RobotWorld: a Simulation Environment for Introductory Computer Science Education

by

Craig Allen Henderson

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in
Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1999

Author .....
Department of Electrical Engineering and Computer Science
June 29, 1999

Certified by .....
Lynn Andrea Stein
Associate Professor of Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

RobotWorld: a Simulation Environment for Introductory Computer Science Education

by

Craig Allen Henderson

Submitted to the Department of Electrical Engineering and Computer Science
on June 29, 1999, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Conventional approaches to computer science instruction focus too narrowly on the computation to be performed, and fail to illuminate the greater scheme in which the computation plays a role: as one member in an ensemble of interacting entities.

In order to facilitate the design and construction of these ensemble systems at multiple scales, a virtual world inhabited by programmable agents has been developed. Using such a framework, beginning-level students can: wire up an agent brain to discover its overall behavior patterns, program a community of agents to achieve global behaviors, design interactive graphical displays, and experiment with the underlying distributed architecture of the system.

Armed with this knowledge of interactive programming early in their careers, students will have fewer illusions to dispel, more involvement in the course material, and a clearer understanding of software design as it is practiced today.

Thesis Supervisor: Lynn Andrea Stein
Title: Associate Professor of Computer Science

Acknowledgments

First and foremost, I would like to thank Lynn Andrea Stein, for providing the opportunity to put a microworld to practical use, as well as for being an inspiration and a role model. After exposure to your efforts and Rethinking CS101, I can see science education as an enjoyable and fulfilling endeavor. Thanks to the CS101 staff, especially to Robert Duvall, Mike Wessler, and Mike Harder for feedback on RobotWorld's design and usability. I am grateful to Ryan Olson, for introducing me to the beauty of Java and the dream of massively interactive systems. The dream is now more alive than ever. I should also thank Erik Bailey for many fruitful discussions about science and engineering education as well as for deepening my understanding of control theory. Thanks also to Corey Geving, biologist extraordinaire, for keeping my mind open and in search of the strange and wondrous in nature, and by extension, in ourselves. Thanks to my mother and father, whose encouragement and support over the years have been the firmament to this hurling planet... A son could not ask for better parents. And finally, Reem, I wish to thank you especially for your warmth, laughter, curiosity, generosity, perspective, strength, sanity, and deep and profound interest in human nature. As your pupil and friend, I am indebted to you.

This thesis describes research done at the MIT Artificial Intelligence Laboratory, where the author was supported by grants from the National Science Foundation, under the Young Investigator Award No. IRI-9357761 granted to Prof. Lynn Andrea Stein. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation. In addition, the Rethinking CS101 Project has received support from MIT's Class of 1951 Fund for Excellence in Education and Class of 1955 Fund for Excellence in Teaching, the EECS Department at MIT, the Office of Naval Research through the Science Scholars Program at the Mary Ingraham Bunting Institute of Radcliffe College, Microsoft Research/University Curriculum Programs, and Sun Microsystems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 What is RobotWorld and What Can It Do?

RobotWorld is a Java application which serves as an extensible integrated development environment and networked simulation tool. Visual metaphors guide its design wherever possible. It can also serve to supplement the constructive mode of learning with analysis, software design, and communication tools. This virtual world constitutes several layers of observation for the student: Architecture, Habitat, Morphology, Behavior, and Code. The Architecture editor attempts to depict a "topsight" [31] of the activity on the network, which the user observes whenever RobotWorld is stopped or started, to reinforce the spatial metaphors in the user's mind. The Habitat editor depicts a bird's-eye view of the landscape, littered with toys, food, and obstacles among which the agents cavort. The Morphology editor depicts the geometry of an individual agent's sensors and actuators, as well as its other physical characteristics. The Behavior editor depicts the data flow through the network of neuron-like elements that connect an agent's sensors to its actuators. An individual behavior can be a composite of several behaviors, in which case it can be decomposed into more behaviors in the Behavior editor. Otherwise, the final layer of observation, the Code editor, will be used to display the source code responsible for generating the Behavior. Figure 1-1 illustrates how these layers fit together.

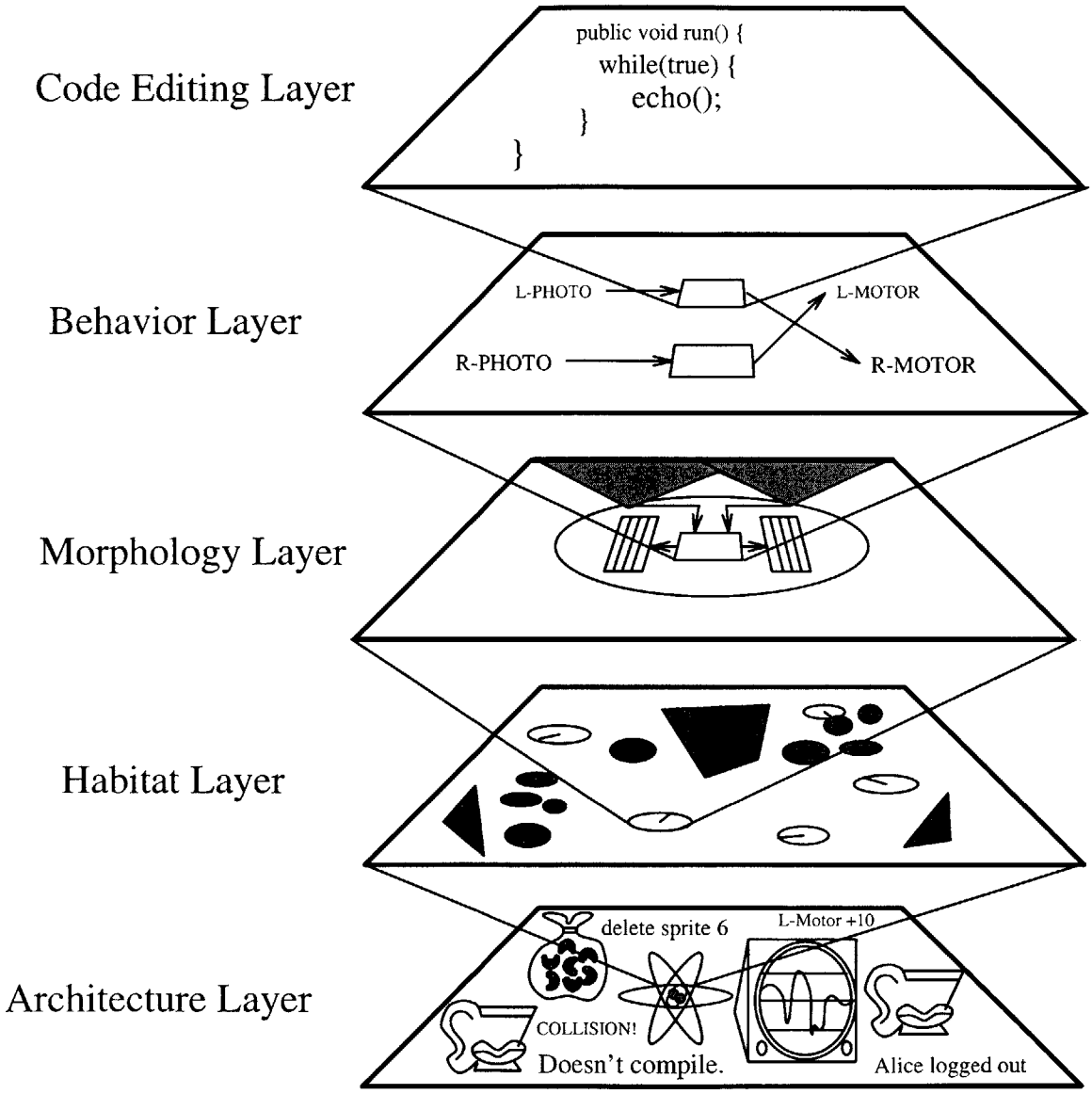


Figure 1-1: An exploded view of RobotWorld, showing the five layers of observation.

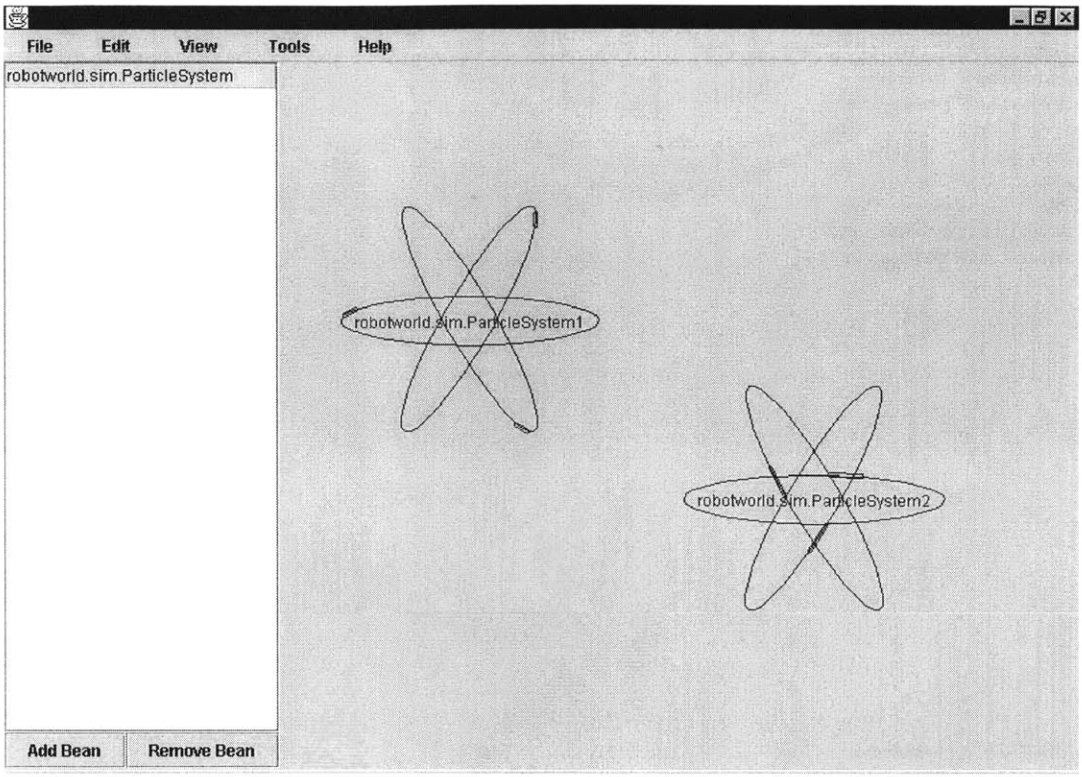


Figure 1-2: A screen shot of the Architecture Editor.

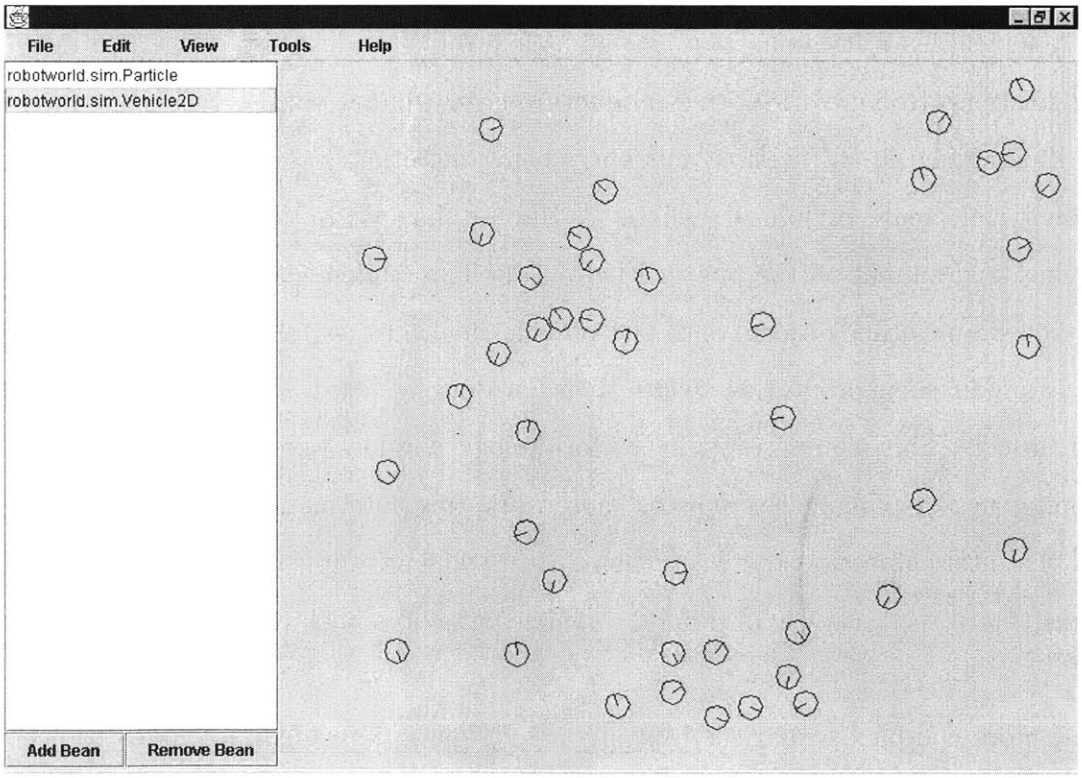


Figure 1-3: A screen shot of the Habitat Editor.

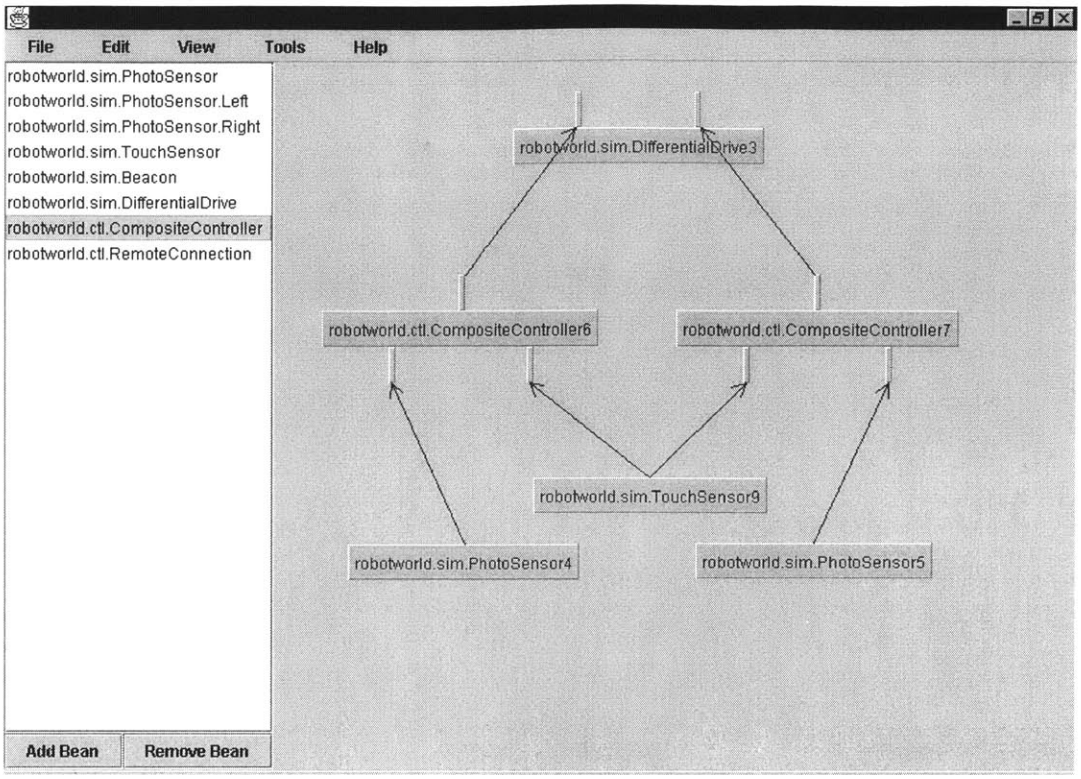


Figure 1-4: A screen shot of the Morphology Editor.

A typical learning scenario might go as follows: a student sits down at a machine and starts up RobotWorld. An Architecture layer window similar to Figure 1-2 pops up, illustrating all the sessions currently being maintained on the class server. The student can create or join a session. A Habitat Layer window pops up, showing a playing field similar to the one in Figure 1-3. The student can instantiate vehicles, obstacles, and other objects onto the playing field, and watch them interact. They can also edit their properties, delete them, or double-click to "zoom in" on a particular object. This would result in a Morphology window like the one in Figure 1-4 popping up, displaying the sensors, actuators, any intermediary processing nodes, and the interconnections between them. By re-wiring simple pre-made components, students will see a variety of different vehicle behaviors back in the Habitat window.

A more complex scenario might involve designing intermediary nodes to accomplish specific goals. In this case, the student will instantiate a behavior node, and

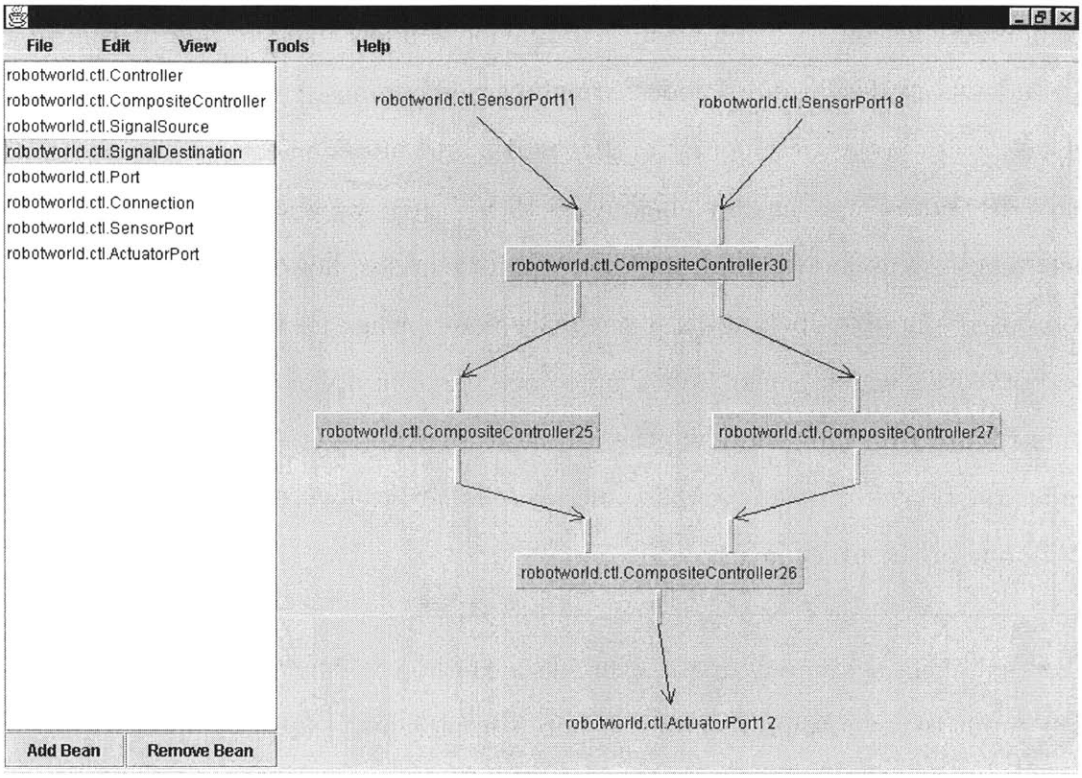


Figure 1-5: A screen shot of the Behavior Editor.

