major cost associated with communications is mainly the number of transactions exchanged, not the amount of data moved. In general, though, the extra overhead for using interlock should be acceptable.

3.5 Internal Architecture

3.5.1 The Threads

One of the strengths of the Java language is the ease with which it can be used to code multi-threaded applications. Table 6 lists the primary threads implemented in the RP1 server.

Table 6 – Primary Threads in RP1 Server .

Scheduler Thread	The scheduler thread is the main simulation thread. This thread creates the simulator's virtual clock and also performs all the computation and modeling required to drive the simulation
AWT Thread	The AWT thread is launched by Java's graphics environment, the Abstract Window Toolkit (AWT). The implementation makes a very straight-forward use of the Java tools
Client-Listener Thread	The client-listener thread accepts connections from client applications, creates client objects and launches new client monitor threads. Optionally, you may disable this thread if you are using Dynamically Loaded Clients.
Client Threads	The Client Thread services *incoming* communications from clients. Each client is assigned a unique thread. When network connections are enabled, Client Threads are launched by the Client-Listener Thread when a new client connection is accepted. When Dynamically Loaded Clients (DLC's) are used, Client Threads are launched by the Scheduler Thread shortly after startup.[3]

[3] Dynamically Loaded Clients are launched by a task which is added to the scheduler shortly after start up. Future revisions of RP1 will feature a browser-style interface that will allow the user to launch a DLC interactively. The class SimClientLauncherTask was designed with this use in mind.

3.5.2 The Scheduler

The scheduler is the heart of the simulator. Essentially, the scheduler implements a task queue not much different from a classic printer queue or batch-processing queue. In the simulator, events are treated as tasks and kept in a list sorted by time. The tasks are processed serially based on their time values. Time is treated using floating-point values for intervals specified in seconds.

Consider the following list which show the potential state of the queue at some time.

time 0.000	start robot motion for 10 seconds
time 0.100	evaluate robot position and disposition
time 0.200	evaluate
time 0.300	evaluate
time 0.301	halt-motion due to collision with wall.

At time 0.000 the client requested a robot motion with a duration of 10 seconds. The simulator determined that after just 0.301 seconds, the motion would result in a wall collision, and so scheduled a halt task at that time. It then queued up 5 tasks as shown in the table. The spacing of $1/10^{th}$ second between tasks reflects the default modelingFrameRate of 10 frames-per-second (see above).

As tasks are performed in sequence, the internal clock is adjusted to the time of the task. In most cases, individual tasks require far less time than the interval allotted. To simulate real-time behavior, the scheduler often introduces waits during which the scheduler is idle and other processes can be completed. In situations where the simulator is instructed to run at faster than real time, the waits are shortened or removed entirely.

At each evaluation, the simulator has the potential to send back information to the client (based on the results of the evaluation). Suppose that at 0.200 seconds, a proximity sensor detected the wall and the client sent a motion-halt request to the simulator. The client thread would create a task to perform a motion-halt operation and would place it at the *head* of the queue in a priority mode. When the halt task was serviced, all remaining motion tasks would be removed from the queue causing the robot simulacrum to halt.

The description of the task queue is algorithmic rather than practical. Certain real-world considerations complicate it somewhat. For example, at the default modelingFrameRate, a one-minute motion requires 600 tasks. Queuing up so many tasks is not a good use of memory or processor cycles. This is especially true when the sequence might be cut short by a client request. So, typically, the simulator queues up only one motion task at a time. Motion task objects have the ability to "recycle" themselves so that they are placed back on the queue (with an adjusted time value) when they complete.

The virtual clock is coupled to the passage of real time. If a client or some other process requests the "simulation time" while the scheduler is resting between tasks, the scheduler will consult the system clock to derive a reasonable value.

Finally, a word on the Java thread scheduling mechanism. Under Windows architectures, threads are allocated a 50 millisecond slice of time to perform various operations. This approach has consequences for applications such as the simulator that depend on timing considerations. For example, consider the case where the code execute two requests for the system time separated by a 5 millisecond wait:

```
long time0, time1, deltaValue;
time0      = System.currentTimeMillis();   // current time in milliseconds
wait(5);                                   // wait 5 milliseconds
time1      = System.currentTimeMillis();
deltaValue = time1-time0;
```

The delta value for the times will probably be close to 50 milliseconds, rather than the 5 milliseconds you might expect. The reason for this is that when the wait is executed, control may be transferred to another thread. Even though the 5-millisecond wait expires, as much as 50 milliseconds may pass before the scheduler returns control to the waiting thread.

Due to this scheduling mechanism, the simulator will often run at slower-than-real-time speeds when we chose modelingFrameRates of greater than 10 Hertz. This is not necessarily a bad thing. It does produce a more accurate model and also washes out some of the timing considerations due to the overhead for client-server communications.

3.6 The Floor Plan

One of the first things the server does on start-up is to load data for the simulated environment. This data is stored in files called "floor plans." Floor plans include data describing the physical layout of walls and other features. Floor plan data is encoded in text-files. At start-up, the Server consults the simulation property file to obtain the name of the desired file. It then reads and parses the data in that file to create a virtual landscape for modeling. If the GUI option is enabled, the display will display a "plan view" of the exercise area similar to that shown in Figure 2.

The floor plan shown in Figure 2 is from the sample file "WhiteRoom.txt" which is supplied as part of the Rev 0.6x RP1 software distribution. Figure 3 shows the code and specifications from which that floor plan file was generated.

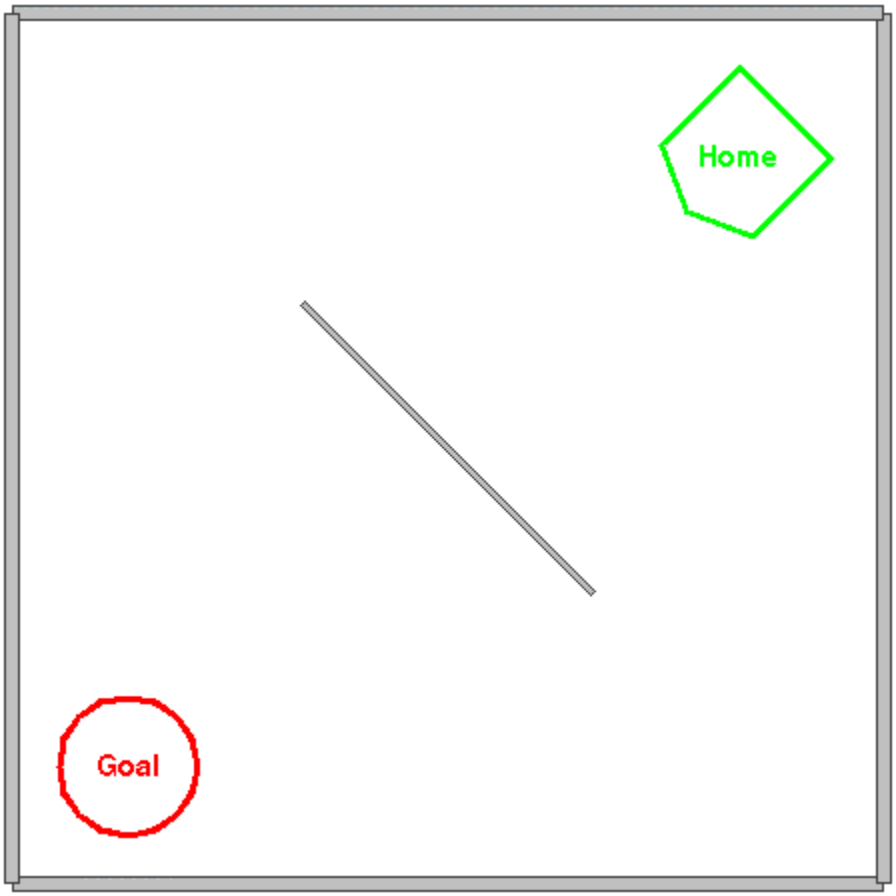

Figure 2 – Floor plan from WhiteRoom.txt example

```
/*  White Room

Sample floor-plan file for Rossum's Playhouse
This encoding is based on the RP1 rev 0.4 floor plan format.

*/


units: meters;
caption: "White Room (RP1 rev. 0.4)";

wall a { geometry:   0.0,   0.0,     3.0,   0.0,   0.05;  }
wall b { geometry:   3.0,   0.0,     3.0,   3.0,   0.05;  }
wall c { geometry:   3.0,   3.0,     0.0,   3.0,   0.05;  }
wall d { geometry:   0.0,   3.0,     0.0,   0.0,   0.05;  }

wall e { geometry:   1.0,   2.0,     2.0,   1.0,   0.025;}

target goal {
  label:     "Goal";   // F is for fire
  geometry:   0.4, 0.4, 0.25;
  lineColor:  red;
  lineWidth:  3;
}

placement home {
  label:     "Home";
  geometry:  2.5, 2.5, 225, 0.275;
  lineColor:     green;
  lineWidth: 3;
}
```

Figure 3 – Contents of "WhiteRoom.txt" floor-plan file.

3.6.1 Syntax and Semantics

The specification of floor plans is based on an very simple grammar. The grammar supports two kinds of statements: specifications and declarations. Specifications are used to supply parameters such as what system of units is to be used for the floor plan or what geometry is to be used for a particular object. Declarations are used to create particular objects such as walls. A specification is a simple statement given in the form:

```
specification:   parameter;
```

or

```
specification:   parameter[1], …, parameter[n];
```

while declarations have a more complex syntax:

```
objectClass  objectName {
    specification[1..n];
}
```

and may contain one or more specifications. Revision 0.60 supports seven kinds of object declarations: walls, obstacles, targets, placements, floor paint, and navigation nodes and navigation links.

3.6.1.1 Specifications

The general specifications in a floor plan (those not included in a declaration) tell something about the overall floor plan. At present, two are supported: caption and units.

The caption specification allows you to specify a caption to be placed on the main frame (window) of the GUI.

The units specification can be either meters, feet, or kilometers. Internally, all units are converted to meters, but the system specified in the floor plan is used for display purposes. The caption specification is used for labeling the main application window (Java frame). No other general floor plan specifications are supported at this time.

3.6.1.2 Declarations

As mentioned above, declarations are used to define objects. Different kinds of object declarations contain different specifications. The following table lists the specifications that can be included in any kind of declaration.

Table 7 – Specifications Used for All Types of Object Declarations.

Specification	Description
fillColor	For those objects which are depicted using filled-polygons, the fillColor can be used to specify the interior color. Each object type has its own default value.
lineColor	Objects depicted using lines or text are drawn in the line color. A filled-polygon may also use this specification to indicate that it is to be depicted with a separate outline color.
color	The color specification sets both the fill and line colors and can be used whenever convenient
label	For objects which include a label, this specification indicates the string to be used.

3.6.2 Walls

In RP1, walls are simple barriers. They are specified as a set of two endpoint coordinates and a wall thickness. The following example shows a declaration for a wall:

```
wall a {
    geometry:   0.0, 0.0,    98.0, 0.0,   0.75;
}
```

The specifications for the wall geometry require 5 real-valued parameters. They provide two-coordinate pairs giving the end points and a value for the thickness of the wall in the order shown:

x coordinate of first endpoint,
y coordinate of first endpoint,
x coordinate of second endpoint,
y coordinate of second endpoint,
thickness.

All geometry parameters are assumed to be in the specified units system (by default, meters).

If a robot simulacrum collides with a wall, all motion will be halted and an RsMotionHalted event will be sent to the client indicating the cause of the halt. Robot simulacrums can detect walls using the range-sensor component included in the standard tool set.

3.6.3 Obstacles

Functionally, obstacles behave exactly as walls, except that they may have arbitrary shapes. The RP1 floor plan format now permits the specification of an obstacle as shown in the example code fragment below:

```
obstacle ArbitraryName {
     geometry:     0.0, 0.0,   1.0, 0.0,   0.5, 0.86;
     offset:       1.0, 1.0;
     orientation:  180;
     color:        orange;
}
```

The obstacle specification above describes the coordinates for an equilateral triangle positioned at offset (1.0, 1.0), and rotated 180 degrees. You may specify any number of *coordinate pairs* for the geometry as long as it describes **a simple, non-self-intersecting polygon**. The offset and orientation specifications are optional and are intended mainly as a convenience.

In place of the *geometry* specification, you may include a *polygon* specification:

```
obstacle AnotherArbitraryName {
    polygon:  36, 1.0;
}
```

The above declaration describes a 36-sided polygon of radius 1.0. The initial polygon is centered at (0.0, 0.0) but may be adjusted by supplying an offset. The initial point is located on the x-axis at coordinate (1.0, 0);

3.6.4 Targets

Targets are meant to model point sources of light or infrared radiation. Essentially, they give the robot a point target that it can detect and identify. The robot component objects include a "target sensor" which can be detect targets and generate sensor events (see "Communication via Events and Requests" above).

The target geometry is specified as

x coordinate of target,
y coordinate of target,
radius for surrounding circle (purely for human reference).

Other specifications include a label, the color, and the thickness (width) of the line used to draw the surrounding circle.

3.6.5 Placements

A placement is used to supply a starting position and orientation for the robot simulation. If desired, there can be more than one. When the client program wishes to place the robot into the simulation environment, it issues a "request for placement" specifying the name of the placement (note that the object name is used, not the label string). If no name is specified, a placement is chosen at random. The server responds with an RsPlacementEvent giving the position and orientation of the robot. The robot cannot otherwise detect or interact with placements.

The placement geometry is specified as:

x coordinate of center (robot placement position),
y coordinate of center,
orientation for robot (in degrees),
approximate radius used to develop the "home-plate" style icon.

3.6.6 Floor Paint

The floor-paint feature allows you to create areas of colored polygons on the base of the floor plan. Simulated robots may be equipped with sensors that allow them to detect floor paint when the sensor is positioned above it.

Floor-paint geometry is specified exactly the same way as the geometry for the obstacle feature (see 3.6.3 above). As with the obstacle feature you may create either an arbitrary **non-self-intersecting** polygon using the "geometry" specification or a regular polygon of n-sides using the "polygon" specification. An example paint declaration follows:

```
paint ArbitaryName {
        geometry:     0.0, 0.0,   1.0, 0.0,   0.5, 0.86;
        region:       1;
        color:        blue;
}
```

The color specification is strictly for depiction purposes. The *region code*, however, can be used by the RP1 client application for a number of purposes. It allows the user to mark the paint region as having a particular meaning. When the simulator sends a paint-sensor event to the client application, it includes the region code in the event. The client may then respond to the region code in whatever manner the developer deems appropriate. For example, some sensors may be sensitive to some colors of paint while insensitive to others. Since color is used only for depiction purposes, it is not available to the client. To interpret color, however, the client may use the region code to determine how a sensor responds. Alternatively, a client might need to treat the robot's behavior as being different in various regions of the simulator. The region code in the floor paint feature provides one way determining where the robot is and how it is to behave.

3.6.7 Navigation Features: Nodes and Links

The node and link features were introduced to RP1 in Rev. 0.48 as a way of allowing a floor-plan to specify a network of "roads and intersections." The motivation for these features is to provide the user a method of specifying data for navigation purposes. Because the layout of an RP1 environment is arbitrary, some developers may require a method to specify a selection of pathways as part of their floor plan. Such a specification can be provide through the use of the navigation features.

While a navigation network is not appropriate for all applications[4], some simulation clients may benefit from the ability to download a floor plan and dynamically constructing procedures for operating in their environment. One such application, the AutoPilot demo, is supplied with the standard RP1 Java source code distribution. Figure 4 below shows the floor plan for the Trinity College Fire-Fighting Home Robot Contest overlaid with a navigation network. Although the network shown is inadequate for efficiently running the contest, it does illustrate the concept of "roads and intersections" modeled as a network of nodes and links.

[4] A robot intended to explore and map its environment would have no use for a pre-determined navigation network.

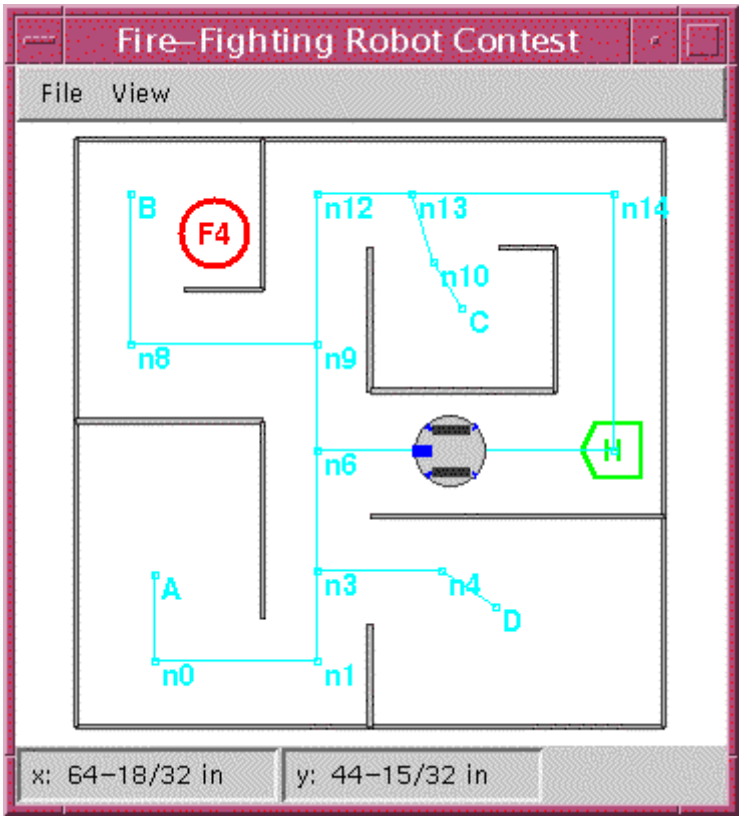

Figure 4 – Trinity Contest Floor Plan with Navigation Network

The following code fragment shows the specification of two nodes and a link that connects them:

```
node n0           { geometry:  13.0, 11.0; label: "n0";  }
node n1           { geometry:  40.0, 11.0; label: "n1";  }

link p0  { nodes:    n0,   n1; }
```

The geometry specification for each node is mandatory. The label is optional. A link connects exactly two nodes. A node can be included in any number of links. RP1 always renders links as line segments. RP1 does not implement logic to determine if links intersect.

At present, the RP1 API does not include a tool kit for manipulating or analyzing the navigation net. Although the AutoPilot implements a limited, ad hoc., solution, it is far from the full-featured, general-purpose tools available for network analysis. The navigation network provides a source of data. Our hope is that in the future, Rossum Project developers will create tools for the analysis of that data.

3.6.8 A Mapping Tool for Developing Floor Plan Specifications

When the Rossum Project was originally started, we hoped that eventually someone would write an interactive authoring tool for creating floor plans. Developing a floor plan using the text format specified by RP1 is tedious, at best. While no such work was ever completed, we are pleased to be able to report that a mapping tool has become available as part of another software initiative. Author Shane O'Sullivan has created a CAD-like tool called Map Viewer that makes it possible to construct floor plans quickly and easily. The Map Viewer application also includes support for alternate robot simulators such as CARMEN, Saphira, and Player/Stage.

Shane developed Map Viewer to support his own robotics research and many of its features reflect that noble pedigree. For example, he includes functions for generating Voronoi diagrams from occupancy grids, generating a parameterized path using a modified A* algorithm, converting grid maps (images) to vector-based maps, etc.

To learn more about Map View visit http://mapviewer.skynet.ie/

4 Building a Virtual Robot

4.1 Introduction

In the Rossum's Playhouse simulation environment, the way to build a virtual robot is to write a client application. This section tells you how to do it.

The point of Rossum's Playhouse is to help the user to test logic that will eventually be integrated into an actual robot. Naturally, doing so requires some method of configuring the simulated robot so that its characteristics follow those of the actual hardware being studied. In the current revision, the information to do so is provided by the client. Paragraph 4.7, *The RsBody and RsBodyPart Classes,* below describes the components that can be used to create a robot simulacrum.

In the real world, building a robot isn't much fun unless you actually run the thing. And in the simulation, a specification for a robot body is rather pointless until you build the logic that lets it play. This section also shows how to write code that enables your simulacrum to interact with its environment. It does so by providing a tour of "ClientZero," one of the two demonstration client applications that are provided with the RP1 software distribution. ClientZero is a simple application that was written expressly for the purpose of providing an example for developers implementing a client.

4.1.1 Thinking about Client Design

Before we talk about the demonstration clients, we want to make an important point about client design in general.

We think that the code for a well-designed client ought to have a life of its own, separate from the simulator. We've already mentioned our hope that the code that you test on the simulator can be ported to an actual platform with little modification. Even if your target is not actual hardware (perhaps you are writing a general-purpose navigation tool kit), you should strive to minimize dependencies on the simulator.[5] The more successful you are at doing this, the more useful your code will be to yourself and to other developers.

Of course, the means of accomplishing this goal are not at all obvious. Certainly, there is no one "right" way to design a client application. Figure 5 below shows the broad outlines of an approach which may be useful. The figure is purely conceptual, and does not illustrate any actual packages, classes, or libraries. In the figure, the high-level logic that makes up the true "brains" of the robot/client is separated from direct dependency from both RP1 and its intended robot platform. In the simulator version, it plugs into a block of interface code which handles issues specific to RP1. In the robot-hardware version it plugs into a block of mid and low-level control logic. The software for simulator interface and the control blocks are unique to their particular implementations. The high-level modules can be shared between both.

[5] This may be the only time you ever hear a software vendor saying "whatever else, don't lock yourself into our product."

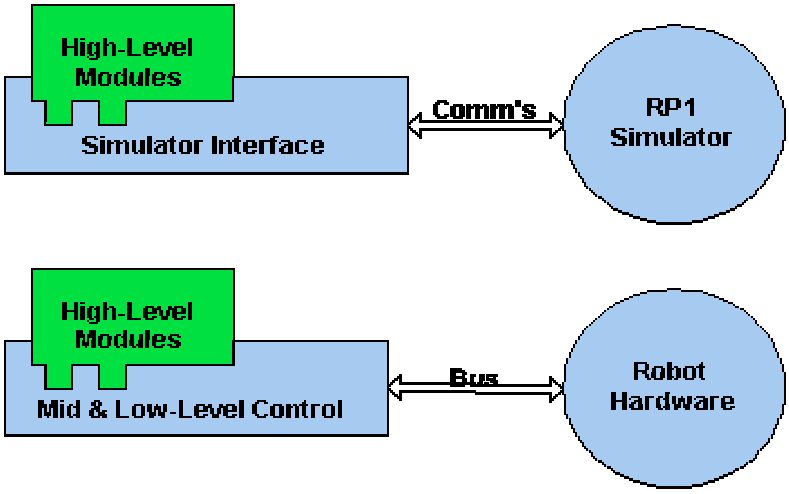

Figure 5 – Design Elements for Avoiding Code Dependency

Incidentally, don't let the placement of the modules blocks in the diagram mislead you into assumptions about the flow of control through the system. Although the high-level modules are positioned on top of the simulator-interface and mid-and-low-level blocks in the diagram, it does not imply that they are actually running the robot. Depending on your needs, you may want to treat them as subordinate functions and let the low-level functions "do the driving."

4.2 The Demonstration Clients

The two demonstration clients supplied with the RP1 software distribution are called ClientZero and DemoZero. ClientZero is a very simple application that performs three basic operations: it connects to the server, it supplies a body definition, and then interacts with events received from the simulator. It has no user interface, but does keep a log of its interactions with the server. All the source code for ClientZero is stored in the Java package rp1demo.clientzero (recall that a Java package corresponds directly to a system directory).