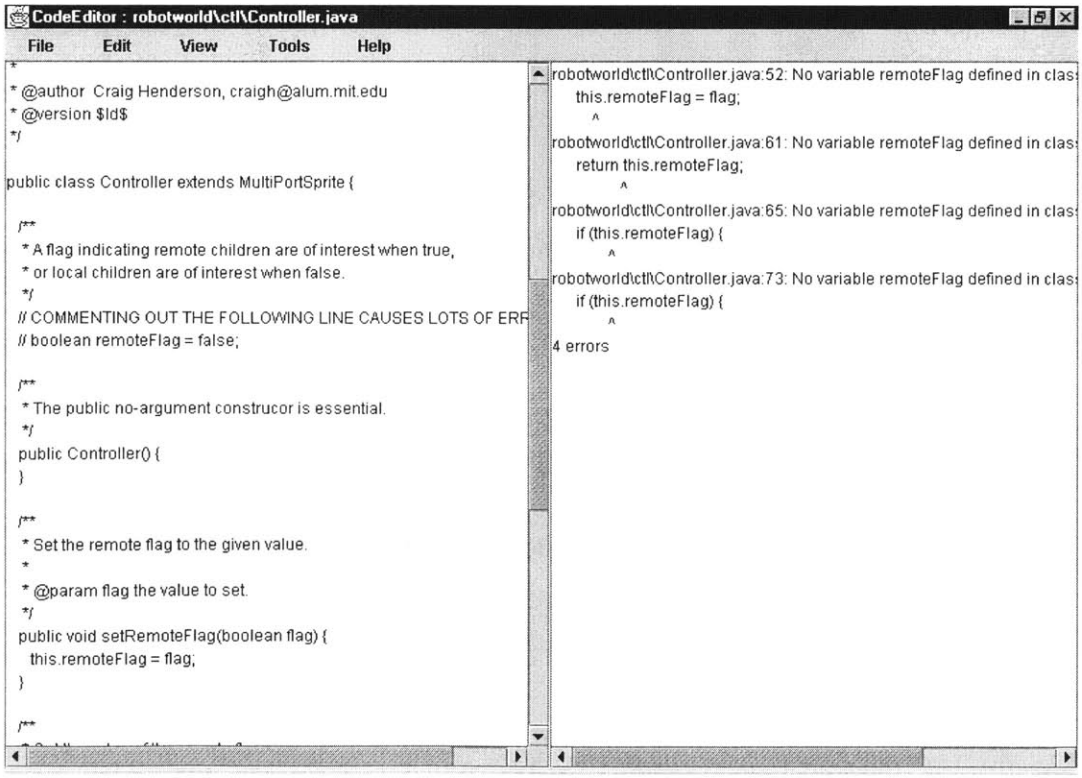


Figure 1-6: A screen shot of the Code Editor.

then double click on it, bringing up a Behavior window like the one in Figure 1-5. The student can design more nodes, which can be composite or primitive in nature. The composite nodes may contain other nodes and interconnections, facilitating recursive decomposition. Double clicking on them brings up another Behavior window which exposes the node's children. The primitive nodes, however, cannot be further decomposed. Double clicking on a primitive node brings up the final level of detail in RobotWorld, which is the Java source code itself, in a Code Editing Window like that in Figure 1-6. This window permits the student to interactively alter the source code for the behavior node, compile, and test the results by creating new components back in one of the other layers.[1]

For instance, a student might override a method which normally just echoes a node's input to its outputs to apply a more complex mathematical operation, or use a control statement to decide whether to output a signal which will back up the vehicle whenever a wall is detected to be nearby. Thus some of the early concepts in computing can be gradually unveiled to achieve more complex behavior first at the component (vehicle) level, and then at a system level, where a collection of objects are to achieve some sort of communal goal. Other possibilities include extending the capabilities of the RobotWorld system by adding interactive displays (e.g. graphing time series and phase plots) or by tinkering with the network protocols that make RobotWorld work.

Another possibility, aimed at more advanced students or the instructors themselves, is to replace or extend some of the functionality provided by the RobotWorld modules. RobotWorld should be able to evolve continuously to meet the needs of students on the one hand, who want a more interactive, hands-on approach to learning the course material, and the needs of instructors on the other hand, who want

[1]It would be much easier if all instances could be auto-magically updated whenever the class changes. However, this would require writing a special virtual machine. The SuperCede integrated development environment offers this feature for true interactive editing, so it is possible. See Future Work, Chapter 11.

to make their material more concrete and accessible so that their students can retain and explain what they have learned.

1.2 Motivation: Rethinking CS101

A beginning-level class in computer science typically begins with the performance of a single computation as the programmer's first goal. Later challenges might include variations on this theme, such as a program that sorts a list of strings alphabetically, or searches for the shortest distance between two nodes in a graph. The illusion of a single thread of control, with compute-and-terminate as the preferred program structure, is maintained until late in the undergraduate curriculum. However, these approaches are still rooted in the dominant paradigm from the nascence of the field: that the primary function of a computer is to perform a single computation, in isolation, and terminate [78]. This may have had more relevance in the era of batch programming, before timesharing and networking. As they are used today, another model is required. Computers need to perform many computations, interdependently with other computational processes, continuously. In other words, they must interact: with people, with other computers, and increasingly, with the physical world in the case of industrial robots, consumer electronics, and other embedded devices.

Lynn Andrea Stein [77] recommends a reworking of the introductory curricula to embrace this new outlook by encouraging students to write interactive software such as robot controllers, graphical user interfaces, web servers and video games. This is not an easy task: in many cases, the students will only have time to complete one piece of any of the above systems. However, encapsulation and procedural abstraction form the basis of modern practice in software engineering. Given an interface to an existing subsystem, a student can write software that interacts with that subsystem to generate the desired behavior.

One such curriculum is underway as a part of the Rethinking CS101 Project here

at the MIT Artificial Intelligence Laboratory. Introduction to Interactive Programming (6.030) is a laboratory-oriented course currently taught at MIT with the goal of introducing students with no prior programming experience to modern computer science and interactive program design using Java. Rather than focus on semantics, the course uses Java as a substrate in which students can learn how to design and construct communities of interacting entities.

Instrumental to the course material are a series of observational laboratories culminating in a final project. However, without a laboratory of sufficient complexity and scope, final projects tend to converge: with few exceptions, everyone just extends the capabilities of one of the later labs. RobotWorld can be used throughout the course as a recurring theme that is presented simply at first, and is progressively uncovered as the material deepens. Chapter 10 gives some indications of how RobotWorld might be integrated into an introductory course on computer science.

RobotWorld forms an interactive, collaborative space in which students can wire up an agent brain to discover its overall behavior patterns, program a community of agents to achieve global behaviors, design interactive displays to integrate into the existing GUI framework, and experiment with the underlying distributed architecture of the system. Students should be able not only to peek under the hood of every major component of the system, but to rip it out and supply one of their own. My hope is that this project will provide a framework which creative students as well as other educators may build upon, due to the modular nature of the software.

1.3 How to Read This Thesis

This thesis arranged into several distinct parts. Chapter 2 is an academic discussion of the motivating principles and inspirations behind RobotWorld. Chapter 3 provides a quick architectural overview of the RobotWorld system. The technical details of the application framework are discussed in Chapters 4–8, while the presentation

layers of the application are covered in Chapter 9. Some examples of RobotWorld in a classroom setting are sketched in Chapter 10, and the numerous features left as future work are recorded in Chapter 11. Chapter 12 concludes the discussion begun in Chapter 2. Appendix A provides a key to the Unified Modeling Language (UML) notation used in several diagrams in this thesis. Appendix B captures the essence of the User's Manual supplied in browsable HTML format with the RobotWorld distribution. And finally, Appendix C is a thorough listing of the RobotWorld source code, broken down by package. This includes the necessary configuration and startup files.

You may wish to visit different chapters of this thesis, depending on your needs. If you would like to use RobotWorld in a classroom setting, read Chapter 3 for an overview and then skip to Chapter 10. Appendix B will also be valuable.

If you would like to extend RobotWorld's capabilities, you should read the quick architectural overview in Chapter 3, skim Chapters 4–8, and read all of Chapter 9 and Appendix B. You should examine the wide array of possible extensions covered in Chapter 10 and especially Chapter 11. (What you want might already be listed as future work, in which case we should avoid duplication of effort.) You may need to backtrack to cover some of the skimmed chapters that are particularly relevant to the problem at hand.

If you would like to understand the motivations, inspirations, and concepts behind RobotWorld, but not necessarily use it, read Chapter 2. You should also touch Chapter 3 and the Background sections of the technical chapters (4–8) to find out how these ideas translated into the application design. RobotWorld is painted as a part of the computer science curriculum in Chapter 10. Finally, Chapter 12 brings the discussion to a close.

Chapter 2

Microworld as Learning Tool

2.1 How Are They Useful?

Microworlds seems to be all the rage these days. From the integrated learning environments reported by the Computer Science Education special interest group (SIGCSE) [37, 47, 21, 58, 68] to robotics and programming toolkits such as the Mindstorms LEGO set, StarLogo [65], and Karel the Robot [60], they are quickly becoming a part of the arsenal of learning tools for enthusiastic educators and self-motivated students as well. Why should anyone take these kits seriously? What do they offer the serious instructor or student of computer science? Section 2.2 describes the guidelines behind the construction of RobotWorld, while Section 2.3 suggests some areas of computer science I hope RobotWorld will let you explore.

2.2 Motivating Principles

The central goal of RobotWorld is to render the core principles of computer science more concrete. Although a robot simulator cannot achieve this monumental task alone, it can be used as a supplement to a larger program of activity. Many other learning tools and styles also aim to achieve this concretizing effect, and RobotWorld draws from the rich history and ongoing progress of these efforts.

2.2.1 Constructivism: Building as Learning

Like many other educational construction kits, RobotWorld encourages students to learn by engaging in free-form building activities. This is motivated in part by the constructivist theory of learning advanced by Jean Piaget [61]. Briefly, learning is believed to be most effective in an active and creative mode, where the learner builds internal knowledge structures in terms of previously learnt structures. This should be contrasted with instructionism, in which the learner is considered a blank slate to be filled with information by repetition and procedure.

Ben-Ari [9] describes the implications of a constructivist approach to computer science education. Each student comes to the field with a different model of a computer with varying levels of viability: some with a picture of a giant calculator, some with a notion of a homunculus or "little person" to perform their tasks, and some with no model at all. The course instructor, then, needs to elicit these models from the students, whether verbally or by written assignment, and help the student to refine or replace them with more viable alternatives for professional and academic studies. If the instructor does not provide a model before delving into programming assignments, students will provide one of their own. This is because programming provides an "accessible ontological reality," meaning that feedback on correctness is both immediate and enforced. Discord between the student's internal models and the reality of the computer—misconceptions—manifest as bugs, which can frustrate or even derail the learning process. The danger lies in accepting the ontology of the computer at a surface level without forming a deeper understanding. To counter this, Ben-Ari suggests that "programming exercises should be delayed until class discussion has enabled the construction of a good model of a computer," and that "group assignments and closed labs should be preferable to individual homework exercises, because they soften the brutality of the interaction with the computer and facilitate the social interaction that is apparently necessary for successful construction." As mentioned before, RobotWorld is meant to be used as part of a larger program of

activity; whatever pedagogical approach is adopted, it will certainly have to address these issues if the students are to form viable models for further work in computer science.

Seymour Papert [59] espouses constructionism, a refinement of this approach which has grown extremely popular. Here, the learner invests personal meaning into an external artifact which serves as representation of their internal resources and as an instrument for further inquiry. Some recent attempts to facilitate this process include: the 6.270 Autonomous LEGO Robot Design Competition [52], Mitchel Resnick's massively parallel StarLogo [65], Programmable LEGO Bricks [70] and Crickets [64], as well as Michael Travers' agent-based programming environment [84], which arose from his much-needed study on the use of metaphor in computer science. Then, of course, there is the venerable LOGO [59], ancestor to all of these efforts.

My interest in computers was shaped to some degree by contact with LOGO as a child, and again by LEGO robots as an undergraduate. As the RobotWorld design progressed, it drew more and more from this prehistory. The link between the constructionist papers which I found myself reading and the resonant learning experiences of the past came something like a revelation. The purpose of RobotWorld, then, is to produce a similar resonance in students of introductory computer science, such that they can form viable models of understanding at the university level.

2.2.2 Transparency Through Visual Design

Multi-layered visual GUIs can provide an alternate learning modality with which to reinforce metaphors. For instance, the regular anatomical structure of Java classes can be stressed by the use of visualization [21]. Alternately, the community of runtime instances can be manipulated by a user in a visual environment [68]. In Mirrorworlds [31], David Gelernter describes how these microworlds, visual or otherwise, can shed light on the inner workings of a complex process. This might be anything from the state legislature to the nitrogen cycle to a virtual machine. The key ingredients to

his vision of a truly powerful microworld are a live, multi-layered picture, complete with agents, a history of past events, and experience. This kind of world can be navigated by successively zooming in on lower layers for more detail, or zooming out for a bigger picture. Between instant messaging systems and agents to carry out user tasks, the microworld becomes an interactive, persistent, concrete model of the target domain. It is not only a learning tool, but an informational tool and a new mode of understanding the target domain.

The goal of a world like this is transparency, but we must ask ourselves, transparency for whom? As Sherry Turkle points out [85], the term once meant transparency for the designer, who renders a complex process like a virtual machine as it actually happens. Lately the term has come to mean transparency for the user, for whom the details have been abstracted away, leaving a simple and intuitive interface like that of the Macintosh. In a sense, this is a core problem of curricular design: what do my students really need to know? A transparent design should allow each instructor to strike a dynamic balance between the two extremes. A multi-layered design should additionally leave each layer open to further inspection [84]. Through its transparent, multi-layered approach, RobotWorld serves this purpose by permitting instructors to conceal or reveal complexity as they see fit.

2.2.3 Experimental vs. Structured Programming

Under the structure of a laboratory setting, students could be encouraged to think carefully about an approach to a programming problem before diving in, and reflect afterwards on the pros and cons of their design. It is all too tempting to jump into a programming assignment and "fly by the seat of your pants." By taking the time to think on paper, students must form a clear picture of the problem, then design a solution. In a closed lab, they could implement and test their designs. Afterwards, they could evaluate its performance relative to other possible designs, and submit a report. This approach will prepare students for a career in business and academia, and is certainly a technique worth investing in students of computer science from an early stage.

On the other hand, there is value in free-form experimentation and observation. Bricolage [86] is an alternate approach, grounded in personal experience and empirical observation. Our discipline stands on a foundation of top-down design and abstraction, yet some students prefer to learn from the bottom up, designing small pieces of functionality which they understand and use as building blocks for larger designs. They abhor treating systems as black boxes—preferring instead to work with glass boxes to see the design as a whole. Are these students wrong? Do they just need time to come around to the abstract way of thinking? Turkle and Papert suggest that students have many voices and styles of learning, and that dogmatically stressing one mode of learning (the abstract, top-down approach) causes discouragement and alienation, not just in our field but in the computer culture at large.

Both modes of learning are supported by the RobotWorld environment.

2.2.4 The Beginner's Development Environment

As much as advanced programmers require sophistication in their development environments, beginning students require simplicity. All too often, however, commercial IDEs become subject to feature bloat and can swamp the beginning programmer in a sea of wizards, menus, and company-specific jargon. Can't we find an intuitive, minimalist interface which makes software development sensible to a nonprogrammer?[1] Perhaps its just better to force students to face the absolute ontology of the command line. But, there is an alternative. It's called "roll your own IDE" and RobotWorld lets you do this.

RobotWorld has a rudimentary IDE for code editing. You can also write your own IDE in Java (code editor and/or compiler), and replace our module with your own. BlueJ [68], which allows you to manipulate the fields and methods of runtime objects,

[1] I grew up on Emacs, but I would hardly call it "intuitive" and "minimalist."

might be a great companion to the framework provided here. The third option is to hook into your favorite external IDE such as Visual J++ or Emacs. You can do all of these simultaneously by the use of the RobotWorld abstract factory mechanism. This is described somewhat in Chapter 5 and more fully in Chapter 9.

2.3 Inspirations

Inspired by several fields of computer science, I tried to design RobotWorld to encourage exploration of the concepts and techniques of each. These include such diverse fields as computer graphics, dynamical systems, artificial life, situated robotics, object-oriented and concurrent programming, and the software development process. One might even be tempted to call these "curricular use cases," meaning the material RobotWorld was meant to help teach. These are not the only fields that could benefit from RobotWorld in particular, or microworlds in general. They are simply the domains I am most familiar with, and have spent time thinking about. If I've been successful, you'll find other ways this framework can be used.

2.3.1 Dynamical Systems

Dynamics is the lingua franca of simulation. Dynamical models are used in computer graphics for rigid body animation, in robotic locomotion, manipulation and some vision tasks. Dynamics may also serve as a compelling approach for the design and analysis of autonomous agents, whether they be humans, nonhuman animals, simulated entities, or physical robots. Typically autonomous agents are modeled from a computational or connectionist perspective. However, these approaches are quite narrow in scope, and are most successful when biological reality is abstracted away. Since both computers and neural networks are instances of dynamical systems, we can see that there is a larger space of possibilities to explore.

So what might a dynamical design look like, and why would we want to teach it? We know what a computational system looks like. An engineer might design

a thermostat using a microprocessor to explicitly measure the current and desired temperatures, compute the difference digitally, and decide on a course of action. This is homuncular design: we postulate a "little person" inside the device who needs all this information to make a decision. Anyone who has played with a mercury-switch thermostat knows that an alternative is possible—a dynamical alternative. In the language of dynamics, the desired temperature is a attractive fixed point, and the system eventually settles into that state.

Van Gelder [87] describes another design, the Watt governor. This 18th century nonlinear dynamical device was designed for the regulation of a steam-powered flywheel, and was only fully analyzed a century later. On the same shaft as the flywheel was a pair of hinged arms with weighted metal balls. When the flywheel velocity increased, the balls were flung outwards and upwards due to centripetal acceleration. This was yoked to a mechanism which closed the valve, cutting off the steam that fed the process. Conversely, as the flywheel velocity fell, so would the arms, opening the valve. Eventually the system settles out to a set velocity. Van Gelder suggests this may be a more natural metaphor for describing the non-representational changes of state in cognition than fitting them to the Procrustean rack of computation. Others have walked down this path, namely the original cyberneticists like Ashby [4] and Wiener [88], as well as latter-day biophysicists like Haken [36], Kelso [44], Maturana and Varela [54].

Why is this desirable? Besides being a venerable and productive branch of mathematics, dynamics offers a set of rigorous analytical tools to explain natural phenomena. For instance, emergence can be often explained through the process of bifurcation[79]. Briefly, bifurcation is a change in stability of at least one fundamental mode of the system.[2] Examples include column buckling and the onset of turbulence in a heated liquid [1]—qualitative changes in the behavior of a system.

[2]A pole of the system crosses the imaginary axis, if you prefer the frequency domain; or an eigenvalue crosses the imaginary axis, if you prefer a state space approach.