DemoZero adds a Graphical User Interface (GUI) to ClientZero, but does not otherwise extend its functionality. In fact, the DemoZero client is built on the ClientZero code. It does not change the underlying logic of the "robot's brains." It simply adds features (the GUI) that are useful to the user running a simulation.

In doing so, it illustrates a point. You may add things to your simulator (client) code that do not have to go into your target robot. DemoZero piggybacks a GUI on top of the ClientZero control logic, but does not alter the ClientZero behavior. Actual clients might extend their robot logic by adding user-controls, performance analysis, or better simulation logging. Remember, though, that an important goal of the simulator is to be able to reuse the code that you test in the simulation environment in an actual platform. The key to being able to do this is to separate

those components that are specific to the simulator from those which go into both the RP1 client and the real-world robot. The DemoZero GUI does not interfere with the ClientZero logic, it simply provides the user with some auxiliary controls.

4.3 Life Cycle of the Demonstration Clients

Both demonstration clients follow the same life cycle:

1. initialize its session with the server (establish network or local connection if necessary);
2. send body plan to server;
3. register relevant event handlers;
4. enter an event-loop.

In the first step, the RP1 client contacts the server and exchanges introductory information. To do so, a client running as an independent client must make a network/local connection (see paragraph 2.4 *Configuration Elements and Properties Files* above). A dynamically loaded client simply accepts the I/O streams provided by the server.

In the second and third step, the client tells the server about the robot it wishes to model and then registers event handlers to manage the communications from the RP1 server (which come in the form of events, see paragraph 2.3.2 *Communication via Events and Requests* above). It is necessary to send the body plan before registering event handlers because many of the events that come from the server depend on components of the virtual robot.

Finally, the clients enter an event loop in which incoming communications are received, processed, and passed to the client-supplied event-handlers. Events and event handlers are discussed in more detail in Section 5 below.

Figure 6. below shows source code for a simplified version of ClnMain.java. We will refer to it in the discussion that follows.

```
package rp1demo.clientzero;
import  rp1.rossum.*;
import  rp1.rossum.event.*;
import  rp1.rossum.request.*;

import java.lang.*;
import java.io.*;

public class ClnMain extends RsClient implements RsRunnable
{
   public static void main(String args[]) throws IOException
   {
      ClnMain c = new ClnMain();
      c.initialize();   // throws IOException if unable to reach server
      c.run();
   }

   public ClnMain(){
      super();
   }

  public void initialize() throws IOException {
      // to connect to the server, invoke the initialize method from
      // RsClient, the super class (parent class) of ClnMain
      super.initialize();

      // create body and send its specification to server -------------------
      body = ClientZero.build();
      sendBodySpecification(body);

      // register event handlers ---------------------------------------------
      addMouseClickEventHandler(new ClnMouseClickEventHandler(this));
      addPositionEventHandler(  new ClnPositionEventHandler(this));

      addPlacementEventHandler(   new ClnPlacementEventHandler(this));
      addMotionHaltedEventHandler(new ClnMotionHaltedEventHandler(this));
      addTargetSensorEventHandler(
            (RsBodyTargetSensor)(body.getPartByName("head")),
            new ClnTargetSensorEventHandler(this));
      addTargetSelectionEventHandler(new ClnTargetSelectionEventHandler(this));
      addTimeoutEventHandler(   new ClnTimeoutEventHandler(   this));
      addPlanEventHandler(      new ClnPlanEventHandler(      this));

      // send a request for a floor plan (used to make a navigation network)
      sendPlanRequest();

      // request that the robot be placed on the floor.
      sendPlacementRequest("home");
   }
```

Figure 6 – Source Code for ClnMain (modified for clarity)

4.4 How ClnMain Extends RsClient and Implements RsRunnable

When you examine the class definition for ClnMain, note the statement *class ClnMain extends RsClient implements RsRunnable.* The ability of one class to extend another is a hallmark of object-oriented programming languages. ClnMain extends a class RsClient which is defined in the java package *rp1.rossum*. In doing so, it inherits all the capabilities of RsClient
So what is RsClient? The RsClient class provides an RP1 client with its interface to the simulator environment. In includes I/O channels, methods for encoding and decoding communications, registering event-handlers, and running the client side of the simulator session. Without some instance of an RsClient object, it would be quite difficult for a client to communicate with the server.

And what about RsRunnable? RsRunnable is an interface. In Java, an interface is a way of specifying that a class will implement a certain set of methods (functions). If a class statement says that it will implement a particular interface, we know that those methods will be available for any instance of that class. Armed with this knowledge, we can use any object of that class, or any object of any other class that implements that interface, interchangeably.[6]

For example, the Java standard Runnable interface implements a method called run(). Because Java knows that any class which implements Runnable includes the run method, Java knows that such a class can be launched as a separate execution thread. RsRunnable is simply an extension of the Java Runnable class. In addition to run(), RsRunnable specifies a few other methods which permit any class that implements it to be treated as a dynamically loaded client (see paragraph 3.3, *Dynamically Loading Clients*). RsClient implements RsRunnable. It was not strictly necessary to include the reference to RsRunnable in the declaration for ClnMain. When ClnMain inherited the properties of its super class, RsClient, it also inherited any of the associated interfaces. We included the "implements RsRunnable" in the declaration for clarity.

4.4.1 The RsRunnable Interface

To load a class as a Dynamically Loaded Client, the RP1 simulator requires that the class implement four methods. At run time, these methods are invoked in the sequence shown in the table below:

setInputOutputStream	allows the server to bypass the network (socket) based communications by supplying I/O
setLogger	allows the server to supply log-keeping information to the client
initialize	tells the client to perform its main initializations; if I/O is not set, the client should establish a connection to the server
run	from Java's Runnable interface, allows the client to be run as a separate thread; this method is the main run-loop for the client.

[6] If one of your class definitions specifies that it implements a certain methods, but you forget to include the method in the class, it will not compile. Thus a Java interface ensures that a class does, in fact, implement all the methods it claims. Of course, determining whether your methods actually work is beyond the scope of the compiler.

The RP1 server always invokes initialize and run, but can be configured to not invoke setInputOutputStream and setLogger. In addition to the methods specified by the RsRunnable interface, RP1 also has a requirement about the constructor:

constructor()	RP1 always invokes a constructor which takes *no arguments*; when you build a Dynamically Loaded Client, make sure that you provide any necessary functionality within such a constructor.

The methods defined by the RsRunnable interface are coded as follows:

```
package rp1.rossum;

public interface RsRunnable extends Runnable {
   public void setInputOutputStreams(InputStream input, OutputStream output);
   public void setLogger(RsLogInterface logger);
   public void initialize() throws IOException;
   public void run();   // actually inherited from java.lang.Runnable
}
```

4.4.2 Building RsRunnable and RsClient into ClnMain..

When we implement an RP1 client, we need to ensure that it includes an instance of the RsClient class so that it may communicate with the server. When we implemented ClnMain, we had two choices. ClnMain could include RsClient object as an element, or it could inherit all the capabilities of RsClient by extending it.

In earlier version of the demonstration software, ClnMain actually did include an object of class RsClient. In the present implementation, it ClnMain extends RsClient. This use of a derived-class follows one of the elegant conventions of object-oriented programming. By extending RsClient, ClnMain inherits all the methods and elements RsClient. Thus, ClnMain can use all the functions and capabilities of the parent class as if they were its own. The approach saves coding and provides a convenient framework for our client implementation.

So why didn't we implement ClnMain as a derived class of RsClient in earlier versions of the demonstration? Well, in earlier versions, we were concerned that by deriving ClnMain from RsClient, the example code would give a false emphasis to the importance of the Rossum environment in building robot software. We thought that if we implemented our main class so that it was "just a derived class of a Rossum fixture" that it would suggest that any client implementation is hard-wired into the RP1 environment. One of the recurring themes in this document is that your robots are more important than our simulation. In our own robotics work, we are careful to separate the code need to write so that we can work with the simulator from that which we need for our robots. Ideally, simulator-interface code like ClnMain should not contain any code related to the real problem of implementing a robot.

4.4.3 DemoMain Implements RsRunnable, But Does Not Extend RsClient

Earlier, we mentioned the DemoZero application that piggybacks a GUI onto ClientZero. Like ClnMain, the DemoMain class implements the RsRunnable interface. It does not, however, extend RsClient (nor does it extend ClnMain, though that would have been an elegant way to implement it). We wrote it as a non-derived class to illustrate the fact that extending RsClient is not the only way to implement a class that can run as a Dynamically Loaded Client.

Instead of extending RsClient, DemoMain contains an object of type RsClient that it uses for its communication with the server. To satisfy the RsRunnable interface, it provides simple "pass through" methods. The source code for DemoMain is shown below. Note that an object of type ClnMain (called "client") is used to support the connection to the RP1 simulator.

```
package rp1demo.demozero;
import  rp1demo.clientzero.*;
import  rp1.rossum.*;
import  rp1.rossum.event.*;
import  rp1.rossum.request.*;

public class DemoMain implements RsRunnable
{
   ClnMain  client;

   // implement a main to allow this class to serve as an Application -----
   public static void main(String args[]) throws IOException {
      DemoMain demo = new DemoMain();
      demo.initialize();   // throws IOException if unable to reach server
      demo.run();
   }

   public DemoMain(){
      client = new ClnMain();
   }

   public void setInputOutputStreams(InputStream input, OutputStream output){
      client.setInputOutputStreams(input, output);
   }

   public void setLogger(RsLogInterface logger){
      client.setLogger(logger);
   }

   public void initialize () throws IOException {
      client.initialize();

      DemoFrame  demoFrame = new DemoFrame(client);
      demoFrame.show();

      client.setProtocolShutdownHandler(
           new    DemoProtocolShutdownHandler(demoFrame));

      client.addMotionStartedEventHandler(
          new DemoMotionStartedEventHandler(demoFrame));
      client.addMotionHaltedEventHandler(
          new DemoMotionHaltedEventHandler(demoFrame));
      client.addPositionEventHandler(
          new DemoPositionEventHandler(demoFrame));
      client.addTimeoutEventHandler(
          new DemoTimeoutEventHandler(client));
      client.addPlacementEventHandler(
          new DemoPlacementEventHandler(demoFrame));
      client.addTargetSensorEventHandler(
         (RsBodyTargetSensor)(client.getBody().getPartByName("head")),
         new DemoTargetSensorEventHandler(demoFrame));

      client.sendTimeoutRequest(1000);
   }

   public void run(){
      client.run();
   }
}
```

Figure 7 – Source code for DemoMain.java

4.4.4 The Execution of ClnMain

Anyway, ClnMain inherits all the features of RsClient. RsClient itself, implements RsRunnable. We included the statement "implements RsRunnable" in the declaration as a reminder that the interface was part of the class. Since RsClient already includes all the methods in the RsRunnable interface, we don't have to bother with them except where we have special needs.

The first method defined by the class ClnMain is a static main(). This method is a standard Java technique that allows you to launch ClnMain as an application by typing:

```
java rp1.clientzero.ClnMain
```

at the command line.[7] Referring to the code, note that the main() method creates an instance of an object of type ClnMain (by using the Java keyword "new"), and then invokes the initialize() method that ClnMain inherited from its super class RsClient. The RsClient.initialize() method is provided as a convenient way of initializing a connection with the server.

When ClnMain is launched from the command line (or a script, or a Windows .bat file), it runs as a separate program from the RP1 simulator. Java invokes the main() method, which in turn invokes the initialize() method that ClnMain inherited from RsClient. The RsClient.initialize() method recognizes that it does not have a connection to the server and sets out to establish one. If it fails (as when the RP1 server is not running), it will throw an IOException, in this case terminating the program. RsClient.initialize() also loads the properties file for ClnMain (which is included in the clientzero package). If the properties files specifies that it should do so, initialize() opens up a log file.

After the initialize() method is completed, ClnMain.main() invokes run(). The run method, which is inherited from RsClient() transfer control to an event-loop, is which the client receives communications (events) from the server and uses them to invoke event-handlers. Control does not return from the event-loop until the connection with the server fails (as when the server shuts down) or when the application is terminated.

When ClnMain is treated as a dynamically loaded client, the main method is not used. Instead, RP1 loads the ClnMain class and *creates an instance* of that class (creates an object). It then uses that object to invoke the methods that were specified in the RsRunnable interface. The first of these, setInputOutputStreams() tells the client object that it does not need to establish I/O. The second, setLogger() suppresses the usual logging for the client and redirects it to the main simulator log.[8] Once these preliminaries are complete, it invokes the object's initialize() method. The initialize() method that ClnMain inherited from RsClient is smart enough to check whether I/O and logging have been specified and to use them rather than attempting to set up its own functions.

---

[7] *Applets*, which are launched from a Browser, behave slightly differently than *Applications* which are launched directly from the system or command line.

[8] Both setInputOutputStreams() and setLogger() can be suppressed using configuration elements in the RP1.ini properties file. This feature is intended for testing purposes (so that a dynamically loaded client can exercise its ability to establish a network connection, or so that it may be configured to keep an independent log file).

4.4.5 Uploading the Body Plan

Once the connection is established, the next thing the client needs to do is to give the server a physical description of kind of robot it will be modeling. It does so by creating and uploading a body plan. For demo purposes, the body plan is created using a static method in the class ClientZero.

The RP1 rossum package defines a class called RsBody that is used by the simulator to model the robot body. The RsBody class provides an easy way for defining and transmitting a body plan to the simulator. This approach has another advantage in that some of the logic that the simulator uses also has value in client applications. We will get into the details of the body plan later on when we look at the ClientZero class. For now, it is worth noting that ClientZero implements a round body with a target sensor at its front.

4.4.6 Registering Event Handlers

The RsClient class, which is used by ClnMain, allows applications to run in an "event loop" which allows them to communicate with the server.

Event loops are a very common feature in many graphics environments. Developers who have had experience with Unix and X-windows will recognize the origins of the RP1 concept immediately. Those who have worked with the Java AWT will see a similar parallel with the ideas of "event listeners" and the Java graphics thread (which is a kind of event loop).

Of course, a robot control system is *not* the same thing as a computer graphics application. But the event-loop concept used in many graphics applications turns out to be quite adaptable to other purposes. As you examine the code for clientzero, you will see how this is accomplished.

To use the event loop, clients register "event handlers" which are methods (or "functions") that are invoked when the client receives an event (message) from the server. Once ClnMain invokes the RsClient.run() method, it enters an endless loop in which RsClient receives messages from the simulator/server and invokes the methods that were registered by the client application. Recall that the client communicates to the server by sending it "request" messages and that the server responds by sending the client "event" messages. An event might be something like "the robot hit a wall," or "the robot's sensor detected a light source," or "a timer expired." When these event messages are received, any registered event handlers are invoked

Clients may register event handlers at any time. The ClnMain.java function registers most of them before it enters the event-loop, but only because it is convenient to do so. It is also possible to register multiple event handlers for any event. The Demo application which is also included in the RP1 software distribution adds a GUI to the Client application by simply piggybacking additional event handlers to those already registered by ClnMain.

4.4.6.1 Multiple Event Handlers and the RsEvent.consume() Method

When multiple handlers are registered for a particular event, they are invoked in the order in which they were registered. Sometimes, it is useful for one event handler to be able to prevent the event from being passed on to other event handlers. It may do so by invoking the consume() method of the current event.

4.4.7 Running the Event Loop

Once the ClnMain.initialize() method registers event handlers, it sends the RP1 simulator a request for a placement on the floor plan. Up until the time the simulator receives the request, the robot is not visible on the display. Once RP1 receives and processes the request, it places the robot simulacrum on the floor plan and issues a placement event.

ClnMain will not process the incomming events until it enters its event-loop. The event-loop is embodied in the RsClient.run() method. Since ClnMain extends RsClient, it inherits that method. The run() method implements an endless loop in which incoming events are processed and the corresponding event handlers are invoked. The choice of the name "run()" for the event-loop method in RsClient is not an accident. It allows client applications to take advantage of Java's multi-threading (multi-tasking) feature.

4.5 How the Demo Clients Work

The demonstration clients take advantage of a feature of the simulator that has no counterpart in the world or real robotics: the mouse click event. For demonstration purposes, the mouse click is used to tell the robot where to go. When you point the mouse in front of the robot and click the left button, the robot will move toward the position you indicate. If you point the mouse somewhere else, and click the right button, the robot will turn to face the indicated direction.

Referring back to the code for ClnMain, note that the first event-handler statement

```
addMouseClickEventHandler(new ClnMouseClickEventHandler(this));
```

establishes a mouse-click event handler. ClnMouseClickEventHandler extends the class RsMouseClickHandler which is included in the rossum package. When you click the mouse button, the Server sends an event message to any clients that have registered event handlers of mouse clicks. ClientZero responds by performing some simple navigation computations and moving toward the click point..

ClientZero is not especially smart. It will go where you direct it and eventually may crash into a wall. When this happens, its motion will be halted and the server will issue an RsMotionHaltedEvent.. Motion-halted events are handled by ClnMotionHaltedEventHandler which extends RsMotionHaltedEventHandler.

```
addMotionHaltedEventHandler(new ClnMotionHaltedEventHandler(this));
```

As you navigate the robot simulacrum, you may eventually read a point where it detects a target. When that happens, the Server will send an RsTargetSensorEvent to registered clients. The specification for the target sensor event is a little more complicated than other event handlers.

Because a robot can have multiple sensors, you can register multiple sensor-event handlers. To do so, you have to be able to tell the simulator which sensor is associated with which handler.

The RP1 simulator allows you to name body parts when you create them. In ClientZero, we assigned the name "head" to the forward-looking target sensor on the robot. So when we register a target sensor for the robot, we included it in the specifications.

```
addTargetSensorEventHandler(
   (RsBodyTargetSensor)(getBody().getPartByName("head")),
    ClnTargetSensorEventHandler(this));
```

Two other event handlers are established by ClnMain.initialize(): a position-event handler and a placement-event handler. We'll talk about the placement handler first. The placement handler is used to establish a starting position for the robot when we run a simulation. The event is generated on request, and only by request. Note that the last statement in ClnMain.initialize() is

```
sendPlacementRequest("home");
```

When the Server receives a placement request, it cancels all robot motions and moves it to the named placement. The placement also specifies orientation for the robot. The example floor plan contains only a single placement feature, though any number can be included. Upon placement, an RsPlacementEvent is sent to the client. The placement event contains fields giving the position and orientation for the robot.

The other event handler is a request for position. RP1 allows you to obtain the position and orientation of the robot simulacrum at any time. ClientZero uses this data for navigation purposes. When you send it a mouse-click event, the mouse-click handler issues a position request. When the position event is sent, the client uses it to compute a path to the mouse-click position.

Frankly, using position events for navigation is cheating. Real robots seldom have the means to query for their exact position. ClientZero is an extremely simple-minded implementation and uses position data to keep the amount of code in the demo to a minimum. The position event does, however, have legitimate uses for testing purposes. It can be used as a way of evaluating navigation logic by providing a baseline of true position for comparison with dead-reckoned or computed position. Or it can be used to drive position display elements in a GUI. Many of the RsEvents include position-related data fields for this purpose.

4.6 Physical Layout of ClientZero

The body plan for ClientZero is established using a method defined in the Java class file "ClientZero.java" which is part of the clientzero package. Referring back to the code for ClnMain.java in Figure 6. above, note the statement

```
RsBody body = ClientZero.build();
```

It produces a simple robot body plan that appears as shown in Figure 8. below.

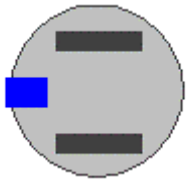

Figure 8 – ClientZero Body Plan

The blue, rectangular element on the front of the robot is a target sensor. The dark rectangles within its body are wheels. The design for ClientZero is deliberately simple and cartoon-like. In particular, the wheels are much thicker than might be found on an actual robot.[9] It is possible to code more realistic depictions.

The code for ClientZero.build() is shown on the page below. Note that build() is a *static* method.

A static method can be invoked from a code without ever instantiating the class that defines them. The build() method returns an object of type RsBody. Objects of class ClientZero need not be instantiated and would be of no value if they were.

[9] Thicker wheels provide better traction and stability, but tend to degrade accuracy when computing position during turns if you are using wheel position for dead-reckoning.

```
package rp1demo.clientzero;
import rp1.rossum.*;   // note: event and request aren't required for this module
import java.awt.Color;

public class ClientZero
{
   public static RsBody  build(){
   RsBody body = new RsBody("ClientZero");

   double trackWidth       = 0.18;
   double driveWheelRadius = 0.075;
   RsDifferentialSteering wheelSystem =
           new RsDifferentialSteering(trackWidth, driveWheelRadius);

   double tireWidth  = 0.03;
   wheelSystem.addDefaultWheels(driveWheelRadius, tireWidth);

   wheelSystem.setFillColor(Color.darkGray);
   wheelSystem.setLineColor(Color.darkGray);


   RsBodyCircle chassis = new RsBodyCircle(0.0, 0.0, 0.15); // 0.3 meters across
   chassis.setName("main chassis");
   chassis.setFillColor(Color.lightGray);
   chassis.setLineColor(Color.darkGray);
   body.addPart(chassis);

   double  d1[] = {
      0.09, -0.025,
      0.16, -0.025,
      0.16,  0.025,
      0.09,  0.025
   };

   RsBodySensor s1 = new RsBodyTargetSensor(
      d1, 4,
      0.16, 0.0, 0.0,
      45*Math.PI/180.0, 1.0,
      3, 3);
   s1.setName("head");
   s1.setFillColor(Color.blue);
   s1.setLineColor(Color.blue);
   body.addPart(s1);

   body.addPart(wheelSystem);
   return body;
  }
}
```

Figure 9 – Source Code for ClientZero

4.7 The RsBody and RsBodyPart Classes

This paragraph uses some of the terminology from the Java language and object-oriented programming in general. If you are unfamiliar with these terms, you may want to review paragraph 1.6, *A Very Quick Introduction to Java*, before proceeding.

Referring to Figure 9, note that the ClientZero.build() method returns an object of type RsBody. RsBody objects act as containers which hold objects of classes derived from RsBodyPart. There are a variety of different kinds of body parts, including sensors, wheel actuators, and shapes corresponding to physical components (such as chassis, housings, etc.). These are added to the body using calls to the RsBody.addPart() method.

Body parts are rendered in the order they are added to the body. So if two parts overlap, the last one added will be drawn on top of the first. In ClientZero.build(), the wheel system is defined early on. Note, though, that it is not *added* until the very end. Thus the wheels are drawn on top of other graphical elements in the robot depiction. In your own robot definitions, you might prefer to hide or partially occlude the wheels by adding them first.

The geometry of the body parts is specified using a Cartesian coordinate system with the origin at a reference point defined as "the center of the robot body." The robot's "forward vector" is defined as the x-axis. When the wheel actuators are added to the robot body, their axle is assumed to be centered on the origin and aligned with the y-axis.

Body parts may be assigned two color values for rendering, the fill color (applies to polygonal forms only) and the line color. The line color may be used to add a contrasting edge to polygonal features. Even if you prefer not to add a contrasting edge, you may want to specify a line color with the same value as the fill color.

By default, both the line and fill color values are defined to be Color.lightGray. You may adjust this value by using the methods:

```
RsBodyPart.setFillColor(Color c);    // Color from java.awt.Color class;
RsBodyPart.setLineColor(Color c);
```

If you prefer not to render a particular feature you may suppress them by using the setFillColor() and setLineColor() methods to assign null color values. Note that the specification of a null color does not affect the interactive characteristics of a body part.