Dynamics may be a powerful tool for understanding a wide range of phenomena in science and engineering, but the requisite sophistication in abstract mathematics can be a barrier to entry. For instance, at MIT, an undergraduate must learn single and multivariable calculus, linear algebra, differential equations, and probably freshman physics (mechanics) for it to really sink in. [3] The textbooks by Abraham and Shaw [1] illustrate the possibility for multimodal learning to cut straight to the conceptual matter without a long list of prerequisites. They employed techniques from comics [55] to render the geometry of complex manifolds using large panels, a few simple colors and descriptive text. However, this presentation is deceptive—the concepts range from fundamental (attractors, phase space) to the fiendish (fractals, chaos). Imagine how much more convincing the material would be if the students could wrestle with interactively controlled animations, which they could program on the fly... Microworlds are just one way for students to come to terms with the complex material in a much more personal and interactive way, which will make the abstract math more accessible when the time comes. With the addition of a virtual oscilloscope to the RobotWorld framework, the dynamics underlying its physical simulator will become more visible, and permit an alternate understanding of behavior in robots and other systems.

2.3.2 Artificial Life

For the theoretical biologist, the synthetic methodology of a microworld is the only recourse other than mathematical analysis. A complex web of interdisciplinary research has grown over the past few decades under the umbrella term "artificial life," referring to the study of systems that are "interpretable as lifelike" [10]. This active area of research has been spurred on by the interrelationships between ethology and evolution [32, 56, 20, 43], and decentralized, emergent, or self-organizing behavior [25] as well as advances in mathematical modeling, optimization and learning techniques

[3] Courses 18.01, 18.02, 18.06, 18.03, and 8.01, respectively.

[41], including cellular automata [6], genetic algorithms [46], and neural networks [38]. Why is this promising?

Through the modeling process, the researcher can make discoveries about his or her own thinking process—and possibly even debug erroneous assumptions about the world. For instance, it is easy to believe that traffic jams must be caused by a "seed event," like an accident, when in actuality they are often the product of the dynamics of collective driver reactions and irregularities in vehicle speed and spacing [65]. Michel Resnick describes some of these experiences with high-school students and undergraduates with his StarLogo system. His constructionist approach allows them to grapple with decentralized phenomena by designing incremental models—which might or might not be valid—but provide a series of stepping stones to deeper understanding of natural processes.

Another compelling learning scenario lies in the cybernetic thought-vehicles of Valentino Braitenburg [11]. As described more fully in Chapter 10, very simple arrangements of sensors and motors on a vehicle can result in very complex patterns of motion, depending of course on the complexity of the environment. Analyzing the movements of these vehicles without knowledge of their internals would result in a complex hobbling together of many intertwined attributes, and perhaps some projecting on the part of the researcher. One can easily synthesize a vehicular design that will perplex analysts for years to come. This is the principle of uphill analysis and downhill synthesis [11], which suggests why cognition and the origins of cellular life are so elusive. Indeed, this might be the defining mantra of the Artificial Life movement: microworlds can tell us more about our own world.

Computer scientists wishing to utilize microworlds will always be drawn to simulation rather than physical experimentation: after all, building robots requires adequate funding and other resources. But Artificial Life advocates stress that young scientists must take precautions not to become too overzealous of the power of simulation.

Rodney Brooks [14] repeatedly criticizes the vastly simplified models of agents designed by computer scientists: "Simulations are doomed to succeed." By idealizing significant details such as sensor noise, agent dynamics, or morphology, he argues, these scientists declare premature success, end up with a naive understanding of the complexity of the target domain, and often fail to produce working counterparts to their simulated robots. Taylor and Jefferson [81] are less caustic, but suggest tempering enthusiasm with scientific rigor by attempting to validate the models with real data drawn from natural processes. Indeed, some within the Artificial Life camp argue that its primary value to the scientific community lies in its service to theoretical biology in the ability to complement more analytical methods, especially with regard to phenomena that are difficult to reproduce in the lab [22]. This is where Artificial Life shines.

By offering a new synthetic methodology, Artificial Life can lead the way not only to new scientific discoveries, but new ways of learning science. In brief, this methodology requires the specification and simulation of interactions between many agents. The goal is to obtain high-level behaviors from low-level interactions. The tough part is teasing out and including in the model all the relevant variables for the high-level behavior to emerge. Note that this reveals the reductionist nature of synthesis[10]. We still purport to explain the target phenomena from the components and interrelationships in the model, only the model has grown to include more aspects of the agent-environment system than perhaps in the simpler and analytically tractable models. Once there is a basis for scientific inquiry, instructors can offer model-making as a means to acquiring knowledge.

2.3.3 Behavior-Based AI and Situated Robotics

There is only one way to learn robotics: by building real-world robots. Rodney Brooks [14] dogmatically stresses this point of view. There are simply too many real-world problems that get glossed over by naive simulations. However, funding is short and RobotWorld is for free. And fortunately, since there is a clean separation between

client-side behavior modules and the server-side simulator, real robots can be integrated into the RobotWorld system by swapping out the simulator and providing robot driver code. What would remain are the behavioral design and code editing layers, as well as the architecture layer which allows the behavioral entities on the student workstation and the real-world robots to communicate. This use of RobotWorld is fully described in Chapter 8.

Microworlds like this one can provide an alternative framework for research in artificial intelligence (AI). The classical AI approach might be characterized as tackling the core problems from the top down, relying on personal reflection in the case of planning, positing representations of the real world in the mind to explain autonomy, and using computational strategies to manipulate these symbolic representations [69]. In contrast, the behavior-based approach tackles the core problems from the bottom up via the processes of evolution, where plans are dynamic patterns of behavioral activation [2], autonomy is the product of the agents morphology, sensor-actuator dynamics, and the complexity of the environment in which is it situated [75], and dynamical, connectionist, or evolutionary strategies are used to generate this behavior. This should not be taken as a definitive explanation, but an illustrative sweep at the philosophical differences between the two approaches.

In more concrete terms, RobotWorld draws from previous behavior-based work including design languages such as REX [42], and the Behavior Language [13] that enabled the subsumption architecture [12], as well as more conventional work in feedback control systems [62, 29]. It was also inspired by the work of computational neuroethologists such as Cliff [16, 15, 17] and Beer [8, 7] who use numeric simulation and analysis of neurodynamical structures to explain animal behavior. Other more basic robotics work also guided RobotWorld's design, including the designs of Jones and Flynn [40], the Braitenburg vehicles designed by Hogg et al. [39] using Programmable Bricks, Khepera robots [57], and especially the LEGO robots of competition fame.

Every January, students at MIT have a chance to participate in the 6.270 Autonomous LEGO Robot Design Contest, which is an intense and highly publicized event in which teams design, assemble, and program a fully autonomous robot using Lego pieces and a healthy complement of motors, sensors, batteries, and other materials. Students write multithreaded code which is run on an on-board microprocessor, and is completely responsible for robot behavior during the competition. These fully autonomous robots compete, tournament-style, on a difficult playing field containing obstacles and props such as foam blocks, barrels, moats, drawbridges, and ping pong balls. The specific goals change every year, but the overall aim is to build systems that behave consistently better than one's opponents' (as well as to get hands-on electronics, programming, and systems experience, develop teamwork skills, and to have a blast) [51].

This contest captures in a nutshell many of the difficulties intrinsic to robotics and embodied artificial intelligence in general: navigation, object manipulation, object avoidance, action selection, and goal-directedness in complex, dynamic environments (to name a few) [50]. In addition to engineering concerns (timely, reliable, and effective behaviors), it also captures another set of issues related to the design of interactive software: control loops and conditional dispatch, multithreaded programming, parallelism, robustness, functional decomposition, interprocess communication models, state and synchronization, incremental design techniques, and the software lifecycle [77].

In light of this experience, RobotWorld has been designed to support the collaborative design of agents much like the LEGO robots of 6.270, except that the focus is on the programming aspects. The software will be fully exportable and considerably less expensive (a 6.270 kit costs around $600 at the time of this writing), so that a wider audience may incorporate the material into their own curricula. The agents and their environment are programmed using Java technology in conjunction with

a distributed computing model, allowing an immense design space to be explored relative to the limited time and finite resources of the LEGO contest.

However, RobotWorld should be contrasted with "real" robot simulations. The two have very different goals. "Real" robot simulations attempt to create a physically realistic model of a robot for applications in science and engineering. They provide one or two layers of observation which model as much complexity as the task at hand demands. They use sophisticated math packages such as MATLAB, which require comfortability with advanced mathematics and control theory. They would not really be suitable for a CS101 class. On the other hand, RobotWorld was built for introductory computer science education. Its goal is to expose programming concepts by the use of a robotic metaphor. The application was built in a way that constitutes many layers of observation within the robot, encouraging deeper thinking about all the processes that must happen concurrently to make it run. The physical model of the robot leaves out many real-world details, but includes just enough to encourage interesting behavior in a community of interacting agents (e.g. inertia, damping forces, collisions, sensors/actuators and a radiation model). This educational goal is similar to that of Michel Resnick's in creating StarLogo [65]: to give students enough complexity to explore interesting ideas and to create their own models of reality, without drowning in the the morass of real-world issues that a professional scientist or engineer would have to face. The goal is not to learn the "correct" model of the world, but to learn the process of creating and validating models, which every scientist and engineer must do throughout their academic career.

2.3.4 Computer Graphics

On the opposite end of the spectrum from roboticists, we have the graphics community, which strives for a kind of realism in simulated worlds. Lately, this work has borrowed heavily from dynamical modeling, artificial life, behavior-based AI, and robot kinematics, as well as anatomical modeling and psychophysics to animate virtual characters [5]. Some promising directions include the simple but effective flocking behavior of Reynolds' boids [66] and the schools of virtual fish generated by Terzopou-

los et al. [82] that modeled evasion, pursuit, feeding, and reproduction all at once. Some of the example projects later in this thesis follow in this tradition, albeit in a less sophisticated form.

In its present incarnation, RobotWorld is necessarily simple: a 2D world inhabited by radially symmetric agents. However, it uses the techniques of physically-based modeling [89] to impose forces and torques on the agents as they interact with each other and the environment. This should provide a rich enough space of interactions to satisfy the goals of the project, but an enterprising researcher could extend RobotWorld to 3D [24] or perhaps suggest it as a class project. There has been some work by Reynolds [67] and Sims [74, 73] in using 3D articulated geometry to explore the dynamics of artificial creatures. Such a system would not be incompatible with the present RobotWorld architecture. Also, the naïve collision detection and reaction system could benefit from the techniques of computational geometry [89]. In summary, RobotWorld could serve as a medium for exploration in the animation of simple agents, and any work in this area could also benefit other RobotWorld users by adding to the richness and complexity of the agent-environment interactions.

2.3.5 Object-Oriented and Concurrent Programming

RobotWorld descends from a long line of research projects involving object-oriented and concurrent programming. One close cousin is Karel the Robot [60], an anthropomorphic agent in a discrete-time, discrete-space microworld used as a means of teaching Pascal. It was the intent of RobotWorld from early on to avoid the restrictiveness of the Karel world, which bears a resemblance in its simplicity to the much-criticized blocks world [69] which was a staple of classical AI education in the early years. Instead, RobotWorld strived to be continuous-time, as opposed to being turn-based, and continuous-space instead of being grid-based. RobotWorld offers much more potential richness and complexity, since many robots are supported, new objects can be added to the palette at any time, and many layers of observation are possible. This includes building your own GUI components and communicating across

the network. This allows for a much more complex and interesting set of interactions to take place, as well as a different context for design. Robustness and responsiveness become much more important in a dynamically changing world! [3]

Communication between entities is a central feature of RobotWorld. Older systems such as Act 1 [49] relied on message-passing as a mechanism for distributed concurrency. One benefit of such an approach is the consistent style of programming: everything, including messages, are actors, where an actor encapsulates both data and procedures into one entity. Concurrent Smalltalk [83] supports a similar model, exchanging delegation for inheritance and adding some constructs for synchronization. One significant variant is Dally's Concurrent Smalltalk, which supports the concept of a "distributed object." This is an aggregration of many objects which exhibit the same behavior, but have different local state. To an external observer, it is one unit. Within the distributed object, any constituent object may field an incoming request, and the contract is non-deterministic. The blackboard systems developed by David Gelernter [31] echo this flexibility, and add persistence and template-matching. By communicating via a passive third party, many distributed entitites can interact in a loosely coupled fashion without knowing who will read their messages and without caring about whether anyone is connected to receive them. Many heterogenous entities can respond concurrently to a single message, as well. This permits a very different style of programming and allows a great space of possibilities to be explored.

Lately, Java has become popular as a first professional language partly because concurrency and object-orientation are built in. When covering the basics of expressions, statements and control structures in a CS101 class, it is not much of a leap to fields, methods, inheritance and interfaces. At this point, do you unfold the grand scheme of concurrency in the use of threads, synchronization, exception handling, streams and sockets, and the design of continuously running services? Or do you leave it for them to discover as upperclassmen? An early exposure to interactivity might well prepare students for further work in computer science [76], either by mak-

ing a deeper treatment of the material more accessible [48] or by emphasizing how interactive services play a role in society and the economy at large [34].

2.3.6 The Software Development Process

During an introductory computer science course, students will have imprinted the programming habits that will stay with them, for better or worse, until they enter the workforce. Even then, they will find it easy to slip back into old habits, such as neglecting to comment code, or adhering to the "big bang" model where copious amounts of functionality are thrown together and expected to work without incremental testing. As they say, an ounce of prevention is worth a pound of cure. The form of this prevention is up to the instructor, but a recipe that has worked for our project is to consistently encourage students to design incrementally. This means to decompose a programming project into a set of easily testable functionalities, and then order these by dependency. This requires careful thought and planning. Our approach loosely resembles iterative development practices discussed by methodologists [28] but we have not explicitly borrowed the terminology and concepts. A look at the Unified Modeling Language (UML) might be warranted, since this is being widely adopted as a meta-language for discussing object-oriented design and analysis. Another possible good practice is to introduce some of the simpler Design Patterns [30], so that students are aware that they do not need to continuously reinvent the wheel. It would be unusual to introduce code and design reviews, but this is an effective mechanism in industry and would allow the students to become articulate about their work and the design decisions they made. Lastly, there is nothing better than to talk about how software development goes wrong, especially as they near a large final project. Between all of these options for preventative pedagogy, there must be an approach that potential RobotWorld users will find acceptable.

Now that you have a broad idea of the design choices behind RobotWorld and the goals for its usage, we can turn to the system itself—first a quick overview, and then an in-depth coverage of the application framework.

Chapter 3

A Quick Architectural Overview

RobotWorld is a Java application which serves as an extensible integrated development environment and networked simulation tool. Visual metaphors guide its construction wherever possible. It can also serve to supplement the constructive idiom with analysis, software design, and communication tools. This virtual world constitutes several layers of observation for the student: a network diagram of the activity on the blackboard [31]; a playing field in which the agents interact, Braitenburg-style [11]; the agent's physical model; the agent's sensorimotor connections and behavior modules, inspired by Brooks' behavior language [13]; and most importantly, the code responsible for each of these components. Figure 3-1 illustrates how these layers fit together.

RobotWorld is based on a framework for distributed computation known as a *blackboard system.* These systems permit object entries to be written, read, or erased from a common object store (the 'blackboard'). They can be retrieved by the use of template matching, where specified fields are required to match exactly and unspecified fields are treated as wildcards. This allows for loosely coupled inter-object communication, where the sender and receiver need not know each other's location (IP address), and don't even need to be synchronously connected. Note that a blackboard system is not an object database. All entries which match a given template are considered equal, and any one of them might be returned. In other words, the

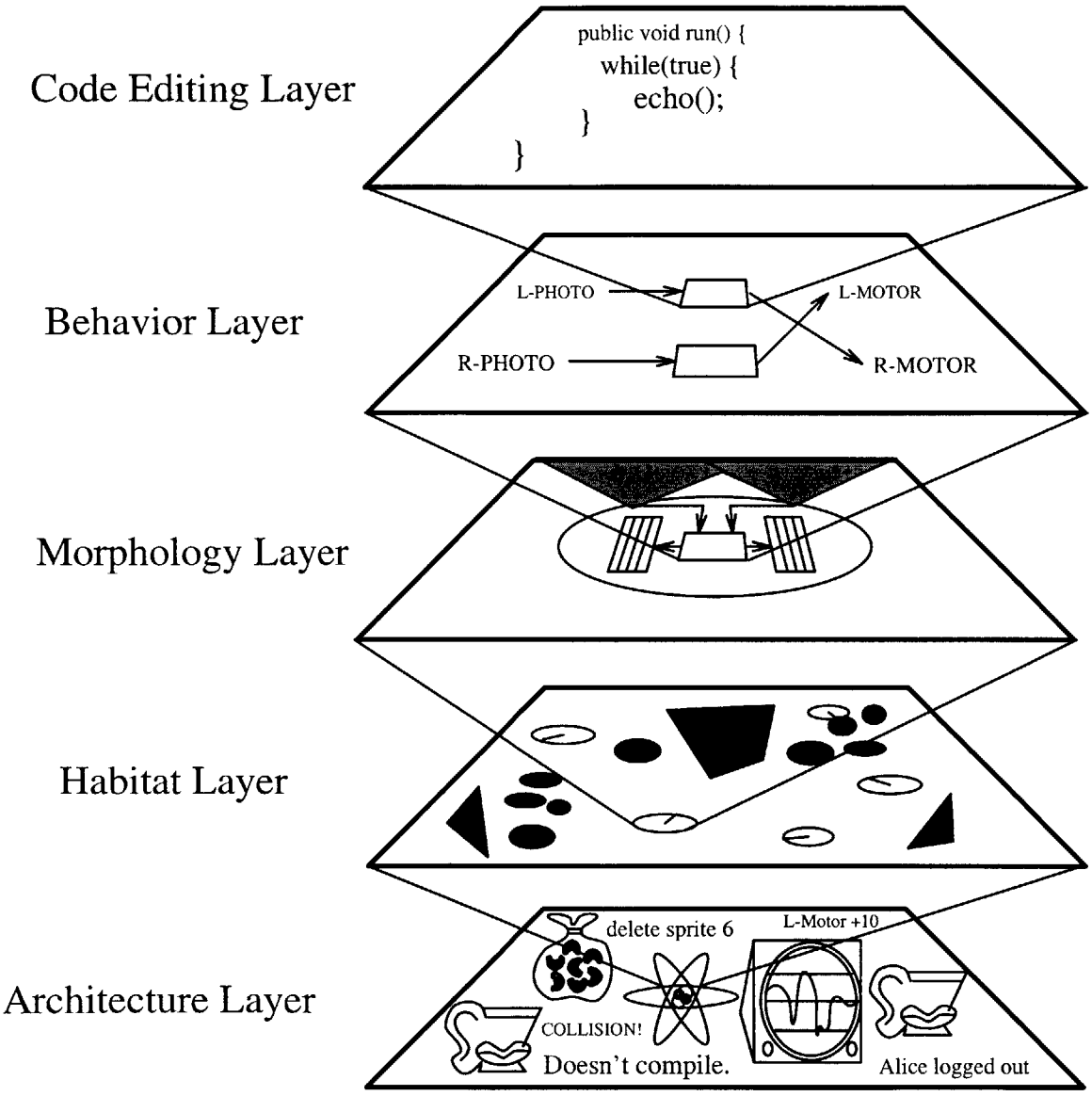


Figure 3-1: An exploded view of RobotWorld, showing the five layers of observation. The Architecture Layer shown here attempts to depict a snapshot of the activity on the blackboard.

contract is nondeterministic, even if a particular implementation is not.