Recall that in Java, the definition of classes is hierarchical. Figure 10 shows that both the RsBody and RsBodyPart classes are derived from the RsComponent class. The arrow notation is used in class diagrams to show inheritance. Remember that a class always "points to the class from which it was derived."

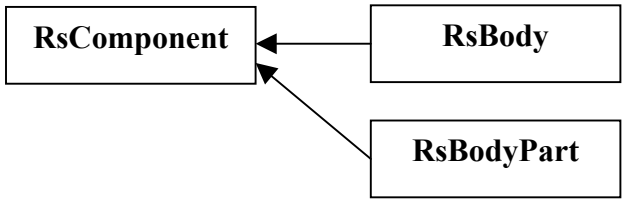

Figure 10 – Inheritence for RsBody and RsBodyPart Classes

The RsComponent class is of interest only from a Java programming point of view. It provides a method for *cloning* (duplicating) objects which is inherited by RsBody, RsBodyPart and all classes derived from RsBodyPart. The clone() methods are used internally by the RP1 simulator. The RsComponent class also implements two Java interfaces, *Cloneable* and *Serializable*. These interfaces are important to Java coding. If they are unfamiliar to you than you need not be concerned with them at this time.

The RsBodyPart class is super class to three major classes as shown in Figure 11 below.

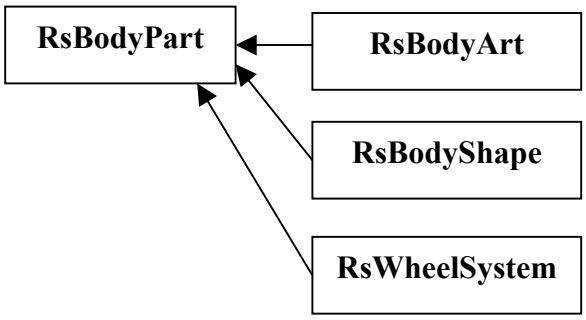

Figure 11 – Subclasses of RsBodyPart

4.7.1 RsBodyArt

RsBodyArt is a partially implemented class. Eventually, it will provide a way of adding decorative elements (lines and polygons) to the robot depiction. These elements will be coupled to the robot body, but will be non-interactive. They might include lettering or markings intended to enhance the depiction of the robot without affect the modeling of its interactions with its environment.

An object of class RsBodyArt will not interact with walls or similar objects.

4.7.2 RsBodyShape

The RsBodyShape class is the super class for all physically interactive body parts, Most of the visible components of the robot are derived from RsBodyShape. These include all passive components, such as the chassis or any housing elements, as well as active components such as sensors. All objects of classes derived from RsBodyShape will interact with walls, resulting in collision events if contact occurs.

The RsBodyShape allows you to create an arbitrary simple polygon for depiction and modeling (the polygon topologically simple… that is to say "not self-intersecting"). Because the RsBodyShape class derives from RsBodyPart, you may specify both a line and a fill color for depiction purposes.

One popular design element for smaller robots such as the Rug Warrior is a circular body plan. The RsBodyCircle class extends RsBodyShape by providing a constructor which allows you to specify the body part using a center and radius rather than a multi-point polygon. RsBodyCircle also overrides the standard paint method defined in RsBodyShape and invokes Java's circle-drawing calls directly in order to create a more pleasing depiction of a circle.

The sensor classes, extend the functionality of RsBodyShape by adding new methods and data elements. These are discussed below.

The constructor for the RsBodyShape class is given below:

```
public RsBodyShape(double []point, int nPoint)
```

Coordinates defining the shape of the body are given in the robot's coordinate system. The point array specifies points as follows:

point[0]	x coordinate of 1st point
point[1]	y coordinate of 1st point
point[2]	x coordinate of 2nd point
point[3]	y coordinate of 3rd point
.	
.	
.	
etc.	

4.7.3 RsWheelSystem and derived classes

The RsWheelSystem and its derived classes are used to model the behavior the wheel actuator for the simulated robot. RsWheelSystem is the base class for a number of existing and planned classes that are intended to model different kinds of robot locomotion systems including those shown in Table 8.

Table 8 – Actual and proposed wheel system classes

Class	Implemented	Description
RsDifferentialSteering	Yes	Simulates the classic differential steering system.
RsAckermanSteering	Yes	Represents an automobile-style steering system.
RsOmniDrive	Planned	Represents an omni-directional system

In the present implementation, two kinds of actuators are available to model robot motion: *Ackerman steering* and *differential steering*. The Ackerman steering implementation represents, at a very abstract level, a steering system to that used in an automobile. Direction of motion is controlled by the adjustment of steering wheels. The differential steering implementation models a wheelchair-like locomotion system in which two independently controlled and powered wheels are mounted perpendicular to a common axis and direction is controlled by varying their speeds. Steering is accomplished by varying the speeds of the wheels. When both wheels turn at the same speed, the robot travels straight. When the right wheel turns faster than the left, the robot steers toward the left, and visa versa. When the wheels turn at different, but fixed, speeds, the robot steers a circular path. If the two wheels turn at the same speed but in opposite directions, the robot can execute a pivot. In real-world implementations, additional wheels, usually free turning casters, are provided for support. In larger, heavier robots, more than one additional wheel may be supplied, but some small robots actually use a skid or even a simple dowel with a rounded end for support. The Rossum Project website at http://rossum.sourceforge.net/papers/DiffSteer/ provides a detailed discussion of the kinematics for a differential steering system.

Because it is an abstract class, the RsWheelSystem is never instantiated directly. Instead, a wheel system is created by instantiating an object of one of the derived classes, such as RsDiffentialSteering.

```
public class RsDifferential extends RsWheelSystem
{
   // constructors

   public RsDifferentialSteering(
      double wheelTrackWidth,
      double driveWheelRadius);
}
```

4.7.3.1 Adding Wheels

By default, the wheels associated with a wheel system are not depicted. If you wish wheels, you have to add them to the wheel system using its addWheel method. As a convenience, each wheel system derived class implements a method called addDefaultWheels. An example of this was included in the sample code in Figure 9 right after the initial declaration of the wheel system.

4.7.3.2 Motion Control

At present, the motion models supported by RP1 are extremely limited. They are based strictly on kinematics and do not attempt to account for dynamics. In fact, the current implementation even ignores the problem of acceleration, assuming that when a simulated robot makes changes in velocity, it does so instantaneously. In future revisions, we hope to introduce specifications for acceleration to the RP1 client API.

Different locomotion systems use different control paramters. Consult the Javadoc based API documentation for more information on the motion control for the RsDifferentialSteering and RsAckermanSteering classes. See also the discussion of Motion Requests in paragraph 5.3.

4.7.3.3 Limits to Realism in Motion Modeling

Even when the RP1 models are improved, the will only represent actual robot behaviors in a very abstract sense. The real world is an interesting place. Accurately modeling the dynamics of a real-world robot requires a great deal of data about the robot's construction and the physical conditions under which it operates, as well as sophisticated analysis of that information. Mechanical devices are noisy systems often difficult to predict and prone to seemingly random error. In the idealized world of the RP1 simulation, things are much simpler. RP1 assumes perfect traction, timing and control of the wheels. Batteries do not wear down. Motors do not overheat. Error factors are not introduced to the model. If desired, developers writing client applications can introduce their own error factors (applying their own special knowledge of the particular behavior of the system they are modeling).

4.7.3.4 Wheel and Path Calculations

Clearly, the path followed by the robot is dependent on the geometry of the wheels (wheel base, tire radius, etc.) and the velocity at which they turn. These specifications are contained in objects of the RsWheelSystem and its derived classes. So it is not surprising that RsWheelSystem also provides methods for navigation and specifying robot control requests.

Two useful methods for performing computations are:

```
public RsMotionRequest getMotionRequest(
    boolean useStepMethod,
    double x, double y, double speed);

public RsMotionRequest getMotionRequestForPivot(
    boolean useStepMethod,
    double x, double y, double speed);
```

These methods general objects of the RsMotionRequest class. Request classes are discussed below. RP1 clients use the motion-request class to instruct the robot simulacrum to rotate its wheels. The getMotionRequest() method returns an RsMotionRequest that will put the robot on a path to the specified (x,y) coordinates. The getMotionRequestForPivot() returns a request which instructs the robot to face the specified coordiantes.

The coordinates (x,y) in these method calls are given relative to the robots current position and orientation. They refer to the "center point" of the robot that is defined when its body is specified. So a coordinate of (x,y) = (1.0, 0.0) would indicate a point 1 meter directly in front of the robots center point. A coordinate of (x,y) = (0.707, 0.707) indicates a point 45 degrees to the front and left of the robot. Note that a coordinate specification of (-1.0, 0) would result in the robot backing up.

The speed value in the getMotionRequest() method describe is the requested speed as measured at the robot's center point. Speed is given in meters per second. It should always be a positive value. In the motion that results from a call to the getMotionRequestForPivot() method, the robot pivots and its center point does not move. In this call, the speed value refers to the speed of a point located at the center of either wheel hub. The wheels for a robot with a 1-meter track width (the distance between wheels) will have to travel 3.14 meters in order for the robot to make a complete 360° pivot. To complete such a motion in 1 second, you would have to specify a pivot speed of 3.14 meters/second.

You may study the code in RsWheelSystem.java for more details. Again, the existing model assumes fixed speeds and does not treat acceleration.[10]

4.7.4 RsBodyPainter

The RsBodyPainter class allows you to create a robot that can leave a "paint trail" on the floor of the simulation environment. This feature allows an application to show the path of the robot as it moves about the simulated environment. It is useful both for analyzing the robot's behavior or producing robot-generated artwork. For example, it might be used in a vacuum-cleaning robot simulation to show the coverage pattern for a particular algorithm.

The RsBodyPainter class extends the RsBodyShape class, but implements additional methods for defining paint-related attributes. The robot body plan may include any number of painters. The constructor follows the general pattern of the RsBodyShape constructor discussed above:

```
public RsBodyPainter(double []bodyPoints, int nPoints)
```

Once you have created an object of type RsBodyPainter, you may set the various painting specifications using the following accessor methods:

```
public void setTrailerPosition(double x, double y)
public void setTrailerWidth(double width)
public void setPaintColor(Color color)
public void setPaintRegionCode(int code)
```

[10] When acceleration models are implemented, getMotionRequest methods with additional arguments for acceleration and initial velocities will be added.

```
public void    setTrailerMinimumSegmentLength(double minLength)
public boolean getPainterActivationStatus()
```

The trailer-position setting gives the position of a point on the robot's body at which the paint trail is generated. Parameters are set in the robot's coordinate system. The trailer width is the width of the paint line that the painter lays down as it travels. By default, the robot draws a zero-width line (the standard thin line in the Java graphics environment), but if you wish you may specify a thicker setting.

Note that the setTrailerWidth accepts a floating-point value giving the line thickness for the trailer in meters. Some users have wondered why the line width is set in meters rather than in pixels as it is in most graphics environments. Even though the real-valued, width-in-meters, specification may seem counterintuitive there are two important reasons why it is preferred. The first is that it will scale in proportion to other objects in the simulation environment if you re-size the simulator GUI. Therefore, having a real-valued width specification is useful for ensuring a consistent visual presentation. The second reason for using metric-valued specifications is that the concept of objects having an actual physical size is fundamental to the design of the RP1 simulation (and any other robot simulator that is even vaguely realistic).

The painter is activated through calls to request methods (see paragraph 5 *Events and Requests* below) once the robot body plan is uploaded to the simulator. Two methods are provides:

```
sendPainterActivationRequest(RsBodyPainter painter)
sendPainterDeactivationRequest(RsBodyPainter painter)
```

Paint settings can me set at any time in the simulation. The activation request enables painting, and the deactivate request deactivates painting. If you wish, you may change painter parameters (color, width, position) at any time by calling the appropriate accessor methods. These changes are uploaded to the simulator when a painter activation request is transmitted. Thus, an application could switch paint colors by sending the initial activation request, setting a new paint color, and then sending an additional painter activation request. The getPainterActivationStatus method is provided for convenience and allows an application to query the body part for its current activation status.

The setTrailerMinimumSegmentLength method sets an important parameter and it should be used with care. It controls the minimum spacing between points used to record the robot's paint path in memory. By default, it is two centimeters. If you make the setting too small, you application could generate too many points and overwhelm the simulation environment. If you make it too large, your paint trail will appear as a number of rather long, low-resolution segments. An optimal selection of segment length would take into account the robot's size and anticipated rate of travel. A home robot that travels at 1 meter per second would have a much smaller trailer segment length than an automobile-scale robot that travels at 100 kilometers per hour. For robots intended for use in inside environments, the default 2 cm setting is probably adequate.

As of release 0.60, the setPaintRegionCode method was not implemented. Eventually, it will allows a client application to set the region values used by the floor paint objects and RsPaintSensor classes (see paragraph 3.6.6 *Floor Paint*). When implemented, a client with a floor-paint sensor will be able to detect paint deposited by other robots.

4.7.5 The Sensor Classes

The RP1 simulator does not attempt to model specific sensors. In the real world, there are a vast number of different kinds of sensory apparatus, with many different operating parameters. An attempt to model even a fraction of the devices available would be futile. Instead, RP1 provides models for highly abstract sensors. Client applications can use the data provided by the abstract sensors as inputs into their own models. This approach puts the ability to simulate the behavior of real sensors into the hands of the client developer (who probably knows more about the specifics of his system than some guy writing a simulator would anyhow).

The abstract sensors include the following:

Target Sensor	a sensor that detects point objects, such as visible light or infrared sources;
Contact Sensor	a “bumper” sensor that detects physical contact with walls
Range Sensor	a sensor that detects the range to walls, the resolution of this sensor can be adjusted so that it behaves as a “proximity sensor.”
Paint Sensor	a sensor that can detect "floor paint" features.

The sensors classes all derive from the RsBodySensor class which in turn derives from the RsBodyShapeClass as shown in Figure 12. below.

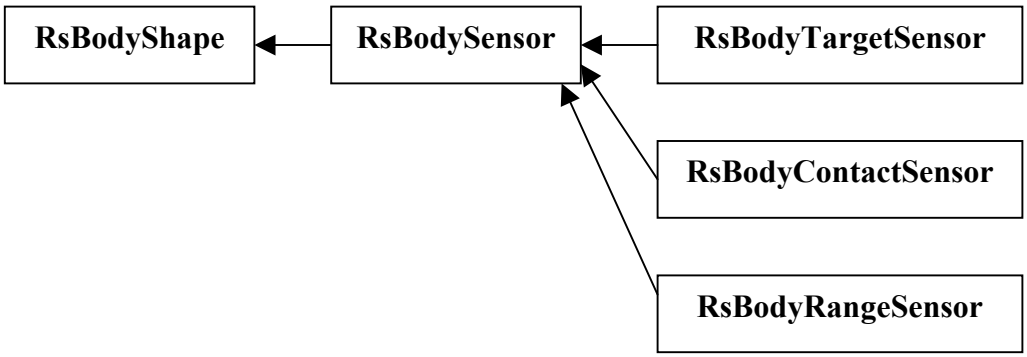

Figure 12 – Derivation of Sensor Classes

RsBodySensor also adds the notion of a "state" to the body part. Sensors are said to be either "hot" or "cold" depending on whether they have a detection or not. If you run either of the ClientZero demo applications, you will note that the sensor on the front of the robot changes from blue to bright orange when it "goes hot" (encounters a detection). The RsBodySensor class adds two color-related methods to the RsBodyPart class:

```
RsBodySensor.setHotFillColor(Color c);
RsBodySensor.setHotLineColor(Color c);
```

If you examine the code for ClientZero.java in Figure 9. above, you will note that these methods were used as part of the sensor definition.

4.7.5.1 Obtaining State Data of a Sensor

Each of the sensor classes provided by RP1 has a number unique parameters describing its state. The particulars of these parameters are described below in the paragraphs dealing with individual sensor classess. RP1 communicates these values to a client application in the form of a "sensor event."

Sensor events are generated whenever a sensor changes state. Of course, there are certain sensors such as a range sensor (which measures distance) which change state pretty much continuously as the robot moves. To prevent the simulator from bogging down with an overwhelming number of events, sensors can be assigned "resolution values" (discrete intervals for which state changes will be reported).

Sensor can also be generated at any time that the client application sends a "sensor request" to the server (a feature which can used to simulate a sensor-polling system).

4.7.6 RsBodyTargetSensor

The RsBodyTargetSensor, which was mentioned above, is used to detect objects of the RsTarget class. RsTarget objects are single-point features that can be added to the simulator's floor plan. They can be used to model visible light or infrared sources.

The constructor for the RsBodyTargetSensor is shown below:

```
public RsBodyTargetSensor(
      double [] point,
      int    nPoint,
      double xDetector, double yDetector,
      double sightAngle,
      double width,
      double maxRange,
      int    nWidthBin,
      int    nRangeBin
   )
```

The specifications for the RsBodyTargetSensor constructor are as follows:

point, nPoint	the physical geometry of the sensor, used for collision and depiction modeling;
xDetector, yDetector	the location of the "focal point" of the sensor; even if the sensor to be modeled is not actually optical in nature, a focal point is used as a conceptual element;
sightAngle	the orientation of the central axis of the target sensor; the sensor can point in any direction that you please; if the sensor is omni-directional, this value will be set to zero;
width	again, an optical analogy: width describes an angle of view in which a point feature can be detected;
maxRange	the maximum range at which a target can be detected;
nWidthBin	the angle of view is treated as divided into a number of discrete bins; each time a target point moves from bin-to-bin, it is considered a "state change" and a sensor event is generated;
nRangeBin	analogous to the width bin, used for range.

As noted above, when a sensor changes state, the simulator generates a sensor event. Client applications can apply their own enhanced models to the data in the sensor event. For example, if you were measuring an IR source with a real-world device, you might expect a drop in sensitivity as the source moved away from the detector axis. A dim source, which could be detected at a range of one meter when the sensor was pointed directly at the target, might not register when the sensor was turned slightly. Your client application could add this level of enhanced modeling based on specific knowledge of the devices you were using.

The fields provided in the RsTargetSensorEvent are described in the section on events below.

4.7.7 RsBodyContactSensor

The RsBodyContactSensor is used to model pressure-sensitive contact sensors. Typically contact sensors are mounted on bumpers on the outermost edges of a robot chassis. The RsBodyContactSensor has two states: detection and no-detection. When a robot simulacrum begins a motion, any contact sensors are set to the no-detection state. If any contact sensors were previously activated, and thus undergo a state change, an RsContactSensorEvent will be issued

with its status field set to false (no-detection).[11] If a collision occurs at the end of a motion, contact-sensor events will be issued for any sensors that are activated.

The constructor for a contact sensor is shown below:

```
public class RsBodyContactSensor extends RsBodySensor {
   public RsBodyContactSensor(
     double []point,
     int nPoint
   )
```

As you can see, the definition of a contact sensor is a simple closed polygon.

4.7.8 RsBodyRangeSensor

Many robots include either range or proximity sensors which allow them to detect objects at a distance. Typically, such devices are based infrared or sonar emitter/detector apparatus though more exotic (and expensive) laser-based systems are also available. Both range and proximity (that is, very-close-range) sensors can be modeled using the RsBodyRangeSensor class. The constructor for this class is shown below:

```
public RsBodyRangeSensor(
   double []point,
   int nPoint,
   double xDetector, double yDetector,
   double sightAngle,
   double maxRange,
   int    nRangeBin
)
```

Data elements include the following:

xDetector, yDetector	position of the detector relative to the robot origin
sightAngle	direction of the central axis for the detector
maxRange	maximum range at which the sensor can detect and object
nRangeBin	resolution of the sensor, divides the maxRange into number of bins.

Since most of the elements of the range-sensor are similar to those of the target sensor described above. they will not be reviewed here.

4.7.9 RsBodyPaintSensor

The paint sensor works in conjunction with the floor-paint feature described in paragraph 3.6.6 above. It provides a way of simulating a variety of real-world sensors and functions. For example, one popular technique in mobile robotics is to equip a robot with a LED and optical sensor device that allows it to follow a track on the floor (such robots are sometimes called "line-

[11] Not all motion requests will result in a contact-sensor status change. For example, if the robot has driven into a wall, a request for a forward motion will just press the robot against the wall harder. No movement will occur, so no sensor state change will result.

followers"). In the RP1 simulation, such a track could be represented using floor paint with the robot using an RsBodyPaintSensor to detect it.

Of course, the RP1 paint sensor doesn't necessarily have to model a real-world equivalent. For example, imaging that you wished to implement areas of "slippery floors" where the robot would lose traction. Although RP1 itself does not provide this capability, it could be implemented by placing a paint sensor near the each wheel and implementing code on the client side to adjust the robot's trajectory accordingly.

The paint sensor is modeled as a single detection point. As is the case with all RP1 sensors, it can be optionally be assigned a polygon to be used for depiction purposes or modeling collisions.

The general declaration for a paint sensor is

```
public RsBodyPaintSensor(
      double []point,
      int      nPoint,
      double  _xDetector,
      double  _yDetector
   )
```

The array *point* and corresponding element count *nPoint*, indicate the geometry of the sensor. The coordinate pair *(xDetector, yDetector)* indicates the position of the sensor.

Additionally, the paint sensor features two methods for specifying the sensor's *sensitivity* to floor paint features with different *region values* as described in 3.6.6. These are:

```
public void setRegionSensitivity(int region);
public void setRegionSensitivity(int []region);
```

5 Events and Requests

Events and requests were introduced in paragraph 2.3.2 *Communication via Events and Requests* above. As mentioned in the preceding sections, once an RP1 client connects to the server and exchanges introductory data, communication between the RP1 server and its client applications is accomplished through the use of what the system calls *events* and *requests*. Events are messages sent to the client when something happens in the simulation environment (the robot collides with a wall, a sensor detects an object, a timeout period expires, etc.). Requests are messages sent to the client by the application when it wants something to happen (the robot is to move forward at some speed, a timeout event is desired, etc.).

Recall that the fundamental entity in Java is the "class" (see paragraph 1.6 *A Very Quick Introduction to Java* above). RP1 implements classes for each kind of event and request that are to be exchanged between the client and server. All RP1 event classes derive from the base class RsEvent. All RP1 request classes derive from the base class RsRequest. In general, the code for client applications makes extensive use of RP1 events. On the other hand, the RP1 client API handles most requests internally and only a few are actually exposed to the client code. All RP1 event classes are defined in the Java package *rp1.rossum.event* and requests classes are defined in the Java package *rp1.rossum.request*.

5.1 Interacting with the Simulator Through Events and Requests

Events and requests are all exchanged between the client and the server through the agencies of the RsClient object that is established when the client first connects with the server. For example, when the client application first wishes to place its robot body in the simulation environment, it sends a placement request as shown in the code fragment below:

```
RsClient myClient;    // assume the connection was previously established

myClient.sendPlacementRequest("$random");
```

In this case, the client requests that the simulator assign its robot to a random position somewhere in the simulated environment. When this happens, the simulator will issue a placement event and send it back to the client. The RsClient object will automatically receive the placement event, but in order for the client to actually do anything with the event, it will have to have registered an object known as an *event handler* with the simulator.

5.1.1 Event Handlers

An RP1 event handler is a class which implements a method called process that can operate on the event when it is received. For example, if there is some action that a robot application must do when it receives a placement event, then it would create an object with the appropriate behavior and register it with the RsClient connection. Examples of such statements appear in Figure 6 and Figure 7 and elsewhere in this document. Once the event handler is registered, RsClient will invoke the process method when the message is received.