This framework enables the design of scalable distributed applications. A wide array of client/server relationships can be represented by a standard entry with five fields. The entry can be read as a request: "[Server] in [Domain], please [Verb] [Subject] for [Client]." There are two basic protocols used in RobotWorld; the first is a one-shot request for a service to be performed. The client writes the request to the blackboard, and the server takes the request from the blackboard. The server can write a response back to the client if necessary. The second protocol allows the server to serialize application state to a common area which remote observers can poll for the latest data. Updates to this application state can be made either directly, by the server, or by a client using the first protocol to make a request of the server.

These protocols can be used to build a *collaborative workspace.* In RobotWorld, this includes basic services such as loading, saving, and printing files, as well as editing and compiling. Most integrated development environments (such as Visual J++ or Emacs in java mode) have special features such as auto-indentation, brace matching, syntax coloring, and reflection which speed up the learning curve. However, these applications are usually platform-dependent, as are the compiler toolkits (such as the JDK). Using the protocols described earlier, one could make a request to compile of a server with access to the platform-dependent service. The class file or error log could then be written back to the client. The final step includes identifying class files (beans) which are to be associated (via beanbags) with the various presentation layers of a running RobotWorld application.

As described earlier, RobotWorld presents a visual *graphical user interface* in which objects can be added, removed, dragged, opened, and inter-connected. These specialized workspaces can be used to depict either a static data flow diagram, or a playfield full of animated agents. Two distinct kinds of entities are required to do this—sprites and layers. A sprite is an object which knows how to paint itself onto

a canvas. A layer is just the class of workspace that is specialized for interacting with a given class of sprite. Sprites can be recursively nested: each sprite can have many children, but only one parent. The resulting data structure is a tree of sprites. In the RobotWorld GUI, a user can traverse the tree by double clicking on a sprite. A workspace window corresponding to the sprite's layer will pop up, rendering all of that sprite's children. The user can continue clicking on sprites until they reach the bottom of the tree. The sprite tree state is maintained by the blackboard, and it is the responsibility of the workspace to poll continuously for updates (e.g. for animation) and post change requests based on user input.

The above structure is sufficient for animating the sprites in a playfield and drawing data flow diagrams, but the sprites are isolated. They still need a *data transfer mechanism* for client-server communication. The basic diagram elements–sources, connectors, and sinks–can be extended so that they actually pass data from source to sink. By default, they transfer data locally. A dual[1] set of elements negotiate data transfer remotely using the blackboard. Note that a source may have many sinks, but a sink can have only one source, as in the Java event model. This helps keep synchronization and computation strategies inside sprites, where animacy is understood to reside for the purpose of student exercises. Custom behavior is handled by embedding sources and sinks within a self-animating sprite, and by invoking their methods. The rest is automatically handled by threads within the sources and sinks which push or pull the data, as appropriate.

The final component of the system is the *physics simulator* itself. The simulator's job is to interactively update the vehicle state using dynamical modeling techniques. Briefly, this means that an ordinary differential equation is solved for the position and orientation of each vehicle using numerical integration. The benefit of this method is flexibility; force laws can be added or altered interactively. Other major functions of the simulator include incorporating actuator commands into the dynamic model,

[1]This source of this local/remote duality will become evident in Chapter 7.

generating vehicle sensor input using a simple radiation model, and handling collision detection and response between solid objects. The clean separation of simulator physics from controller logic means that replacing the simulator with a real robot (or community of robots) should be relatively straightforward. If this were done, RobotWorld could be used for interactive robot monitoring and control as well.

These five system components—a blackboard system, a collaborative workspace, a graphical user interface, a data transfer mechanism, and a physics simulator—make up the architecture of RobotWorld. Each component corresponds to a Java package. Chapters 4–8 detail the conceptual background, core interfaces, and current implementation for each of these packages. Be forewarned that there is not a direct correlation between the five packages and the presentation layers shown in the diagram. Chapter 9 describes these presentation layers, and how these are built from the framework of the five packages.

Chapter 4

The robotworld.net Package

4.1 Background: Blackboard Systems

A simple blackboard model provides the connective glue underlying the entire RobotWorld system, and allows an ensemble of agents to interact regardless of whether they exist in the same Java Virtual Machine, or on far-flung networks across the globe. The interfaces were designed to be clear enough that a previous nonprogrammer, at the end of an Interactive Programming-type course, could build one of any number of networked applications for a final project. [1]

This blackboard model permits the same general operations as the Linda systems developed by David Gelernter [31], with the idiomatics of JavaSpaces [80] in mind. Blackboard systems allow loosely coupled inter-agent communication by permitting agents to write, read, and remove entries from the blackboard. The sender and receiver don't need to be on the same machine, and don't need to be synchronously connected. By providing a persistent passive repository, blackboard systems relax the tight coupling required by traditional message-passing systems [49], and allow a different approach to distributed computing [31].

[1] Of course, there is a certain pride one feels after mastering the gritty details of socket programming... and since the interface does not necessitate the use of RMI, requiring the students to implement their own socket-based solution is perfectly acceptable. But I digress.

Persistence Model	Relation	Tuple	Attribute	Values
Relational Database	Table	Row	Column	Data
Object Database	Collection	Object (unique)	Field	Object (unique)
Blackboard System	n/a	Entry (copy)	Field	Object (copy)

Table 4.1: How blackboard systems and databases compare.

In a sense, blackboard systems resemble database systems in that they attempt to satisfy persistence and transaction safety via the ACID properties (atomicity, consistency, isolation, and durability). However, blackboard systems differ from both object and relational databases. See Table 4.1. Object databases preserve object identity, whereas blackboard systems operate only on copies [80]. Relational databases permit general querying on a field, whereas blackboard systems are restricted to either an exact match or a wildcard. The central difference, though, is in the intent: to permit a distributed application to be modeled—and implemented—as a "flow of objects" through a set of clients and servers.

4.2 Core Interfaces

See Figure 4-1 for an O'Reilly style class diagram[2] of everything in this package. Conceptually, the RobotWorld Blackboard maintains a list of BlackboardEntry objects as well as a list of BlackboardListeners. Classes wishing to read() a BlackboardEntry must first provide a template BlackboardEntry to the Blackboard. If the template matches a stored entry, that entry is returned. The process is similar for take(), except the entry is first removed from the Blackboard. Classes wishing to be notified of writes must provide a BlackboardListener and a template using Blackboard.listen(). Every time someone writes a new BlackboardEntry onto the Blackboard using write(), all listeners with a matching template will receive a BlackboardEvent using Blackboard-

[2]The O'Reilly Java series of books [23] use a concise notation to illustrate class inheritance, interface implementation, and package structure.

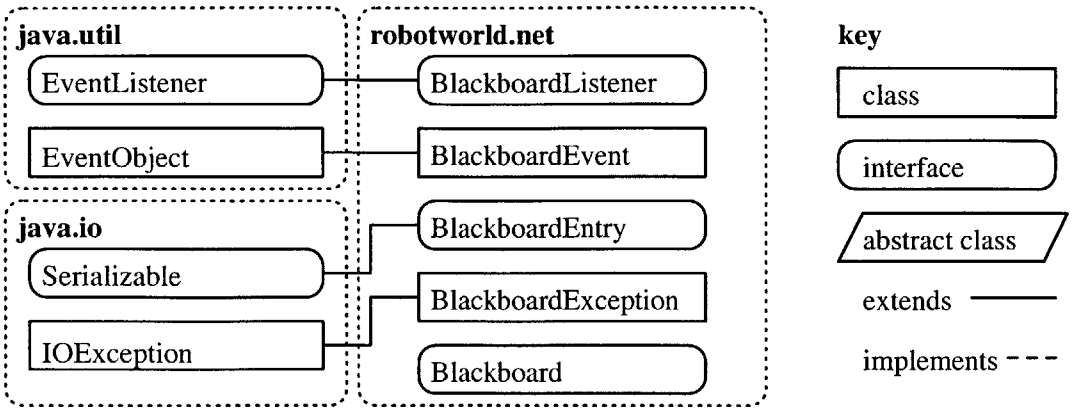


Figure 4-1: Class diagram of package robotworld.net

Listener.notify(). Finally, any of these operations can throw a BlackboardException, a refinement of IOException, if something goes awry.

Objects implementing BlackboardEntry must take the same precautions as any other serializable object, plus additional design constraints. To facilitate its use as a template, an entry must have a no-arg constructor and no public fields of primitive type. Also, the public fields of a BlackboardEntry must be either references to Serializable objects, or null, indicating a wildcard. Generally speaking, an entry can be retrieved from the blackboard by providing a template object, which is matched against candidate entries field for field. If a template provides a non-null value, that value must be matched exactly. Otherwise any candidate entry may be returned. More specifically, a template can only successfully match against an entry on the blackboard when the following conditions are true:

- The type of the entry is assignable to the type of the template.
- Each of the template's public fields must either be null, or bytestream-equivalent to the corresponding field of the entry when serialized.[3]

[3]One might have expected the use of object equivalence, via Object.equals(). The implementors of JavaSpaces were concerned about the binary compatibility of evolving serialized objects, and use java.rmi.MarshalledObject.equals() instead.

Without distributed events, the blackboard model would only support pull-based flow. With them, the blackboard model can support both push and pull-based flow. This adds flexibility to the design of RobotWorld components, which can handle unexpected, low-volume events such as agent collisions, the arrival of error messages, or requests for services. At the same time, other components can poll the blackboard for continuous, high-volume sensor and actuator data at their own pace.

This requires the use of a BlackboardListener, which has one method, notify()[4], which is passed a BlackboardEvent that records the source of the event, an event descriptor, and a monotonically increasing sequence number.

So why go to all the trouble? Compatibility. Multifarious services, whether implemented now or unanticipated add-ons, can communicate using a simple, extensible template. The required fields are: Server, Domain, Verb, Subject, and Client. The message is to be read as follows:

"[Server] in [Domain], please [Verb] [Subject] for [Client]."

Some possible uses:

- "Anybody in the Habitat Editor, please edit Agent NX-9 for pepsi.ai.mit.edu." This request would pop up a window which allows the user to tinker with the innards of Agent NX-9.
- "Everybody in the Morphology Editor, please delete the Left Photosensor for yourself." This would cause all appropriate sprites as well as all the nested SpriteWindows and CodeEditorWindows related to the Left Photosensor to disappear (prompting the user to save any necessary files, of course.)

[4]Don't confuse this with Object.notify(), no arguments, which is used for a completely different purpose. BlackboardListener's notify method takes one argument, a BlackboardEvent, which distinguishes it from Object.notify().

- "Anybody in the Code Editor, please compile Foo.java for me." This might be a remote server with a platform-dependent compiler, which could post the .class file (or any errors or warnings) on the blackboard.

- "Ping, wherever you are, please echo-to-the-user 'THINK FAST' from Pong." Here Ping's agent requests Pong's agent to display a message in the Chat console.

- "Everybody monitoring State Space, please take note of the state of Agent NX-9 for yourself." This is how in fact sensor and actuator data are updated to the blackboard. Of course, all the myriad physical quantities will be written separately to the blackboard as well, since they might be updated at different rates, which will influence the design of their listeners.

There is a utility interface upon which most RobotWorld communication hinges, which is WorkspaceEntry. Implementing classes must provide getter and setter methods for each of the following: server, domain, verb, subject, client, all of type Object.

WorkspaceEntry is not an abstract class because the primary implementation, JavaSpaces, provides an AbstractEntry class with reference behavior. For the JavaSpaces implementation, having WorkspaceEntry subclass AbstractEntry would be sufficient. However, this would introduce a dependency on the JavaSpaces package which would be unacceptable. And since Java does not support multiple inheritance, an interface was chosen instead. [5]

4.3 Current Implementation

The RobotWorld distribution comes with two flavors of Blackboard: a simple, naive, shared address-space server for use on a single machine ('SimpleBlackboard'), and a

[5] Also note that an 'AbstractBlackboardEntry' could have been introduced, with behavior mimicking the reference behavior in AbstractEntry. Since this class would provide the fundamental basis for RobotWorld communication, it would have to be subjected to thorough testing for correctness, and tuned for optimal performance. Then WorkspaceEntry could just subclass AbstractBlackboardEntry. However, this seems like a lot of work to promote a ten-method interface to a five-field class.

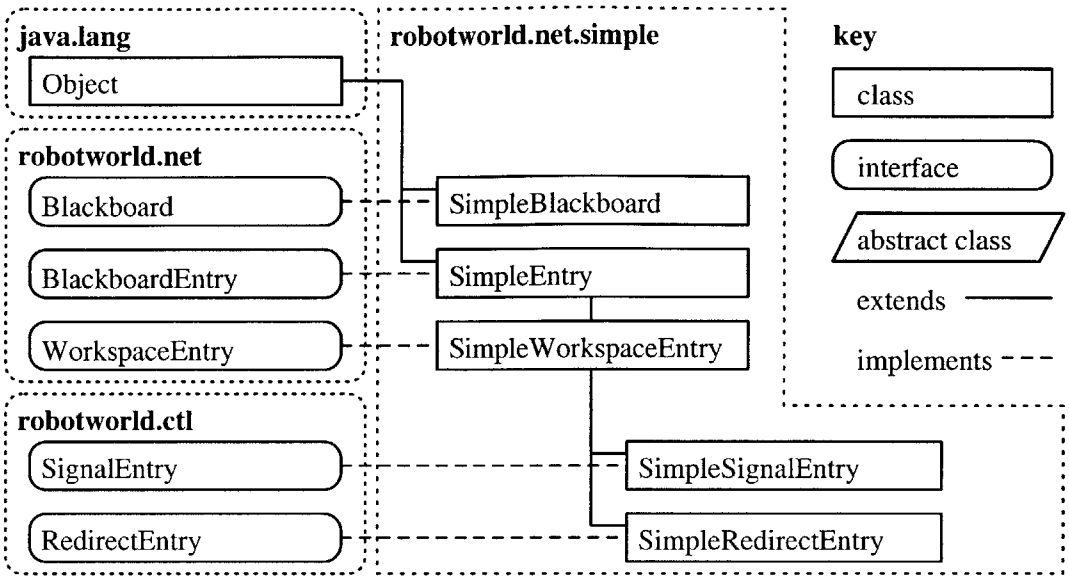


Figure 4-2: Class diagram of package robotworld.net.simple

full-blown Jini/JavaSpaces client('JiniBlackboard'). The flavor to be used is determined by the properties file provided on the command line at startup.

4.3.1 SimpleBlackboard

This version was designed to be independent of any vendor libraries, so that instructors wishing to evaluate RobotWorld with a single download could do so. Due to Sun's software licensing, JavaSpaces must be downloaded directly from the Sun website, complicating the RobotWorld installation process. By providing an implementation based on the core API, RobotWorld is also somewhat insulated from unanticipated problems: Sun might choose to charge for JavaSpaces, discontinue the project, drastically alter the API, etc. It also serves as a reference for testing and debugging applications depending on either JavaSpaces or another implementation, perhaps a socket-based version developed as a class project.

It's quite simple; synchronized, but not too efficient. Reads and takes require $O(N)$ time, where N is the number of entries stored. The entries on the blackboard are iterated over, until a match is found. In the case of a take, the entry is removed.

Writes are $O(L)$, where L is the number of listeners registered. All listeners are iterated over, and those with matching templates are notified. Registration requires $O(1)$ time, where a map entry is added with the listener as the key and the template as the value. The only other interesting behavior is in the class method matches(), which returns true if the given template matches the given entry. The logic governing this method was derived directly from the behavior of the AbstractEntry class as described in the JavaSpaces specification, and uses the Reflection API to accomplish this behavior. Briefly, there is a match if: the template class is assignable from the entry class, and for each field, either the template specifies a wildcard, or the values are equal in the sense of equals(). Otherwise, there was not a match.

4.3.2 JiniBlackboard

This implementation serves as a wrapper (object adapter) around the Jini/JavaSpaces specification developed by Sun Microsystems [80]. Most blackboard calls simply delegate the work to a net.jini.space.JavaSpace implementation. One difference is where listen() is called. A second wrapper is needed to deliver events via BlackboardListener.notify() from the remote event model–a feature of Jini/JavaSpaces which we desire to keep abstract from the RobotWorld user. Therefore we have JiniListener, which implements the RemoteEventListener interface and extends UnicastRemoteListener (for simplicity). JiniListener, also an object adapter, maintains a BlackboardListener instance. Whenever a remote event is delivered via JiniListener.notify(), it generates the appropriate BlackboardEvent and delivers it to the BlackboardListener.

Finally, the entry hierarchy deserves explanation (yes, there is method to the madness). Out of a desire to subclass net.jini.space.AbstractEntry and gain the functionality therein, a JiniEntry was designed which does so and also implements BlackboardEntry. Accordingly, a JiniWorkspaceEntry subclass provides workspace functionality by implementing the WorkspaceEntry interface. Note that WorkspaceEntry refines the BlackboardEntry interface, so that a JiniWorkspaceEntry is twice a descendant of BlackboardEntry. This relationship (between BlackboardEntry and

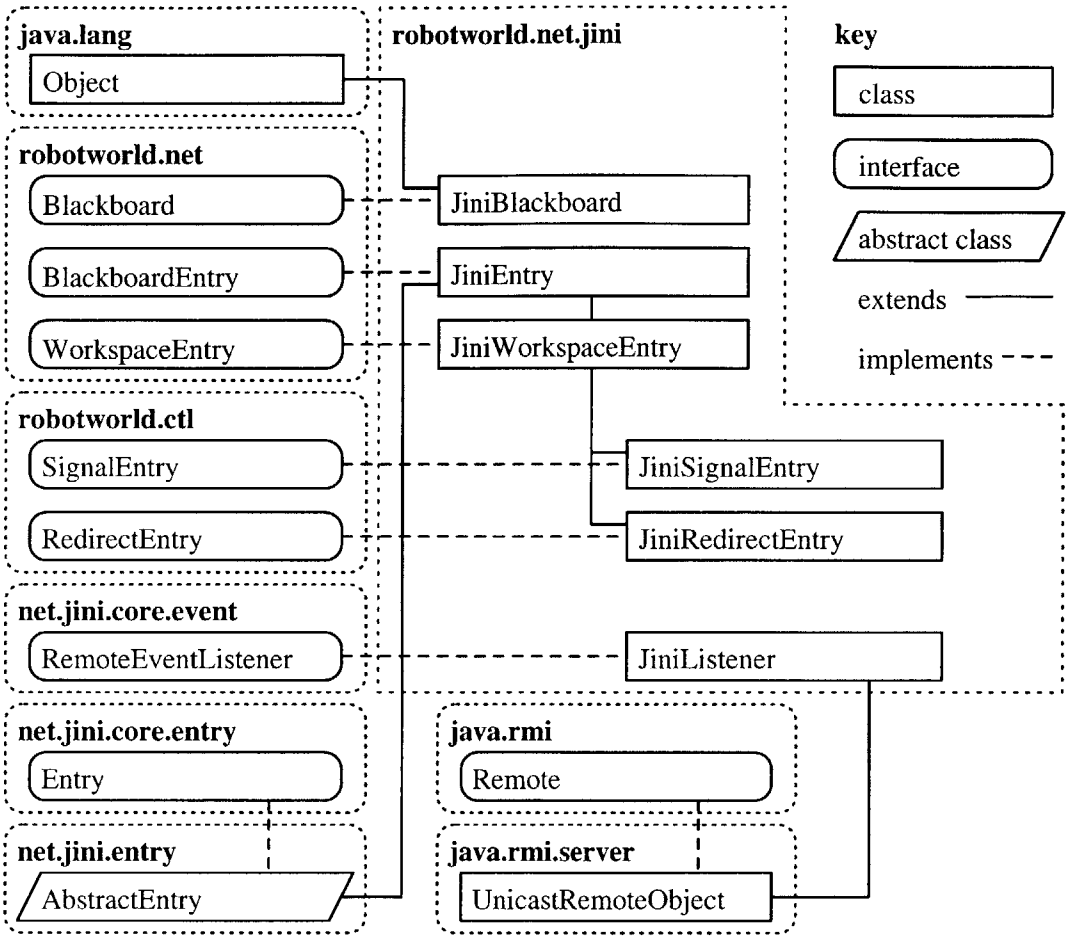


Figure 4-3: Class diagram of package robotworld.net.jini

WorkspaceEntry) was designed to tighten inheritance requirements for future classes, since a WorkspaceEntry does not have to descend from a BlackboardEntry implementation. Implementing an interface multiple times does not pose any problems in the Java language.