The definition of event handlers is actually quite simple. Figure 13 shows the source code for the RsPlacementEventHandler and the RsTimeoutEventHandler. Note that these event handlers are not implemented as Java classes, but rather as Java interfaces. All that means is that any class that is intended to act as an event handler must implement the method declared within the interface definition. In this case, it means that a class that is to act as an RsPlacementEventHandler must implement a method called "process" that takes an RsPlacementEvent as an argument. A similar rule applies for an RsTimeoutEventHandler. An example of an event handler implementation is shown in Figure 14. Note the use of the Java keyword "implements" in the class declaration. In Java, a class can implement as many interfaces as desired. So if the developer wished, the class shown in Figure 14 could perform double duty as, for example, a timeout event handler. It could do so by implementing a second method called process that operated on timeout events (Java permits classes to overload methods, specifying multiple methods with the same name, provided they operate on different arguments). In general, however, most developers find it easier to code one event handler per class.

```
package rp1.rossum.event;

interface RsPlacementEventHandler extends RsEventHandler  {
   public void process(RsPlacementEvent event);
}

interface RsTimeoutEventHandler extends RsEventHandler  {
   public void process(RsTimeoutEvent event);
}
```

Figure 13 – RP1 source code for RsPlacementEventHandler

```
package aTypicalPackage;
import rp1.rossum.event.RsPlacementEventHandler;
import rp1.rossum.event.RsPlacementEvent;

public class ATypicalClass implements RsPlacementEventHandler {
   public void process(RsPlacementEvent placement){
      System.err.println("received placement at "+placement.name());
   }
}
```

Figure 14 – A typical event handler implementation

All RP1 events are defined as classes in the package rp1.rossum.event. Since they all derive from RsEvent, they all carry at least one common element, the simulation time, which is defined in the base class. Some, of the event classes, RsSensorEvent, serve as super classes to special subgroups of classes. The best place to read about the individual event classes is in the API documentation (Javadoc) for RP1.

5.1.2 Requests

There are a number of requests defined in the package rp1.rossum.request. In most cases, however, a RP1 client does not have to actually instantiate requests, but simply invokes them through a call to a method in RsClient. We've already discussed how the client could send a request for robot placement by invoking the sendPlacementRequest implemented by RsClient. Similar methods exist for all requests supported by RP1. Although RsRequestHandler interfaces are defined in the RP1 code, they are strictly the concern of the simulator itself and need not be considered when implementing client applications. Again, the best place to read about the individual request classes is in the API documentation (Javadoc) for RP1.

5.2 Placement Requests and Events

The placement request is used to obtain an initial position and orientation for the robot in the simulated environment. The robot can be positioned only at specific positions called "placements" which are specified in the floor plan (see paragraph 3.6.5 above). When a placement request is issued, the client indicates the desired placement by name.

5.2.1 Random Placements

The RP1 server also provides two methods for obtaining "random" placements. First, if the floor plan includes multiple placements, the client application may specify that it wishes to choose one of them at random. To do, it needs only specify a blank or null placement name. Additionally, the client may specify that it wishes for the robot to be placed at a random position within the floor plan by specifying the special placement name "$random".

5.2.2 Initializing a Placement

The robot simulacrum does not exist in the simulated landscape until a placement request is received and processed. Any request for wheel movement of other robot-specific events will be ignored until a placement has been obtained. If the graphics option is selected, the robot will become visible as soon as the placement request is processed.

```
public class RsPlacementRequest extends RsRequest
{
   // constructors
   public RsPlacementRequest(String _name);
}
```

```
public class RsPlacementEvent extends RsEvent
{
   // constructors
   public RsPlacementEvent(
      long    _simTime,
      boolean _valid,
      String  _name,
      double  _x, double  _y, double  _orientation);

   // elements
   public final String  name;
   public final boolean valid;
   public final double  x, y;          // in meters
   public final double  orientation;   // in radians
}
```

5.2.3 Valid and Invalid Placements

The element "valid" in the RsPlacementEvent indicates whether the robot could successfully be situated at the placements specified in the RsPlacementRequest. Placement requests may fail if:

1. A name is supplied in the request which does not match a placement in the floor plan.
2. The robot would not fit in the specified placement because it would overlap a wall or obstacle.
3. The random placement function could not find an open space in which the robot would fit without overlapping a wall or obstacle.

5.3 Motion

5.3.1 Motion Requests

5.3.1.1 Starting a motion

Once the client has obtained a placement, it becomes possible to move it around the simulated environment. There are two ways to initiate a motion, through an RsMotionRequest or through an RsActuatorControlRequest. The RsClient class allows a client to convey these request to the simulator through calls to its sendMotionRequest and sendActuatorControlRequest methods. The RsMotionRequest request allows the client application to specify an arbitrary motion based on a linear travel velocity and turn rate without considering the actuator control parameters necessary to achieve those behaviors. For example, if you wished the robot to travel a straight path at a fixed velocity, you could specify an RsMotionRequest with a non-zero linearVelocity and a zero rotationalVelocity The constructor for an RsMotionRequest is shown below:

```
public class RsMotionRequest extends RsRequest
{
   // constructors
   public RsMotionRequest(
      double linearVelocity,
      double rotationalVelocity,
      double durationSeconds);
}
```

An actuator control request allows the client application to control the motion behavior by setting actuator controls. Actuator control requests are usually not instantiated directly (though there is nothing to prevent an application from doing so), but rather instantiated by using a creation method in the corresponding actuator class. For example, to control the wheel velocities of a differential steering system, you could use a method from the RsDifferentialSteering class as shown below:

```
RsDifferentialSteering steeringActuator;

RsActuatorControlRequest acr =
   steeringActuator.getControlRequestForWheelRotation(
           leftRotationalVelocity,
           rightRotationalVelocity);
```

Different steering mechanisms have different capabilities. For example, an Ackerman steering system (an automobile-style system) cannot execute a pivot. Thus classes derived from RsWheelSystem (see paragraph 4.7.3) implement specialized methods for creating valid motion requests based on the system the represent. Code examples for these methods are included in ClientZero and other demo applications. Even if the methods provided are not adequate for your needs, it may be useful to review the code before writing your own.

5.3.1.2 Halting a motion

A motion sequence may be terminated early by sending an RsHaltRequest. Typically, an application does not instantiate RsHaltRequest directly, by uses the sendHaltMotionRequest method provided by RsClient:

```
public synchronized sendHaltMotionRequest();
```

5.3.2 Motion Events

Motion events are issued at the beginning of a movement and at its termination. The RsMotionStartedEvent and RsMotionHaltedEvent classes are shown below.

```
public class RsMotionStartedEvent extends RsEvent
{
   public final double linearVelocity;     // meters/sec
   public final double rotationalVelocity; // radians/sec
   public final double duration;  // in seconds

   // initial position data
   public final double x;
   public final double y;
   public final double orientation;
}
```

```
public class RsMotionHaltedEvent extends RsEvent
{
   // data elements
   public final int    causeOfHalt;
   public final double x, y;
   public final double orientation;

   // integer codes for causeOfHalt field
   public static final int    HALTED_ON_COMPLETION=0;
   public static final int    HALTED_ON_REQUEST=1;
   public static final int    HALTED_ON_COLLISION=2;
}
```

Note that the RsMotionStartedEvent includes the parameters from the RsMotionRequest that caused it. It also provides "cheating" elements giving the position, orientation, and velocities of the robot at the start of the motion. These elements are convenient for validating your navigation calculations and driving a user interface. The RsMotionHaltedEvent includes an integer element, causeOfHalt, that tells the reason for the halt:

completion	the motion is completed without incident;
request	the client issues an RsHaltRequest or an RsMotionRequest before it has time to complete the current motion sequence;
collision	the client collides with a wall.

5.4 Timing Events

The RP1 simulator currently supports two events related to timing: heart-beat events and time-out events. The RsHeartbeatEvent is issued periodically at a user-specified interval of time. The RsTimeoutEvent is issued once, after a user-specified interval of time has elapsed.

It is important to note that the RP1 timing events are based on simulation time, not the real-time clock. If the simulation speed is set to 1.0, simulation time will correspond closely to real time, but it will not be an exact match. Issues such as system process scheduling and communications overhead can degrade both the *precision* and the *accuracy* of the timing interval.

5.4.1 RsHeartbeatEvent

The RsHeartbeatEvent is a very simple event that is issued only on request by a client application. An initial request starts the heart beat, resulting in a sequence of events being issued by the server at fixed intervals of simulation time (specified in seconds). The heat beat will continue until cancelled by the client.

One common use of the RsHeartbeatEvent is to drive a clock display in a GUI or to model a sensor-polling implementation. The selection of time interval is arbitrary, but only one heartbeat can be initiated at a time.

The RsClient methods related to heartbeat events are:

```
public synchronized void addHeartbeatEventHandler(RsHeartbeatEventHandler);
public synchronized void removeHeartbeatEventHandler();

public synchronized void sendHeartbeatRequest(double seconds);
public synchronized void sendHeartbeatCancellationRequest();
```

5.4.1.1 A Caution about Heartbeat Backlogs

There is a danger in using the RsHeartbeatEvent. When a client application activates the heartbeat, events are issued whether the client is processing them or not. The events are, of course, intercepted by the RsClient event loop (its "run()" method). If RsClient has transferred control to an event handler which has bogged down, it will stop reading event messages from the Server-communications socket. Events can back up. Initially, event backlogs remain local to the client and only degrade the timing relationship between events and the real-time clock. A sever backlog can propagate to the Server (when the client's socket buffers fill up) and actually interfere with the operations of the simulator itself.

It is unlikely that an active client will result in backlogs of such a degree that they affect the Server. The system is, however, vulnerable to a situation in which a client requests a small-interval heartbeat and then becomes hung up in an infinite loop or other blocking condition.

The heartbeat is a low-priority event. A heart-beat scheduled for time T, will not be issued until all other events scheduled for time T have been sent to the client.

5.4.2 RsTimeoutEvent

The RsTimeoutEvent provides a mechanism for client applications to request that the simulator send it an event after some specified delay. The timeout period is specified in seconds. A delay of zero can be used as a mechanism for requesting the simulation time.

The timeout event is a only little more complicated than the heart beat event. In addition to the simTime field, it also includes the field called the "timeout Index."

```
public RsTimeoutEvent(double simTime, int timeoutIndex);
```

The timeout index is used to allow an application to correlate a timeout event with the request that generated it. Suppose, for example, that you wished your client application to perform a certain action when a timeout expired, but that other parts of the application had also requested timeouts. Since RsClient only supports a single set of timeout event handlers, all registered handlers are invoked when a event comes in. By noting the timeout index returned by the RsClient method sendTimeoutRequest, and comparing that index to the timeoutIndex field of the timeout event when it is received, your application can determine if the event is the one that is intended for a particular module.

The RsClient methods related to timeout event handlers are:

```
public synchronized void addTimeoutEventHandler(RsTimeoutEventHandler);
public synchronized void removeTimeoutEventHandler();
public synchronized int  sendTimeoutRequest(double timeoutPeriod);
```

5.5 Sensor Events

Sensor events are, naturally, coupled to specific sensors. The identity of the sensor which generated the event is often critical to modeling it correctly. RsSensorEvent serves as the parent class for all other sensor classes and introduces the element sensorID which allows you to identify the body part associated with the event.

```
public abstract class RsSensorEvent extends RsEvent
{
    // constructors
    public RsSensorEvent(long _simTime, int _sensorID);

    // elements
    public final int sensorID;
}
```

The sensorID can be used with the method RsBody.getSensorByID() to get the sensor object associated with the event.

When event handlers are added for sensors, a reference to a specific sensor object is always part of the specification. For example, RsClient includes the following declaration for the target sensor:

```
public synchronized void addTargetSensorEventHandler(
      RsBodyTargetSensor sensor, RsTargetSensorEventHandler reference);
```

Suppose that your robot client featured two independent target sensors with much different characteristics (one was a visible light sensor, the other detected IR). You could specify different event handlers for each. Alternatively, if you had two sensors with similar characteristics, you might prefer specify the same handler for both and then decide which sensor generated the event using the sensorID element of the event object.