Chapter 5

The robotworld.ide Package

5.1 Background: Development Environments

When properly designed, integrated development environments, or IDEs, can often mean a significant productivity boost for experienced programmers and a decreased learning curve for beginning programmers. The essential idea is to aggregate the many functions associated with software development into one unified framework of applications. Typically these applications include a text editor, a compiler, a debugger, and perhaps a visual GUI builder or hooks into a configuration management system. To ease the programmer's way, features are added (syntax coloring, auto-indentation) or abstracted away (compiler flags, directory structure). All too often, however, commercial IDEs become subject to feature bloat and can swamp the beginning programmer in a sea of wizards, menus, and company-specific jargon. Concern for Java programmers in particular is library lock, when an IDE is hard-wired to support a particular JDK and no others, and platform incompatibility, made possible by native code software components and compilation options. As a result of these issues, instructors who wish to benefit from IDEs while minimizing the drawbacks will have to be selective, and may have to sample several IDEs before deciding which is best for their purposes.

With this in mind, RobotWorld provides rudimentary IDE functions within a

unified workspace, yet allows instructors to hook into their favorite IDE from RobotWorld with a minimum of work. The base case editor is designed in pure Java. Unfortunately, all compilation is platform dependent. This is inevitable until a stable equivalent of the sun.tools.javac.Main class is released[1]. If this were to happen, it would permit class compilation within the Java virtual machine. Fortunately, requests to compile (or edit) can be transmitted to another machine via the blackboard, localizing the platform dependence if the instructor so desires. Also, hooking into an external IDE does not require any Java programming, only a modification of the command strings in a special properties file. The instructor will need to provide the path and name of the executable, along with any arguments necessary to accomplish the desired task within the IDE. Note that tasks executable in this way are limited by the command-line options of the IDE and the instructor's ability to wade through the documentation to discover them.

5.2 Core Interfaces

See Figure 5-1 for an O'Reilly style class diagram of everything in this package. Workspace is an abstract class which provides centralized functionality for loading/saving text and objects to a file, launching exception dialogs, and closing windows in a consistent way. It also serves as an abstract factory for all RobotWorld classes in its Java virtual machine. It also maintains the blackboard singleton. These are all *static* characteristics of the class, which are available to anyone.

At the same time, concrete implementations of Workspace are the very windows that users interact with. Workspace is based on JFrame, which is the Swing equivalent of a Frame. Subclasses of Workspace are expected to do all the dirty work that is typical of a frame instance. They must implement a common set of menu functions (clear, load, save, saveAs, rename, print, copy, cut, delete, paste) and make those

[1]See the Java 1.2 API for a discussion of why it is a bad thing to release code that uses the sun.* packages

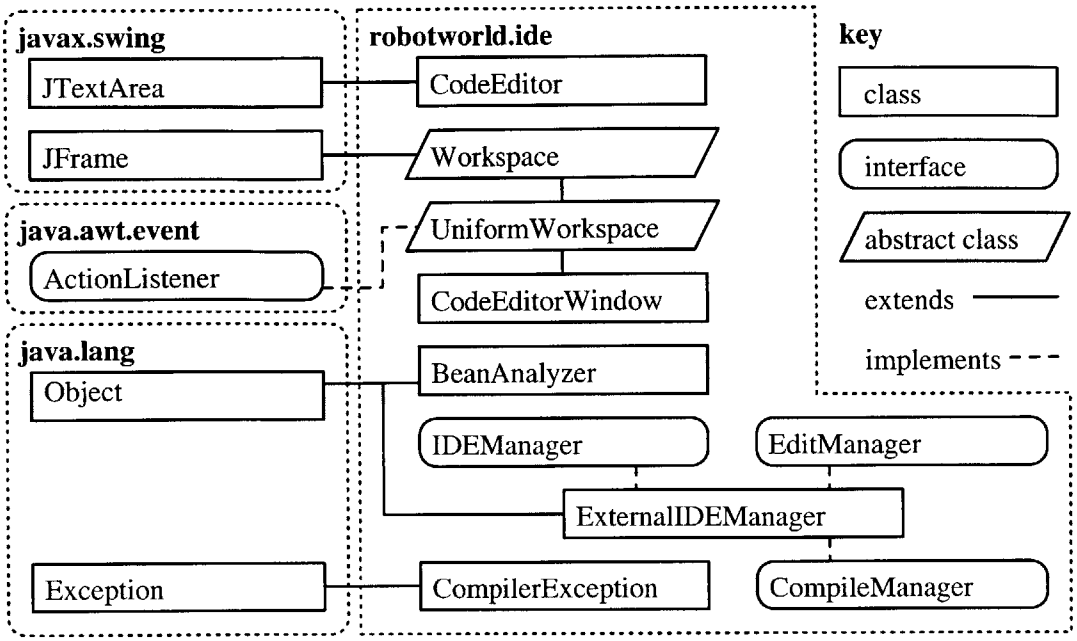


Figure 5-1: Class diagram of package robotworld.ide

behaviors available in a graphical user interface. For instance, UniformWorkspace is an subclass of Workspace that provides a consistent set of menu options, including an action listener that dispatches the request to the proper abstract method (one of load, save, clear, ...) Another important method is configureToEdit(), which takes an argument of type Object and is used to set up a workspace for editing a particular Object instance. [2] Only one Workspace is presented when RobotWorld starts up. Over the course of a RobotWorld session, however, a user may spawn off as many Workspace windows as they wish. Take note of the separation of the class and instance behavior of Workspace. The static class behavior permits RobotWorld as a whole to function. The instance behavior permits one particular window to manipulate its contents, and that is all.

CodeEditor is a simple pure Java text editor with support for auto-indent and brace matching, and CodeEditorWindow extends UniformWorkspace to implement the menu functions (load, save, clear, ...) to control the CodeEditor. This consists

[2]In the current system, this object will always be a robotworld.gui.Sprite.

mostly of calls to CodeEditor's setText(), getText(), and resetParser() methods, but also relies on the built-in file loading and saving functionality of Workspace.

BeanAnalyzer represents the beginning of a larger move towards a class visualizer. BeanAnalyzer uses reflection to break a class down into its anatomical structures: fields, methods, constructors. It was meant to be used with CodeEditor and a BeanSynthesizer (not written, see Chapter 11, Future Work) to encourage a bottom-up approach to class design. In other words, students might be instructed to click on the act() method in a menu, which would bring up a blank CodeEditor text area. They could experiment with writing expressions and statements in this area. When they press compile, the BeanSynthesizer would first assemble their class and then inform them of any problems. Next, they could write an entire method including its signature. Then, they could write an entire class in the CodeEditor. For now, BeanAnalyzer merely breaks down a given class into its component parts as a sort of utility class. Fancy interfaces can come later.

Finally, there are the interfaces for task management: CompileManager, EditManager, and IDEManager. CompileManager is an interface that agrees to take a fully qualified package name of a class, look for it relative to the code base, and return a Class object. If there are any compiler errors, it throws a CompilerException, with the error messages embedded in the exception's message string. EditManager is similar; it also agrees to take a fully qualified package name of a class, look for it relative to the code base. However, it loads the source code into a text editor instead of compiling it. Both of these may throw IOException if anything goes wrong. The third interface is IDEManager, which knows how to find implementations for CompileManager and EditManager.

5.3 Current Implementation

Only one concrete implementation of IDEManager is provided: ExternalIDEManager. It also implements CompileManager and EditManager. For both of these functions,

what it does is call Runtime.exec() and wait for the process to terminate. Any characters on the error stream are collected. In the case of a COMPILE, javac could be used as the command and the given classname as the argument. It is assumed that javac is on the path, and the class file can be found relative to the code base. Any collected error messages are used to generate the CompilerException, otherwise Class.forName() is used to load the newly compiled class into the Java virtual machine.

However, the approach is general and other applications besides javac can be called, for either editing or compiling. This was done in order to provide access to third-party developer tools such as Emacs/JDK or Visual J++. The basic idea is to store string patterns in property files, using a similar process as that used for abstract factories in RobotWorld (see Chapter 9). From the Workspace, a user can request a standard external service (e.g. COMPILE or EDIT). It first looks up the command substitution string in the System properties (which were loaded by robotworld.main at startup), then it performs the substitution, and then it spawns off the external process using Runtime.exec(). All the substitution work is performed by java.text.MessageFormat.format(), which is similar to the C function sprintf(). This procedure allows invocation of any command line accessible executable, with as many arguments as needed. For instance, an EDIT command could cause Emacs to open up with the requested file, and COMPILE could call javac from within Emacs, so that the error highlighting and jump-to-file functionality could be utilized.

Chapter 6

The robotworld.gui Package

Once the blackboard system is in place, and a workspace for editing and compiling has been built, the next step towards an interactive laboratory is to design a visual component framework. These components, or sprites, will mirror the java.awt.* containment hierarchy in that they can be nested within each other, forming a tree. In general, each sprite can have its own window, within which its children are drawn. These will very often be simple graphics—a box with text—or perhaps a more complex shape to depict a vehicle body. These sprites can be added to, moved about, or deleted from their parent window. They can also have their properties edited, or be opened. Opening a sprite causes a new window to pop up, displaying its children (if any). This provides a visual rendition of object encapsulation, but more functionality is needed. Sprites can also be connected. This is accomplished by the use of a set of marker interfaces, and a special canvas that recognizes sprites who implement those interfaces. Now, arrows can be drawn between sources and destinations using a special arrow class. This basic foundation can be extended in the robotworld.ctl.* package to actually transfer data between sprites, either locally via method invocation, or remotely using the blackboard.

6.1 Background: Dataflow Analysis

The appearance of the particular sprites is meant to conform in some sense to the semantics of dataflow diagrams [19, 18, 72, 71]. This software architecture methodology is meant to depict the flow of data, independent of timing and control flow, between modules of a complex system. The diagrams form a tree of recursively decomposed modules, using a few simple primitives and rules for interconnection and layout, with the overarching goal of clarity and simplicity. The components are as follows:

- Terminator. This is an entity external to the system and outside the scope of design, with which the software exchanges data.
- Node. The software as viewed at some level of abstraction. Every node has a unique number.
- Connector. An abstract datum or set of data. It is irrelevant whether it represents a method argument, return value, message, stream, record, or file. Also irrelevant are the client-server model used to deliver the message, the encoding, and the amount of time required for delivery.

Terminators are only permitted at the topmost level ("Diagram 0"), which has only one node, representing the entire software system. Diagram 1 then represents this node, only in more detail, perhaps with three to five nodes (each labeled 1.1, 1.2, ...) Each of these can be recursively decomposed, until the design goals of clarity and simplicity are at stake.

6.2 Core Interfaces

Figure 6-1 shows an O'Reilly style class diagram of everything in this package. Sprite is the central class; it is to robotworld.gui what Component is to java.awt. Sprite

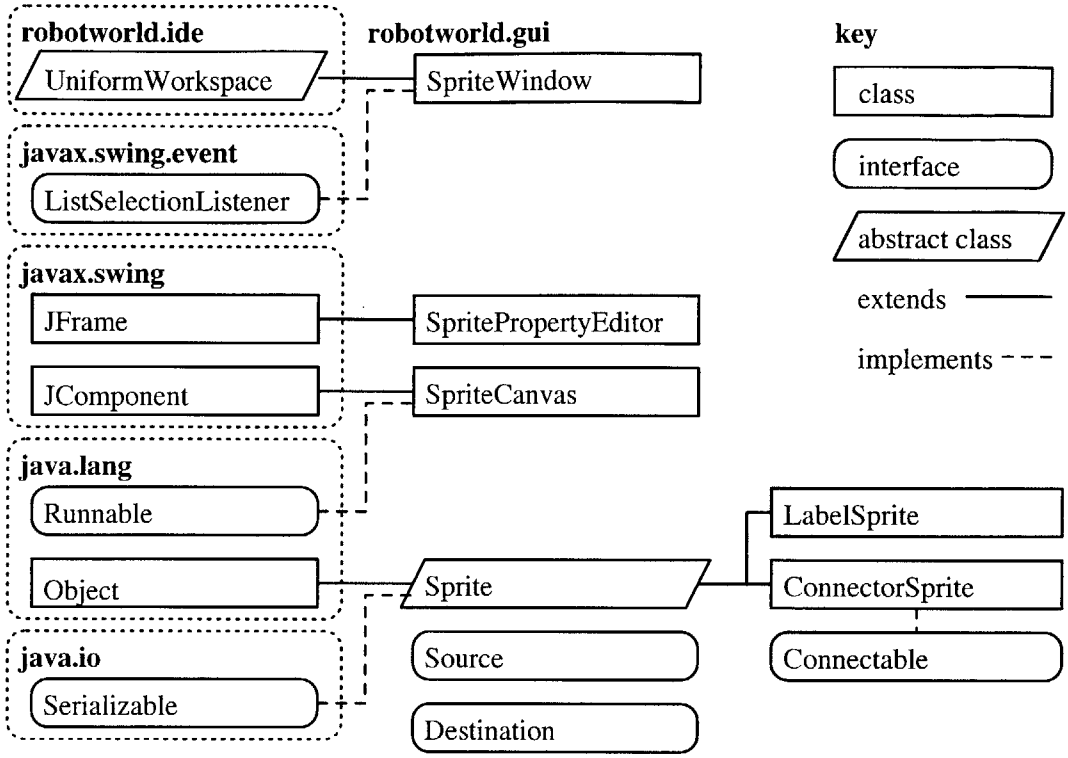


Figure 6-1: Class diagram of package robotworld.gui

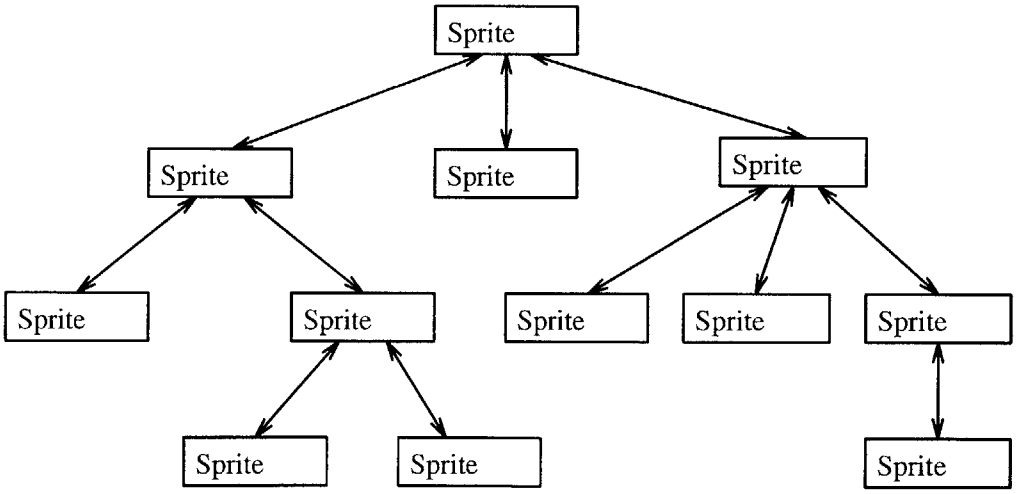


Figure 6-2: An example of a Sprite tree. Each box represents an instance of a Sprite, each of which may have a variable number of children, but at most one parent. This allows any Sprite to traverse up the tree to the root sprite, or down the tree to the leaf sprites.

maintains a list of children as well as a reference to its parent. This forms a non-uniform tree structure with bi-directional links, referred to as a sprite tree. See Figure 6-2 for an example. The root of the tree has a null parent, as one might expect. Sprites also maintain an extra field (a "link" sprite) to assist in setting up other data structures (e.g. robotworld.ctl.Pin). Sprite also maintains a number of graphical properties such as a String label, a bounding Rectangle, a boolean to record whether it is selected, an integer "layer" number which can help distinguish which of two overlapping sprites is on top (the higher number wins). There are several location manipulation methods, as well as a Point field to maintain a relative offset from the link sprite if the link sprite is non-null, and a field to record the Component it's being drawn in, mostly as a convenience for specialized listeners. The interesting methods (besides setters and getters for the above fields) include dispose(), which is an opportunity to clean up before being deleted, edit(), which is a command to bring up the appropriate window to edit this sprite's contents, paint(), which tells this sprite to draw itself, paintChildren(), which tells this sprite to draw its children, find(), which indicates whether a mouse click hit this sprite, and findChildAt(), which indicates whether a mouse click hit one of this sprite's children.

SpriteCanvas, a self-animating component, is the brains of the GUI. It provides enough listeners to permit the basic mouse operations on sprites (ADD, MOVE, OPEN). SpriteWindow just maintains a list of classes which can be instantiated on the SpriteCanvas and tells the SpriteCanvas when a new class is selected. It also notifies the SpriteCanvas when the user wants to edit the sprite properties, or delete the sprite. Recall from Chapter 5 that each Workspace (like SpriteWindow) is custom designed to handle a particular class of Sprite. Note that double clicking on a Sprite not only causes a SpriteWindow to spring into being, it also calls that SpriteWindow's configureToEdit() method, which in turn calls SpriteCanvas.setRootSprite() to inform the SpriteCanvas whose paintChildren() method to call every paint cycle. SpriteWindow and SpriteCanvas supply only the basics—no sprite-to-sprite connectivity. This is supplied by subclasses which check to see if Sprites implement any of the marker in-

terfaces: Source, Destination, or Connectable—and take the appropriate actions and state changes to allow this to happen. These are discussed further in Chapter 9. Besides two basic Sprite classes, one which draws a boxed label and another that draws a directional arrow, there is only one more class of interest: SpritePropertyEditor. It is a dialog that does what its name implies. The diagram indicates its eventual dependence on Swing; currently it extends java.awt.Frame. See Chapter 11, Future Work, for details.

6.3 Current Implementation

All visible elements appearing in the SpriteWindow lists should be descendants of the class Sprite. SpriteCanvas and its descendants attempt to maintain a consistent user interface: sprites may be selected/deselected with a single click, opened with a double click, dragged about the screen, and deleted with the DELETE key. Additionally, clicking in open space in a SpriteCanvas will instantiate the most recently selected bean from the BeanBag. The SpriteCanvas double buffers its display, so that sprite animation is not plagued by the distraction of flicker.

Currently, a new canvas and window are required for each layer. This is partly because of special modes (states) that the canvases need to provide for the connectability interface to work. In a superior design, there would be a generic canvas or window and external classes would tell it how to operate, using a model-view separation. See Chapter 11 for more details on this.

Chapter 7

The robotworld.ctl Package

Building on the infrastructure of Chapters 4–6, we now need to add data transfer abilities between sprites on the blackboard. Previously, one could connect sprites, but the connections were inert. Now we design a set of self-animating classes that, all together, result in local or remote data flow between sprites.

7.1 Background: Client-Server Models

You are probably familiar with the concepts of server push, and client pull. In each case, there are two entities involved in an ongoing set of interactions. For server push, let us define the two entities as the server-push server, who does the pushing, and the server-push client, who is passive. A magazine subscription is a good example of server push. The publisher does all the work. Contrast this with client pull. We have

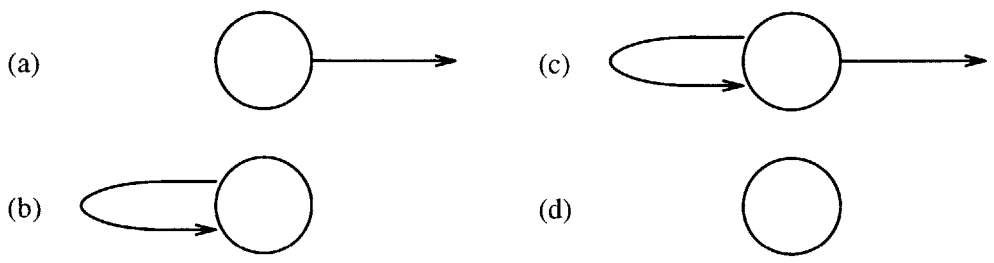


Figure 7-1: Client-server elements: (a) server push server, (b) client pull client, (c) active constraint, and (d) passive repository.

the client-pull server, who is passive, and the client-pull client, who does the pulling. Cats, being object-oriented creatures [45], are a good example of client-pull. If they want attention, *they will let you know.* [1] There are also two other entities of interest. One we shall call the active constraint. It combines the pulling aspect of client-pull with the pushing aspect of server-push. The result is something like a tornado. It rips up houses and trees and spits them out somewhere else. The last entity is the passive repository. For example, a blackboard. The only way to get chalk onto it, or to remove chalk from it, is to do it yourself. Now that you've been introduced to the whole cast of characters, let's simplify the script a little.

There are four elemental building blocks for client-server communication. See Figure 7-1. They include the server push server, the client pull client, the active constraint, and the passive repository.[2] Server push servers can be chained, as can client pull clients, but the two cannot be directly connected: they are duals. Their active ends can be bridged by a passive repository, and their passive ends can be bridged by an active constraint, which continuously polls its source and updates its destination. Note that passive repositories and active constraints are also duals. The two can be chained in an alternate fashion.

7.2 Core Interfaces

RobotWorld uses all four elements. Figure 7-2 explains how they all interconnect for remote data transfer. The server side of the diagram will be explained in Chapter 8; for now, suffice to say that all elements on the client side have an analog on the server side. In the diagram, all elements (except for the blackboard) are considered active. The arrows demonstrate with whom lies responsibility for data transfer, but

[1] Similarly, one would be tempted to suggest that Schrödinger's cat is event-driven [35]. The cat's fate is only decided when you look in the box.

[2] I don't consider the server push client and client pull server to be elemental, since their role can be played by some of the other four. In other words, they are relative definitions.

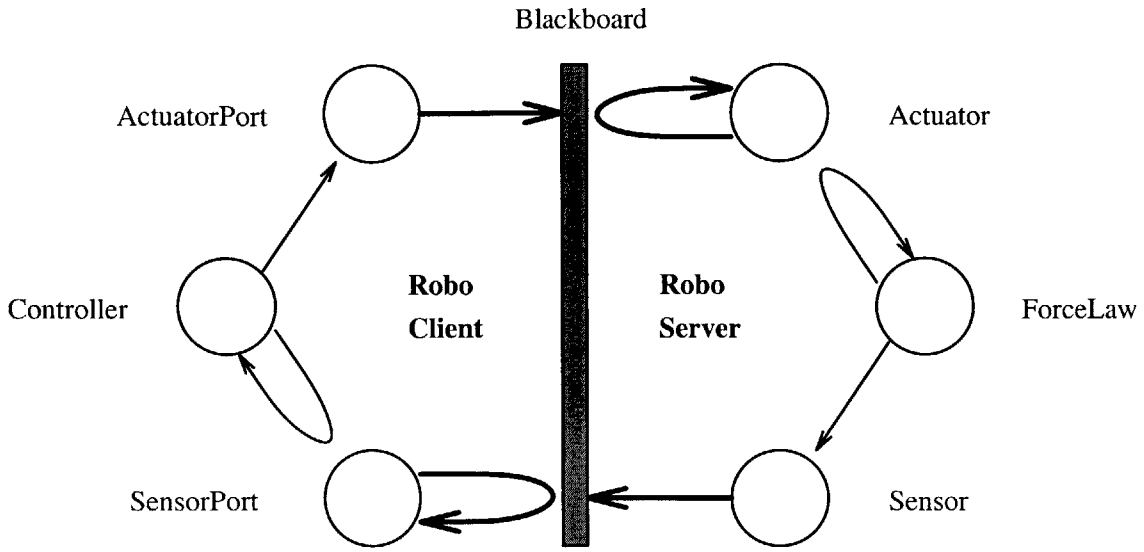


Figure 7-2: Entities involved in remote data transfer.

note that bold arrows indicate automatic transfer via animate while-true loops. The remaining responsibilities are the domain of either student behavioral programming (client side) or the real or simulated robot (server side). For the simulated robot, the sensor and actuator morphology as well as the force laws in play are accessible via the RobotWorld GUI (more about this in Chapter 8). In a real robot, real-world dynamics and morphology will complete the cycle of information from actuator to sensor.

Local data transfer is much simpler. Figure 7-3 demonstrates, in slightly more detail, how local data transfer occurs. This should illuminate the interrelationship between Connections, SignalSources, SignalDestinations, and Controllers. If Controllers are active constraints, and Connections are passive repositories, then SignalSources implement server-push, and SignalDestinations implement client-pull. A Controller might be implemented as a single animate object, or perhaps as a CompositeController which contains an ensemble of other objects. Viewed from the outside, however, it pulls data from its inbound connections and pushes data to its outbound connections. A Connection is quite simply a one-way conduit from a SignalSource to a SignalDestination. SignalSources can be thought of as the embedded output-ports of a controller: they generate data to be sent to other parts of the system. SignalDes-

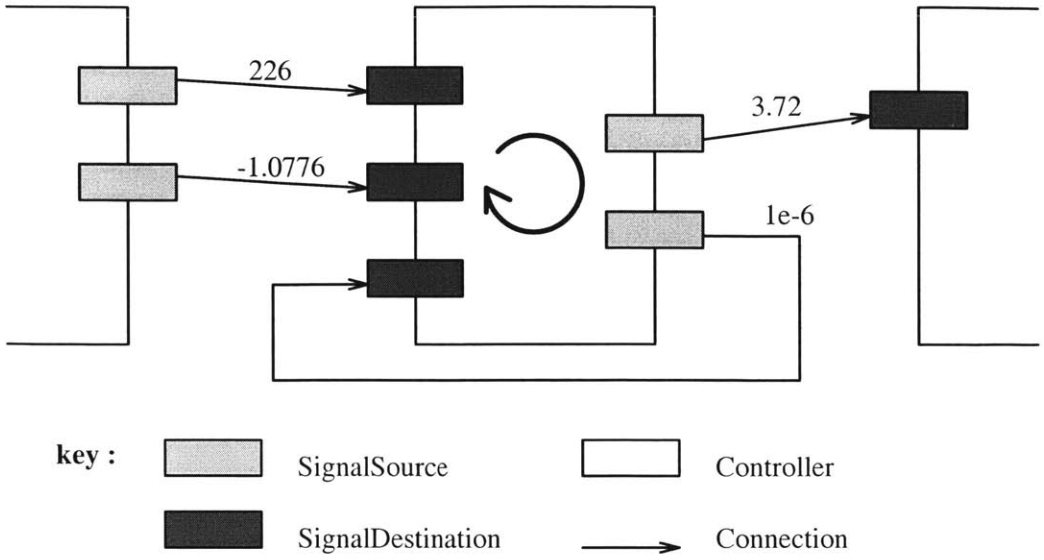


Figure 7-3: Entities involved in local data transfer. The student's domain is the self-animating class indicated by the bold arrow.

tinations, then, are the embedded input-ports of a controller: they collect data for the controller's use. Operationally, SignalSources may have many SignalDestinations, but a SignalDestination may only have one SignalSource. This is meant to keep the design of synchronization coping strategies inside the controller, where animacy is understood to reside.

Most likely, students will be designing a self-animating subclass of Controller. Conveniently, Controllers maintain lists of SignalSources and SignalDestinations. Keep in mind that these class names reflect external, as opposed to internal, connectivity. Thus a student while-true loop will most likely request a particular SignalDestination, and obtain its current value using getSample(). They will then request a particular SignalSource, and generate its next value using setSample(). Note that SignalSource and SignalDestination do not maintain state, only Connections (a list of Connections for SignalSource, and a single Connection for SignalDestination). Connection maintains the signal state. Thus when a student calls a SignalSource's setSample(), it iterates through its list, calling each Connection's setSample() in turn. Similarly, when a student calls a SignalDestination's getSample(), it calls its

Connection's getSample() and returns the result. Since the intermediary Connection decouples the source and destination while-true loops, a variety of behaviors is possible. A Connection can resample or drop incoming values, depending on the speed at which signals are generated (by a SignalSource) versus the speed at which they are sampled (by a SignalDestination). If generation is faster, some values will be dropped. If sampling is faster, some values will appear multiple times.

There are other self-animating entities on the client side, most notably the SensorPort and ActuatorPort. The SensorPort is a subclass of a SignalSource whose job it is to continuously poll the blackboard for data, and push this data into the behavior system. The ActuatorPort performs the dual duty of continuously pulling data from the behavior system and writing it to the blackboard. Thus the details of the morphology and the physical laws which govern it are completely abstracted away from this layer.

Figure 7-4 is a class diagram of all of the client-side classes described so far. There is a ladder of dual relationships embedded in this diagram, which have been blown up in a separate diagram, Figure 7-5. The only thing that might surprise the reader is that a class can be simultaneously a local source and a remote destination, or vice versa. Sounds weird, but remember that remote relationships are just the duals of internal relationships, as we saw in the naming anomalies of the Controller.

An example will help clarify this local/remote duality. Let us take the perspective of a student writing an animate while-true loop inside the Controller of Figure 7-3. How do values propagate through the system? Assume that the student invokes a method on the upper SignalSource, with an argument of 3.72. The SignalSource writes this value to its Connection, modifying its state. Let us further assume that the other end of this Connection is an ActuatorPort. The ActuatorPort is self-animating: it continuously polls for the latest value from the Connector. Now what? The only way it can get this value to the RobotWorld server (i.e. the real or virtual robot) is

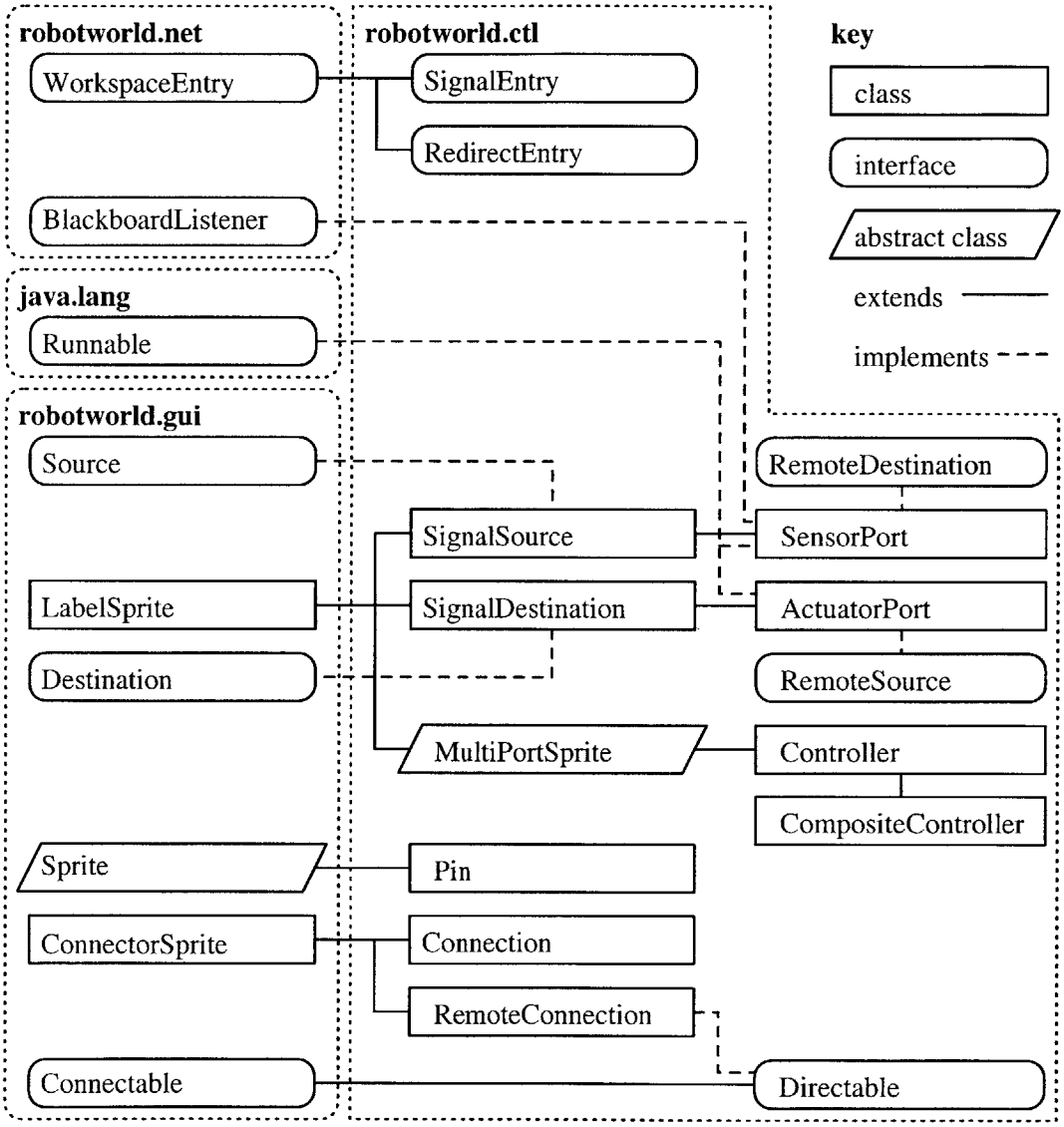


Figure 7-4: Class diagram of package robotworld.ctl

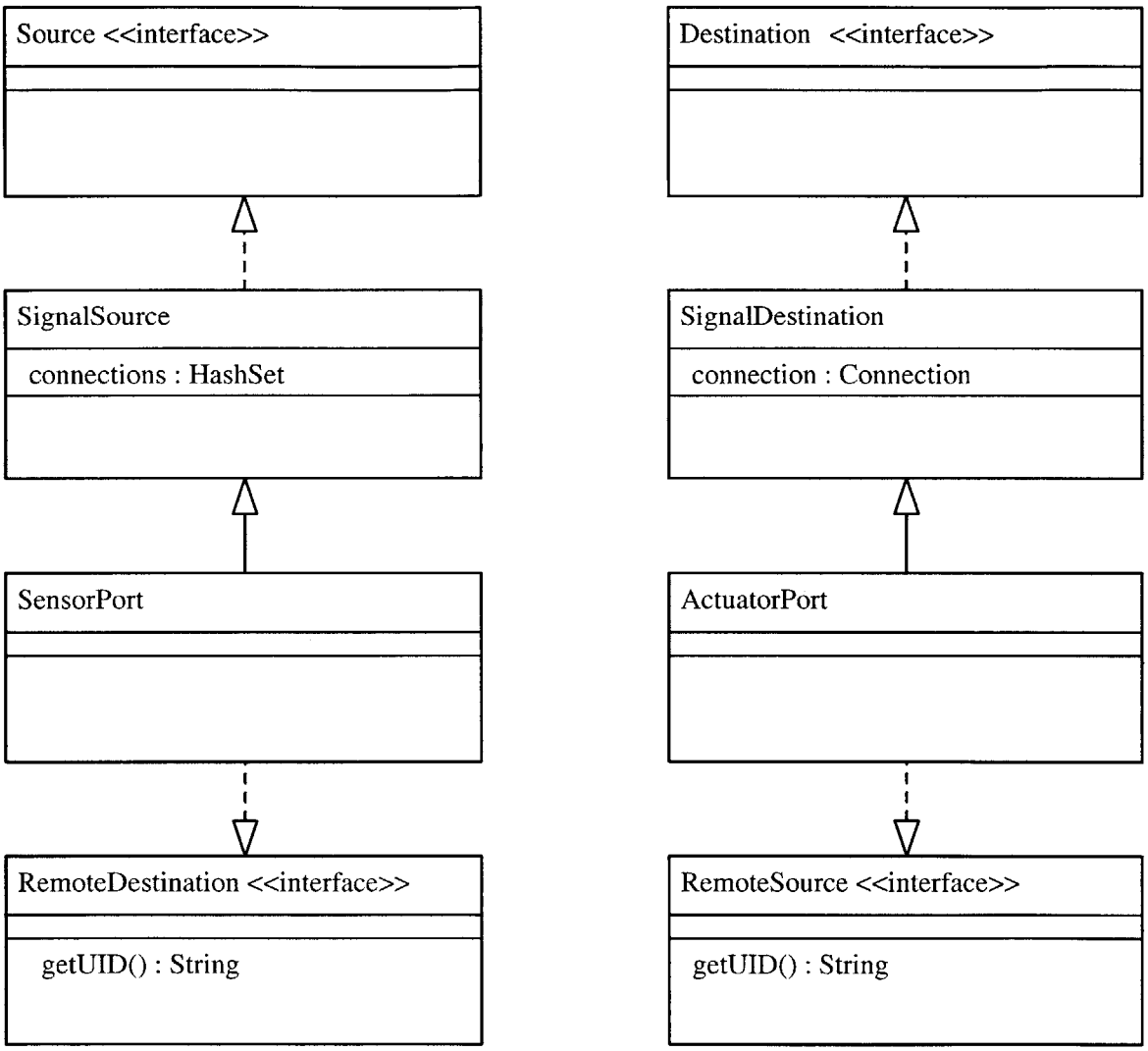


Figure 7-5: The dual relationship between sources and destinations (UML).

to have it write the value to a remote "connector" : the blackboard. The illusion is maintained, since many remote "destinations" can receive information from the same "source" using the blackboard, just like a local connection. Thus a local destination and a remote source are identical.