5.5.1 The RsTargetSensorEvent

The target sensor is used to detect the presence of a target point in the RP1 simulator. It does not correspond to any real-world sensor but provides data that can be used to model the behavior of an actual device.

To understand how the RsTargetSensor works, it is convenient to think of it as a flashlight. The light beam has a width and a range. If it sweeps a target point (an RsTarget object in the simulation floor plan), it experiences a “detection.” A detection event leads to the simulator

issuing an RsTargetSensorEvent. In fact, any change in the sensor's state produces a sensor event. So if it acquires a target, or loses it, an event will be issued.

The position of the detection is relative to the sensor's position and orientation and is given as a range and bearing. In paragraph 4.7.6, which introduced the RsBodyTargetSensor class, we mentioned that the sensor's beam can be divided into a number of angular sections and range bins. When the target point moves from one section or bin to another, the movement results in a state change. As we noted, any sensor state change generates an RsTargetSensorEvent. Therefore, it is important not to specify too fine a resolution for these values. If you do, even the slightest movement of the robot will result in a flood of events, which will bog down the simulation.

The RsTargetSensorEvent includes a couple of fields which would not be available from a real sensor: absolute position (x, y) of the sensor and a unit vector (ux, uy) giving the line-of-sight direction. These elements are for diagnostic or human-interface purposes. Using the diagnostic elements as part of a client application's navigation logic is cheating.

```
public class RsTargetSensorEvent extends RsSensorEvent
{
    public final double   x, y;
    public final double   ux, uy;
    public final boolean  status;
    public final double   range, bearing;
    public final double   xDetection, yDetection;
}
```

Detection range is measured in meters. Detection bearing is measured in radians, counter-clockwise from the central axis. To avoid confusion, note that the bearing behaves differently than the traditional compass bearing (which would be measured clockwise, in degrees, from the central axis).

x, y	The current position of the sensor's "focal point" (see section 4.7.6), mapped according to the simulacrum's current position and orientation. Note that these values are "cheating fields" and are defined only when status is true.
ux, uy	A unit vector giving central axis (line-of-sight) of the sensor, mapping according to the simulacrum's current position and orientation. Note that these values are cheating fields and are defined only when status is true.
status	Sensor status (true when there is a detection).
range, bearing	Range and bearing of target relative to the sensor's central axis.
xDetection, yDetection	bearing*cos(range), bearing*sin(range), respectively

5.5.1.1 Computing Target Position from an RsTargetSensorEvent

The target-sensor event includes "cheating fields" that allow you to compute the absolute position of the target. In a real-world application, you would have to resolve the actual position based on information about the robot's position and knowledge of its physical layout. Using the cheating fields, you may compute the absolute position using the following calculation:

```
given RsTargetSensorEvent e:

      x = e.x + e.xDetection*e.ux - e.yDetection*event.uy;
      y = e.y + e.xDetection*e.uy + e.yDetection*event.ux;
```

For this calculation, we use the adjusted central-axis vector as the basis vectors for a rotated coordinate system with rotated x-axis (e.ux, e.uy) and rotated y-axis (-e.uy, e.ux). We then multiply each vector by the coordinate in the direction x (e.xDetection) and the coordinate in the direction y (e.yDetection). We could also have used range and bearing for these calculations.

5.5.2 The RsContactSensorEvent

The contact sensor event is issued in response to a state change for a contact sensor. Contact events occur only at the beginning of a motion (if the pressure on a sensor is released) and at the end (if the motion results in a collision). The RsContactSensorEvent class adds only a single field, "status," to those defined by its parent class RsSensorEvent.

```
public class RsContactSensorEvent extends RsSensorEvent{
   public final boolean status;     // true when contact is made
}
```

5.5.3 The RsRangeSensorEvent

The RsRangeSensorEvent is issued when a range sensor undergoes a state change. Of course, when a robot is in motion, its range relative to nearby objects changes continuously. As in the case of the target sensor, the range sensor implements a resolution scheme to prevent an excessive number of state changes. The resolution is specified using the nRangeBin parameter. The fields included in an RsRangeSensorEvent are shown below:

```
public class RsRangeSensorEvent extends RsSensorEvent{
   public final double  x, y;
   public final double  ux, uy;
   public final boolean status;
   public final double  range;   // valid only if there's a detection
}
```

The coordinate pairs (x,y) and (ux,uy), give the absolute position of the sensor and a unit vector indicating its orientation. The absolution position of the detection point can be computed using the formulae

ax = x + range*ux;
ay = y + range*uy;

where (ax,ay) are the desired coordinates. These values are not usually available from real-world sensors and are provided only for diagnostic and user-interface purposes.

5.5.4 The RsPaintSensorEvent

The RsPaintSensorEvent is issued when a paint sensor experiences a state-change. Recall that the sensitivity of paint sensors can optionally be restricted to paint features with specific region codes. The fields included in a paint-sensor event are shown in the code fragment below. Note that the x and y coordinates shown in the fragment specify the position of the detection. As usual, we note that these values are not usually available from real-world sensors and are provided for diagnostic and user-interface purposes.

```
public RsPaintSensorEvent(
      long    simTime,
      int     sensorID,
      double  x,
      double  y,
      boolean status,
      int     region)
```

5.6 Adding Realism by Filtering and Intercepting Events

The RP1 simulator's ability to model real-world components and behaviors is limited. As a developer, you know far more about how your robot performs than the simulator does. One useful technique for adding realism to the system is to implement an event handler that intercepts an event, modifies it according to your needs, and then either consumes the event (to prevent it from reaching other event handlers) or invokes alternate event handlers. As long as your filter event handler is the first one registered, it will be the first one called.

To improve on the simulation capabilities available to your client, you may add noise, randomness, or more realistic behaviors to your application. One technique is to insert a block of adapter code between your client logic and the RsClient-based interface. Such an adapter block can also be used to make the Rossum API look more like the actual hardware on which your robot logic is to run.

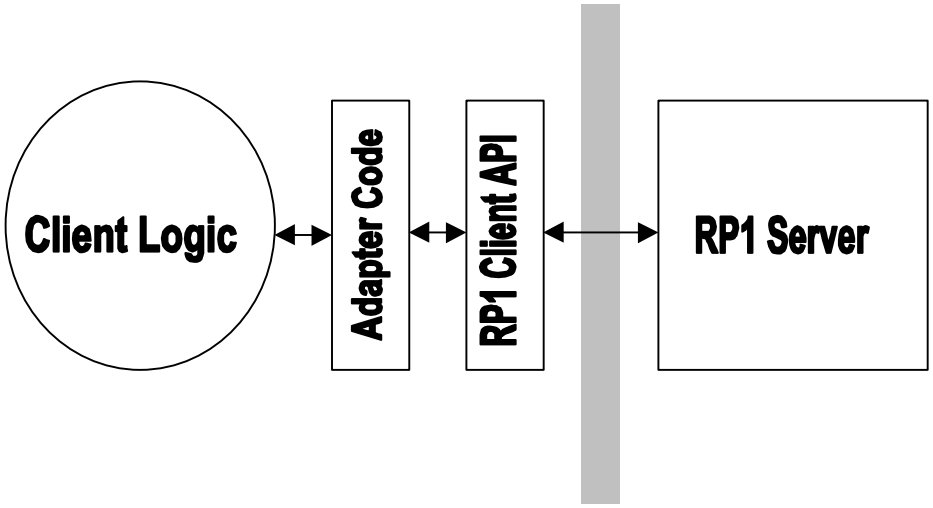

Figure 15 – Adapting an RP1 Interface to Add Realism, Randomness, and Noise

Note that you cannot alter the fields in an event (they are declared as "public final"). To change event parameters, simply create a new event by invoking the appropriate constructor. If you have registered multiple event handlers, be sure to call the consume method for the current event to prevent it from being processed by any remaining handlers.

Appendix A. Migrating Client Applications from Revision 0.50 to 0.60

The changes to the RP1 API that were introduced for Release 0.60 will render some code written to earlier versions incompatible and, in fact, unable to compile. These notes provide hints for migrating pre-existing code to the new API.

To compile these notes, we took the 0.50 version of the ClientZero demonstration program and attempted to integrate it to the new RP1. Thus, they reflect our experiences in migrating the software. The total time required to make the migration was approximately 1/2 hour. Of course, the migration went quickly for us because we knew the stuff pretty well and already had a pretty good idea of what needed to be done for the migration before we started. Still, we hope that the migration task should not be unduly difficult for those who need to perform it.

In all cases consult either the UsersGuide or Javadoc based API documentation for further details.

Finally, we should note that we do not expect to have to perform such extensive API changes to the RP1 in the future. When changes do become necessary, we hope to be able to preserve backward compatibility.

1. Check your CLASSPATH settings

An incorrect CLASSPATH setting is, by far, the most common problem users have in migrating RP1. If the CLASSPATH is not correct, code will not compile. Refer to the RP1 Users Guide or http://rossum.sourceforge.net/sim/install.html document for further details.

2. Change the import statements

Recall that the package organization of RP1 changes for Revision 0.60. To import RP1 client side classes, you will need the following import statements in your code

```
import rp1.rossum.*;
import rp1.rossum.event.*;
import rp1.rossum.request.*;
```

In many cases, it make be possible to omit the import statement related to requests. Very careful programmers might prefer to change the import statements to include only those classes that they absolutely need as in:

```
import rp1.rossum.event.RsTimeoutEvent;
import rp1.rossum.event.RsTimeoutEventHandler;
```

3. Replace RsWheelSystem constructors

The meaning of the RsWheelSystem class changes in 0.60. You should not attempt to instantiate it directly (it is now an abstract class). Instead, you should replace it with one of its two currently available derived classes, either RsDifferentialSteering or RsAckermanSteering. The RsDifferentialSteering class is the direct replacement for the old RsWheelSystem class, though its constructors take slightly different arguments than the original.

Note that the differential steering class with not include wheels for depiction unless you add them using either the addWheel or addDefaultWheels method. We recommend that in your initial implementation, you use the addDefaultWheels call.

4. Event handlers change

The RsTransaction class no longer exists. It was a mistake, and we're glad it's gone. If you developed code following the examples provided in the earlier release, than all your event handlers will feature a method called

```
processTransaction(RsTransaction ...)
```

This method should be removed. In the example code, processTransaction was usually just a "pass-through" to a call to a method called process which operated on the specific event of interest. So, fortunately, it should be enough just to remove processTransaction. Otherwise you will have to add an event-specific process method such as

```
void process(RsTimeoutEvent timeout);
```

5. The getMotionRequest method changes

The first argument of the old getMotionRequest, a boolean flag, is no longer used. Remove it. In the new RsAckermanSteering class, the getMotionRequest will not always be able to support activities for certain parameters (for example, an automobile type locomotion mechanism cannot execute a pivot operation).

6. The constructors for RsMotionRequest change

The design of the old motion request was deeply flawed. It was coupled to a differential steering system, but a motion request is really intended to be a platform-independent request for motion. So the paramters have changed and new constructor is incompatible with the old. If the way the new RsMotionRequest operates is incompatible with your application objectives, you may find it useful to look at the new RsActuatorControlRequest.

7. All Time Values Are Now Expressed In Seconds

This is a significant change. In debugging, it was the one with which we had the most trouble. In the old system, times were expressed in integer milliseconds. In the new, they are expressed as floating-point seconds. Here are a few hints for finding instances of time specifications

Methods with the substring "Millis" in their name no longer exist. The compiler will help you find places where these are used. The string "Millis" does not appear in any element or method of any public class in the RP1 package (though there are still Java methods such as System.currentTimeMillis that do use that convention).

Look for elements named "duration", "simTime", and time. These will all need to change.

Look for the constant 1000 in computations. If you were converting seconds to milliseconds, you no longer need to do so.

Double check all references to sendTimeoutRequest and sendHeartbeatRequest, both of which took time values are arguments.