There are a few other classes that deserve explanation. The MultiPortSprite class provides special Sprite functionality that causes visual "pins" to be displayed as Sources and Destinations are added to it via addChild(). Right now, MultiPortSprite is responsible for creating and maintaining Pins, as well as deciding whether mouse clicks land on itself or one of its Pins. The Pin class is very simple, and only maintains a reference to its link sprite (the Source or Destination), knows how to paint() itself, and decide whether to reveal its link sprite on a find(). Finally, we have the interfaces SignalEntry, RedirectEntry, and Directable. SignalEntry is just a WorkspaceEntry with an additional (conceptual) field for a signal value. Directable is just the remote refinement of a Connectable. It too is a marker interface that assists the canvas classes in setting up special user interface behavior. RedirectEntry, however, needs further explanation.

RedirectEntry is a special WorkspaceEntry which provides a message redirect protocol. Whenever a RemoteConnection is used to connect a RemoteSource and a RemoteDestination, the RemoteConnection writes a RedirectEntry to the blackboard with the RemoteSource.getUID() string in the subject field and the RemoteDestination.getUID() string in the client field. The domain is RedirectEntry.REDIRECT. At construction, a RemoteDestination must listen() for RedirectEntries with its UID in the client field. When the notify() arrives, it reads the subject field to find out which RemoteSource to poll from until the next REDIRECT. This allows a user to connect pieces of a robot "brain" to a robot "body" using the GUI, when the underlying software is resident on multiple machines and otherwise would not have known that the other pieces exist.

7.3 Current Implementation

Each blackboard implementation requires a corresponding RedirectEntry and SignalEntry implementation. See Figures 4-2 and 4-3. These "implementations" do nothing at all special except inherit from the appropriate superclass. It is my hope that a better design might someday come along to eliminate all these crazy Entry classes. The current design requires one interface for each conceptual entry type, and N implementations, one for each Blackboard implementation. See the Chapter 11 for some ideas on how to solve this.

Chapter 8

The robotworld.sim Package

Despite the focus on simulation in RobotWorld, this is by no means the only way it should be used. Here's one reason you may want to tack to a different wind: perhaps you want students to design software to control their robots, and test it out in a simulated environment. Then, when they are comfortable that their designs will succeed, just change the RobotWorld configuration to support real-world robots (assuming you have the facilities and have written drivers). Some designs will translate surprisingly well, and some of them will not. This would be a great two-pronged approach to facilitate hands-on learning of simulation science (e.g. artificial life) and situated robotics that truly underscores the essential differences. Alternately, RobotWorld could be used completely with real-world robots as an interactive development environment.

Ultimately, robot support requires the following: writing a "driver" for your robot hardware that communicates with RobotWorld via the blackboard. All that RobotWorld cares about is that it has named sources and destinations which adhere to the basic protocols. The driver software could simply mirror the RobotWorld.sim.Sensor and RobotWorld.sim.Actuator code, and use the Java Native Interface (JNI) to read sensor values and write actuator commands, either directly with the robot board via memory-mapped I/O, or indirectly via serial port and the Java Serial Communication API (COMM). It all depends on how much interface software is provided with the

robot board; the MIT Handyboard [53] and the Khepera controller [57] are probably good choices from this standpoint. Then, to integrate the driver into RobotWorld, simply write a properties file which instantiates your driver instead of the simulator at server startup. RobotWorld clients won't be able to tell the difference: they keep sending actuator commands and reading the sensor data, only this time it will be live. They will however, have to bind to the correct name to have anything happen. A consistent naming scheme between virtual and physical objects will cause the least headache: assuming that a team of students uses a particular robot, they should name the virtual robot, and all of its components, identically.

Now, students will benefit from all the features of the environment: interactive code and behavior editing, and all of the potential of a blackboard as a communication medium. For example, a virtual oscilloscope can plot a time-series or a phase diagram simultaneously on the fly; this is because a producer can have many consumers in the basic RobotWorld protocol. Also, the entire session can be recorded for later playback; this is because of the persistence capabilities of blackboard systems. And because many systems can be networked together, one could devise experiments in distributed computing. By implementing a RemoteDestination that performs a majority vote on all incoming signals, one could bind many software controllers to one robot in a fault-tolerant fashion, and see how the robot performs as it loses each controller, one by one. Blackboard systems make possible many kinds of coordinated activities that a creative instructor could take advantage of. That being said, I turn back to the pure Java simulator which makes this toolkit portable and cost-effective for those instructors who don't have access to a robot laboratory.

8.1 Background: Physically-based modeling

At its core, the physics of RobotWorld consists of a 2D particle simulation, where the particles are allowed to have a finite extent and orientation. On top of this, a

sensing and actuation framework allows these particles to control their movement and orientation by manipulating internal forces and torques.

8.1.1 2D Particle Systems

Using the physically-based modeling techniques of Witkin, Baraff, and Kass [89], the set of particles are treated as a dynamical system and manipulated by forward integration of a ordinary differential equation (ODE) in state-variable representation:

$$\dot{\vec{x}} = \vec{f}(\vec{x}, t) \tag{8.1}$$

where $\vec{x}$ is the current state of the system, and t is the time elapsed since the beginning of the simulation. This allows us to express the first derivative of the particles' state as a function $\vec{f}$ of the current state, and time.

What state do the particles need to maintain, then? In the current RobotWorld system, particles are restricted to the plane, so the state variables should be (x, y, θ). The orientation θ is defined to be zero when pointing to the right side of the screen and increasing as the particle turns counter-clockwise. The position (x, y) is defined to be identical to screen coordinates, with $(0, 0)$ defined to be the upper left-hand corner, with x increasing to the right and y increasing downwards.

For the purposes of numerical integration, we wish to find a first-order expression. However, Newtonian mechanics dictates a second-order differential equation. For example, Newton's law tells us that $F = Ma$ for a one-dimensional system. Expressed as an ODE, we have:

$$\ddot{x} = F/M \tag{8.2}$$

which is second-order. This can be transformed into state-variable representation by doubling the dimensionality of the system:

$$\dot{x_1} = x_2 \tag{8.3}$$

$$\dot{x_2} = F/M \tag{8.4}$$

This is truly a second-order system (two initial conditions are required, $x(0)$ and $v(0)$); however, we have expressed it as a first-order system. There is only one problem. We will have to estimate the velocity using some numerical integration scheme, which can be subject to instability. Witkin, Baraff, and Kass recommend the use of conserved quantities for the additional state variables, which allow one to measure the amount of drift due to the integration scheme.

Now we can determine the full state-variable representation for the RobotWorld particle system. For a 2D particle of mass M, the relationship between the position $\vec{x}$ and the applied force $\vec{F}$ and linear momentum $\vec{P}$ is:

$$\vec{P} = M\vec{v} = M\dot{\vec{x}} \tag{8.5}$$

$$\vec{F} = M\vec{a} = M\ddot{\vec{x}} = \dot{\vec{P}} \tag{8.6}$$

For a particle of rotational inertia I, the relationship between orientation θ and the applied torque τ and angular momentum L is:

$$L = I\omega = I\dot{\theta} \tag{8.7}$$

$$\tau = I\alpha = I\ddot{\theta} = \dot{L} \tag{8.8}$$

Putting this all together, we have:

$$\begin{bmatrix} \dot{\vec{x}} \\ \dot{\theta} \\ \dot{\vec{P}} \\ \dot{L} \end{bmatrix} = \begin{bmatrix} \vec{v} \\ \omega \\ \vec{F} \\ \tau \end{bmatrix} \tag{8.9}$$

where $\vec{v} = \vec{P}/M$ and $\omega = L/I$. This results in a particle with a 6-dimensional state

vector. Additionally, given the density and radius of the particle, we can determine the mass M and inertia I of a disk in the plane:

$$M = \pi R^2 \rho \tag{8.10}$$

$$I = MR^2/2 \tag{8.11}$$

Note that this formulation was derived from the more complex rigid body model in Witkin, Baraff, and Kass, in which a particle had three positional variables as well as three orientational variables (six degrees of freedom). The variables used in RobotWorld were obtained by mapping the vectors $\vec{x}, \vec{v}, \vec{P}, \vec{F} \in \Re^3$ to $\Re^2$, and mapping the vectors $\vec{L}, \vec{\tau} \in \Re^3$ to $\Re$. Additionally, in order to remove the need for matrix algebra, the 3x3 rotation and inertia tensor matrices R, I were replaced by the scalars θ, I. In RobotWorld, the 2x2 R matrix is constructed from the state variable θ instead, in the following manner:

$$R = \begin{bmatrix} \vec{R_x} & \vec{R_y} \end{bmatrix} = \begin{bmatrix} \cos\theta & -sin\theta \\ -sin\theta & -cos\theta \end{bmatrix} \tag{8.12}$$

This rotation matrix may seem unfamiliar to some readers. This is because y increases going down the screen, contrary to the usual definition of Cartesian coordinates. This matrix maintains the counter-clockwise rotation property despite a left-handed coordinate system. Extending to a 3D coordinate system:

$$\vec{R_x} \times \vec{R_y} = \begin{vmatrix} \hat{x} & \hat{y} & \hat{z} \\ \cos\theta & -sin\theta & 0 \\ -sin\theta & -cos\theta & 0 \end{vmatrix} = \hat{z} \cdot (-\cos^2\theta - \sin^2\theta) = -\hat{z} \tag{8.13}$$

Since we have a left-handed coordinate system, $\hat{z} = \hat{x} \times \hat{y}$ points into the computer screen, which corresponds to clockwise rotation. The extra negative sign in the above expression gives us the counter-clockwise rotation property.

With that in mind, the full-blown state equation for a 6DOF rigid body is simply:

$$\begin{bmatrix} \dot{\vec{x}} \\ \dot{R} \\ \dot{\vec{P}} \\ \dot{L} \end{bmatrix} = \begin{bmatrix} \vec{v} \\ \vec{\omega}^* R \\ \vec{F} \\ \vec{\tau} \end{bmatrix} \tag{8.14}$$

where $\vec{v} = \vec{P}/M$ as before, but $\vec{\omega} = I^{-1}\vec{L}$. This result is a rigid body with an 18-dimensional state vector. The operator $(*)$ above indicates to replace the vector $\vec{\omega}$ with a matrix $\vec{\omega}^*$ such that for any vector $\vec{r}$, $\vec{\omega}^*\vec{r} = \vec{\omega} \times \vec{r}$. The matrix is simply:

$$\vec{\omega}^* = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} \tag{8.15}$$

Note that the inertial tensor matrix is $I = RI_{body}R^T$, where R is changing in time, and I_{body} is a constant matrix defined by the mass distribution of a rigid body. However, so long as the moments of inertia are all identical and the products of inertia are all zero, the matrix I_{body} is simply the identity matrix, multiplied by a scalar η. As a result, the time-changing inertia tensor matrix becomes $I = \eta \cdot R\mathbf{1}R^T = \eta \cdot RR^{-1} = \eta\mathbf{1}$. At this point, only the scalar η is relevant. Since RobotWorld only uses disks of uniform density, there is no need for a 2x2 inertia tensor matrix, and an explicit formula can be used to determine η.

8.1.2 Braitenburg Vehicle Model

Our default 2D vehicle derives from the quirky designs of Valentino Braitenburg. In short, this means the vehicle is a "tank" with two differential drive motors, each of which can be run independently, either forwards or in reverse. Recall that the body is modeled simply as a disk of uniform density. In the simplest case, sensors placed around the vehicle's circumference can be wired to the motors such that the motor command value is monotonically increasing or decreasing with signal intensity. As a result, the vehicle moves forward when the two command values are equal, and turns

when they are unequal. The forward and angular acceleration can be determined by assuming a different rigid body model, with two masses at the centers of the wheels. We know that the sum of the forces on the center of mass determines its forward acceleration:

$$M * a_0 = \sum_j F_j = \sum_j M_j * a_j \tag{8.16}$$

In our particular case, $M_l = M_r = M/2$, and the acceleration of the center of each wheel is simply the acceleration of the contact point with the ground, which is the angular upright acceleration of the wheel times R, the radius of the wheels. Thus:

$$M * a_0 = \sum_j F_j = M_l * a_l + M_r * a_r = R * M/2 * (a_l + a_r) \tag{8.17}$$

Dividing out by M gives us:

$$a_0 = R/2 * (a_l + a_r) \tag{8.18}$$

Next, we determine the angular acceleration by attaching an object frame that moves with the vehicle but has its axes aligned with that of the external world frame through which the vehicle travels. In this object frame, the axle will sweep out a circle of radius of $D/2$, where D is the distance between the centers of the wheels. If the angular acceleration of the left wheel is zero, then the angular acceleration of the axle would be equal to the instantaneous acceleration of the center of the right wheel, divided by the radius $D/2$:

$$\alpha_0 = \frac{\alpha_r * R}{D/2} \tag{8.19}$$

If the angular acceleration of the left wheel is not zero, then it will act to oppose the rotation of the axle:

$$\alpha_0 = \frac{(\alpha_r - \alpha_l) * R}{D/2} \tag{8.20}$$

Using the linear acceleration a_0 and angular acceleration α_0, we can determine the self-generated forces and torques on the vehicle. The only thing missing from the picture is the force-torque coupling: otherwise the vehicle will spin while moving in a straight line, just like in free space! The solution is to impose a centripetal acceleration force law; in reality this is imposed by the friction between the wheels and the driving surface. In centripetal acceleration, $v = r * \omega$, so that:

$$\alpha_c = \omega^2 * r = v^2/r = \omega * v \tag{8.21}$$

This formulation is handy, since it only depends on the two state variables ω and v, making the cross-coupling explicit. Otherwise, we would have to calculate the radius of curvature r from the motor rates ω_r and ω_l, forcing us to model the motor state as well as the rigid body state.

8.1.3 Vehicular Feedback Control

Our vehicles will want to alter actuator command values using sensor feedback in order to interact with the world. This requires some simple state-space control theory. Our particle system can be expressed in state-variable representation as follows:

$$\dot{\vec{x}} = \mathbf{F}\vec{x} + \mathbf{G}\vec{u} \tag{8.22}$$

$$y = \mathbf{H}\vec{x} + \mathbf{J}\vec{u} \tag{8.23}$$

where:

$\vec{x}$ is the state vector,

$\vec{u}$ is the (time-varying) control input,

$\vec{y}$ is the system output,

$\mathbf{F}$ is a matrix of system coefficients,

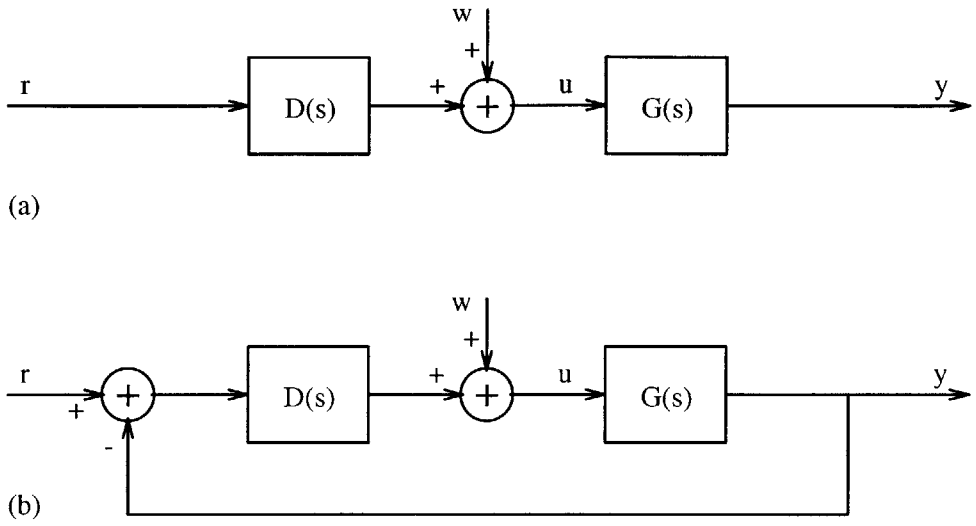


Figure 8-1: A block diagram of (a) an open-loop controller and (b) a closed-loop controller.

G is a matrix of control input coefficients,

H is a matrix of output coefficients, and

J is a matrix of direct transmission coefficients.

Our high-level goal is to be able to specify a reference input r, say, a constant linear velocity v and a constant angular velocity ω, and have the system respond with a force or torque u that will bring the vehicle state towards the desired value. In other words, we need a control law which gives u in terms of r and $\vec{x}$, and possibly, a system disturbance w.

First of all, let us design a simple proportional gain controller using standard block diagrams. Each component in the diagram represents a transfer function in the frequency domain:

$\mathcal{G}(s)$ is the plant transfer function, and

$\mathcal{D}(s)$ is the controller transfer function

The figure shows both open-loop and closed-loop controllers. We will consider each in turn. Before we proceed, however, we need the transfer function $\mathcal{G}(s)$ of the

plant (vehicle). This can be calculated directly from the state-space representation [29]:

$$\mathcal{G}(s) = \frac{\det \begin{bmatrix} s\mathbf{I} - \mathbf{F} & -\mathbf{G} \\ \mathbf{H} & J \end{bmatrix}}{\det [s\mathbf{I} - \mathbf{F}]} = \frac{\det \begin{bmatrix} s & -1/M & 0 \\ 0 & s & -M \\ 0 & 1/M & 0 \end{bmatrix}}{\det \begin{bmatrix} s & -1/M \\ 0 & s \end{bmatrix}} = \frac{s}{s^2} = \frac{1}{s} \tag{8.24}$$

In essence, our system is a simple integrator. This makes sense, because our control signal u entering the plant is an acceleration, and the observed output y coming out of the plant is a velocity. In the open loop case with the proportional gain $\mathcal{D}(s) = A$, the overall transfer function of the system is:

$$\mathcal{H}(s) = \frac{\mathcal{Y}(s)}{\mathcal{R}(s)} = \mathcal{D}(s)\mathcal{G}(s) = A/s \tag{8.25}$$

The control law is then:

$$u = Ar + w \tag{8.26}$$

where w is a system disturbance (e.g. from collisions).

The open-loop system has a pole at the origin. This means that in the presence of any disturbances, the plant will integrate these errors and the "cruising velocity" will increase or decrease, even if the reference input is zero. It also imposes an particular interpretation on the reference input: since an impulse integrates to a step, and a step integrates to a ramp, students desiring a constant forward velocity must design their vehicles so that the actuator ports output zero, unless a velocity change is desired. A vehicle whose actuator ports output a nonzero constant will result in a ramp in velocity: probably not what was intended.

This system would be sufficient for the purposes of RobotWorld, but a simple alteration will permit students to design vehicles in a more intuitive sense. By closing

the loop, vehicles whose actuator ports output a nonzero constant will approach and maintain a cruising velocity. In addition, the gain can be increased to compensate for any possible disturbances in the environment, without any loss of stability. The transfer function for a closed-loop system is:

$$\mathcal{H}(s) = \frac{\mathcal{Y}(s)}{\mathcal{R}(s)} = \frac{\mathcal{D}(s)\mathcal{G}(s)}{1 + \mathcal{D}(s)\mathcal{G}(s)} = \frac{A/s}{1 + A/s} = \frac{A}{s + A} \tag{8.27}$$

The control law is then:

$$u = A(r - y) + w \tag{8.28}$$

where w is a system disturbance (e.g. from collisions).

The closed-loop system has a pole at $-A$. With a pole in the left hand plane, the system is stable, and will settle out to $y = r$. This means that the response to a step input is exponential decay from the old value to the new value. Also, by making A sufficiently large, any disturbances will only affect the output with scaling factor $1/A$. The one caveat of using large gain values is that the time step must be small enough that the target velocity is not overshot. This is the well-known stiffness problem in numerical integration [33], and is an artifact of the simulation resulting from coarse time steps in the presence of large derivatives. One solution is adaptive time steps, which is not recommended for the RobotWorld system due to the necessity of interactive playback. Instead, a tradeoff must be made between large gains and small time steps, keeping in mind that smaller time steps means a larger share of CPU time for the ongoing simulation thread. The problem can also be alleviated somewhat by clamping the control signal u to lie within a range of accelerations which are reasonable, considering the maximum distance (in number of pixels) a vehicle should move for a given frame rate in order to maintain a smooth animation effect. These numbers were determined empirically.

8.2 Core Interfaces

There are five basic subsystems that comprise the simulation package: the dynamic model, GUI, controls, collision detection, and radiation model. This section explains the responsibilities of and interfaces between each of these components. The five accompanying diagrams each provide a 'cross-section' of the simulator, highlighting only the features relevant to the discussion. The union of all of these diagrams would produce the full, albeit complex, picture.[1]

The dynamic model provides a framework for numerical integration in an object-oriented setting. As you can see from Figure 8-2, the ParticleSystem is the central class, maintaining a list of RigidBody instances and a list of ForceLaws for which it is responsible. Since it implements the DynamicalSystem interface, it can request a new time step from a DiffeqSolver, which will update the system state. The ForceLaws implicitly influence the model by altering the expression for the derivative at the time of update. This means that an EarthGravity instance could be added to the system, which would cause a constant to be added to the expression for acceleration for every particle in the simulation. It is the responsibility of the ForceLaw implementation to influence one, two, or all RigidBody instances as appropriate. ParticleSystem also maintains an optional CollisionDetector and RadiationModel (expressed in later diagrams).

The control aspect of the simulation is provided by the abstract classes Sensor and Actuator, which implement the interfaces RemoteSource and RemoteDestination, respectively. This means that an Actuator is responsible for collecting information from the appropriate ActuatorPorts residing on the network, while a Sensor is responsible for broadcasting information to the appropriate SensorPorts. Actuator also implements ForceLaw, so that command values influence the simulation. The resulting changes in system state can then be observed by Sensors which choose to interact with the collision detection system or the radiation model, described below. This completes the cycle of data flow (see diagram). Each RigidBody then maintains a list

[1] I drew it on paper. It wasn't pretty.

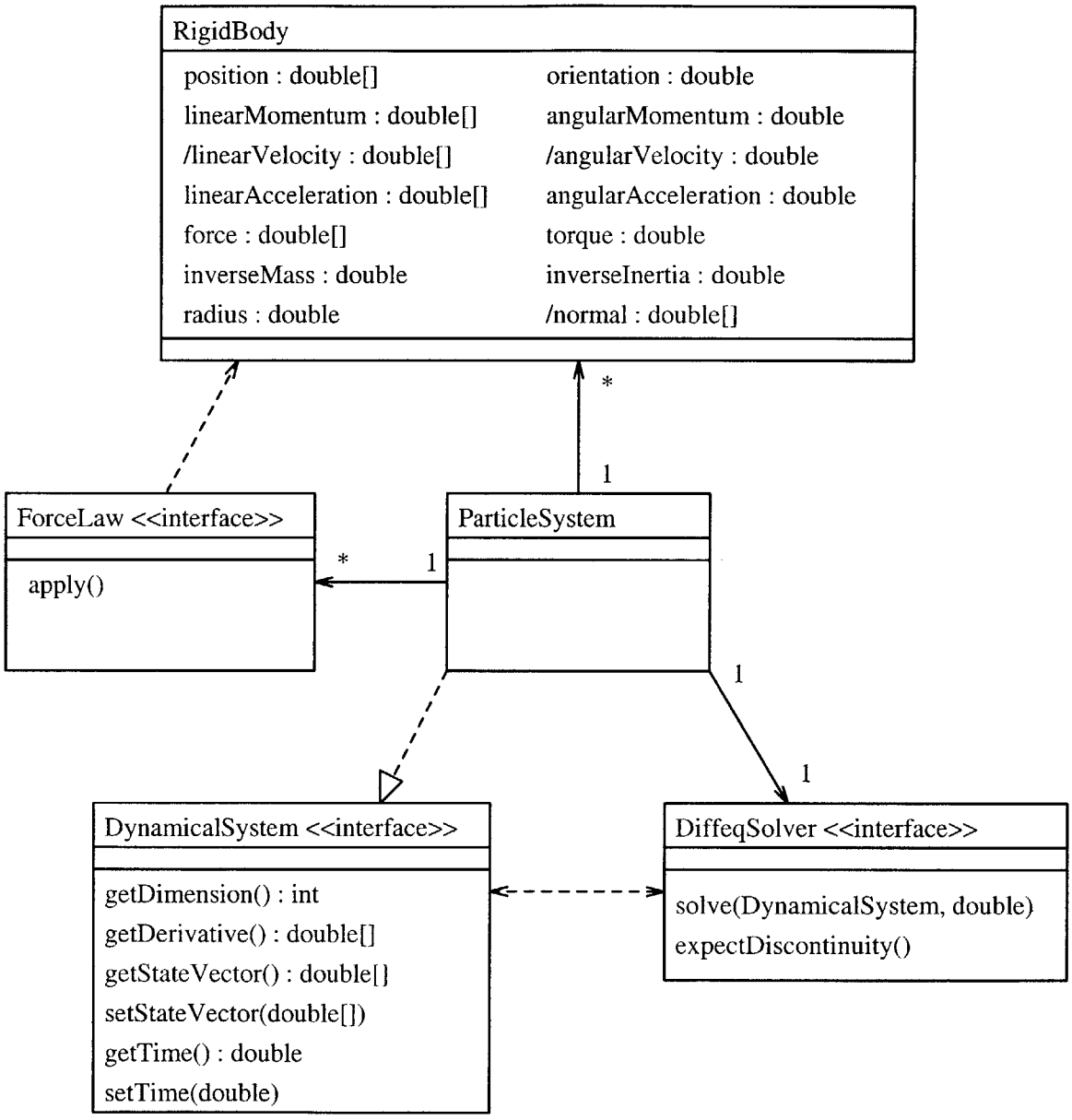


Figure 8-2: The dynamical core of the simulator (UML).

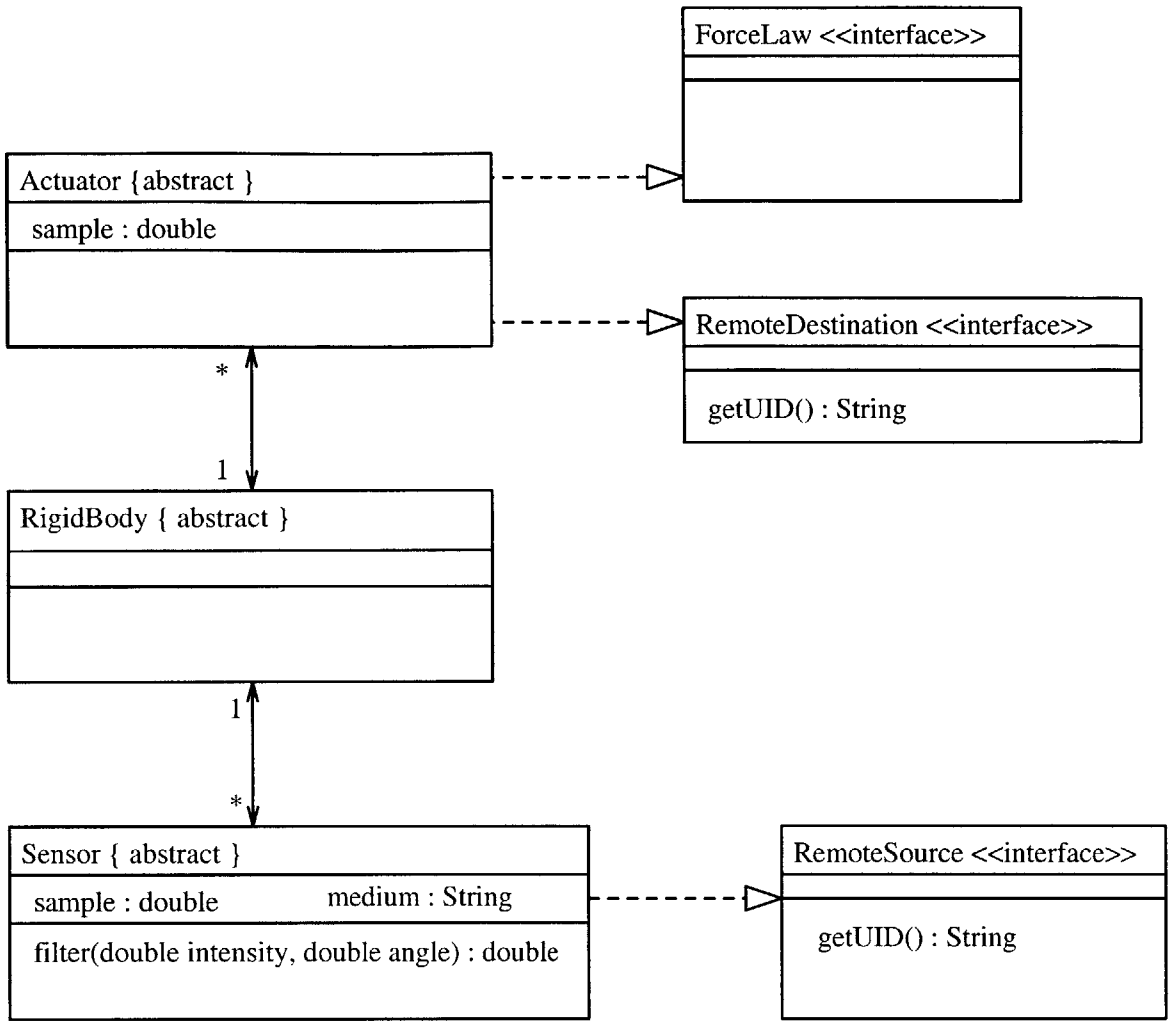


Figure 8-3: The control core of the simulator (UML).

of Sensors and Actuators which are embedded within in it and whose position and orientation depend on that of the RigidBody.

The GUI aspect of the simulation is accomplished by subclassing. RigidBody extends Sprite, while the Sensor and Actuator classes extend MultiPortSprite, which also derives from Sprite but provides specialized pin layout and labeling functionality. ParticleSystem extends LabelSprite, since it has no need of pins but does need a label. See the class diagram in Figure 8-6 for these relationships.

The collision detection system requires only two classes, CollisionDetector and ContactForce. The CollisionDetector maintains a list of RigidBodies which are capable of collision, as well as a list of ContactForces generated by the current state of the

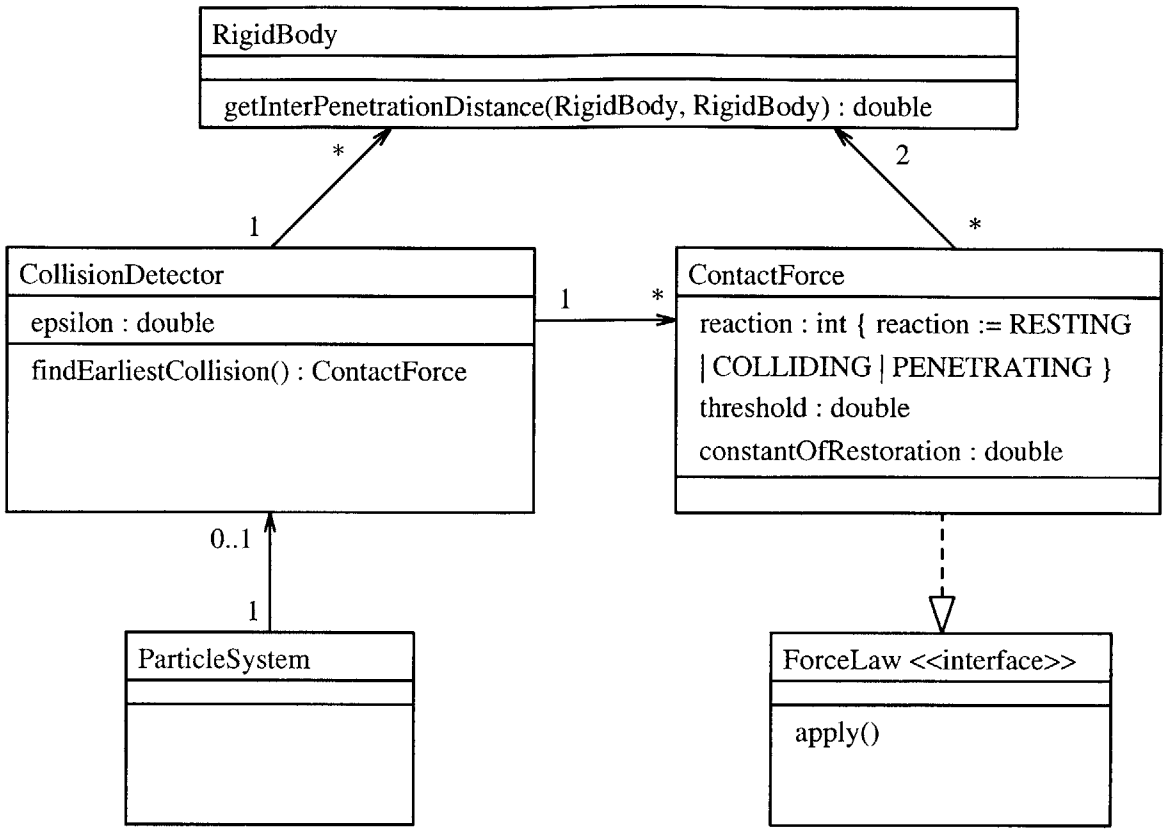


Figure 8-4: Collision detection and response (UML).

system. A ContactForce simply identifies two RigidBodies which are in contact and determines a collision reaction by implementing ForceLaw. ContactForce instances are very short-lived, depending of course on the nature of the interactions in the simulation.

The radiation model is similarly designed. It maintains a list of Emitter and Detector instances, and calculates the total intensity perceived at a Detector over all Emitters. Currently the radiation model is encapsulated within the ParticleSystem, but it should eventually be promoted to its own class. See Chapter 11 for a discussion of this future work. The Emitter interface depends on the Detector interface, but not vice versa. The Emitter does not reveal its position and orientation, allowing for effects which speed up simulation, such as light sources at infinity and ambient light levels.

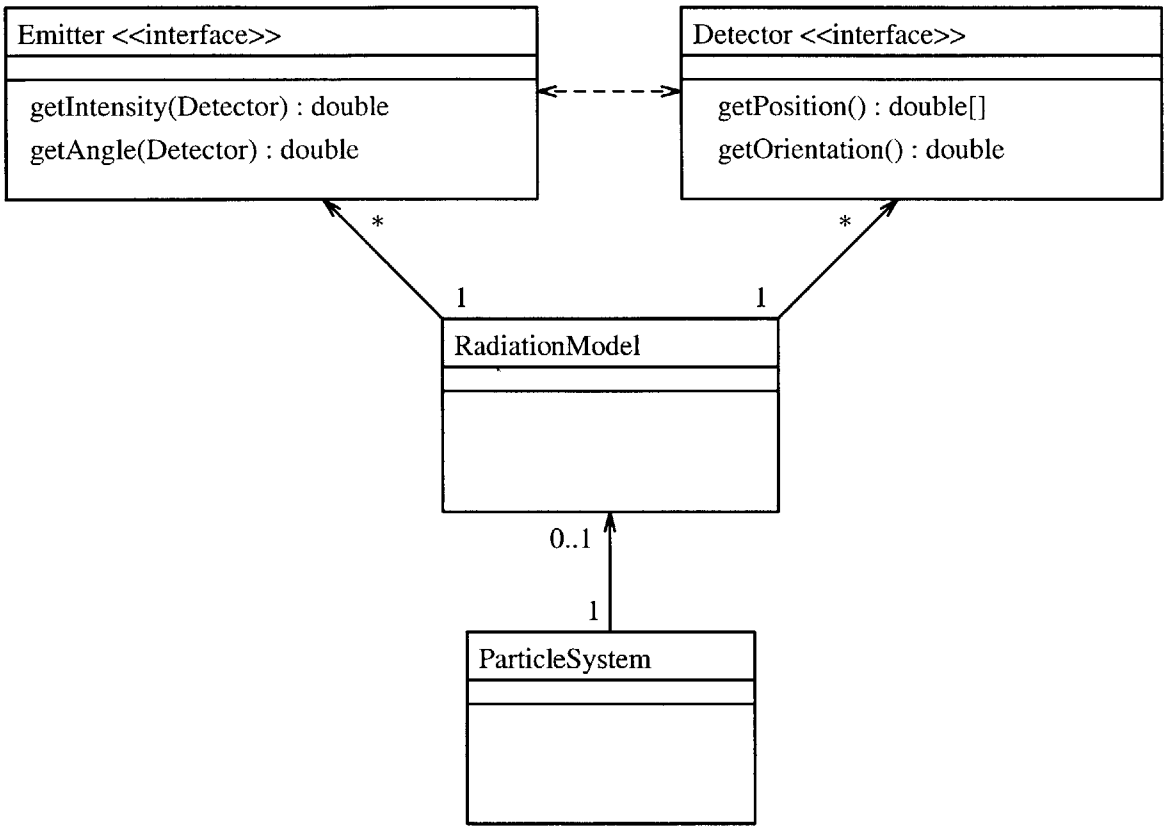


Figure 8-5: The radiation model (UML).

8.3 Current Implementation

Most of the functionality is built into the classes listed above. All that is needed is a few concrete implementations for some of the interfaces and the basic elements of the playing field: a few integration methods, body types, sensors, actuators, force laws, and specialized canvases and windows to paint the new layers. These implementations are listed in the rightmost column of Figure 8-6.

EulersMethod and RungeKutta provide two concrete implementations of DiffeqSolver. EulersMethod is very fast and simple, but is inaccurate and subject to instability. RungeKutta is much more accurate and stable, but takes more time to calculate each time step.

Particle and SeperatingPlane[2] are two very different concrete implementations of the abstract class RigidBody. In essence, a Particle is the disk of finite radius described earlier, while a SeperatingPlane is a geometric convenience for collision detection and response. By default, Particles have null for a surface normal and finite mass and inertia, while SeperatingPlanes have a surface normal but have infinite mass and inertia. There is also a class called Vehicle2D which extends Particle to give it snazzy graphics but no other significant functionality.

PhotoSensor and TouchSensor provide the two major sensing modalities in RobotWorld. TouchSensor reports only the amplitude of collision forces on the body it is attached to, if any. PhotoSensor is sensitive to all light sources on the playing field with a gaussian falloff $\mathcal{F}(\xi, \phi) = \xi g(\phi, \sigma)$ where ξ is the radiation intensity, ϕ is the relative angle from the sensor axis in radians, σ is a relative angle that defines the cone of sensitivity, and g is the gaussian function $g(\phi, \sigma) = e^{-|\phi|^2/\sigma^2}$. The radiation model is responsible for calculating ξ and ϕ at the sensor for each light source on the

[2]Yes, the class name was misspelled. One of my greatest pet peeves is a hacker who can't spell, and yet I let this slip by... Perhaps a spell checker should be built into the Javac compiler. I will take the World Wide Web Consortium approach: standardize your misspellings. ("Referer" is now a standard cookie header, rather than "referrer.") For the rest of this document, I will refer to the class by its misspelled name: SeperatingPlane.

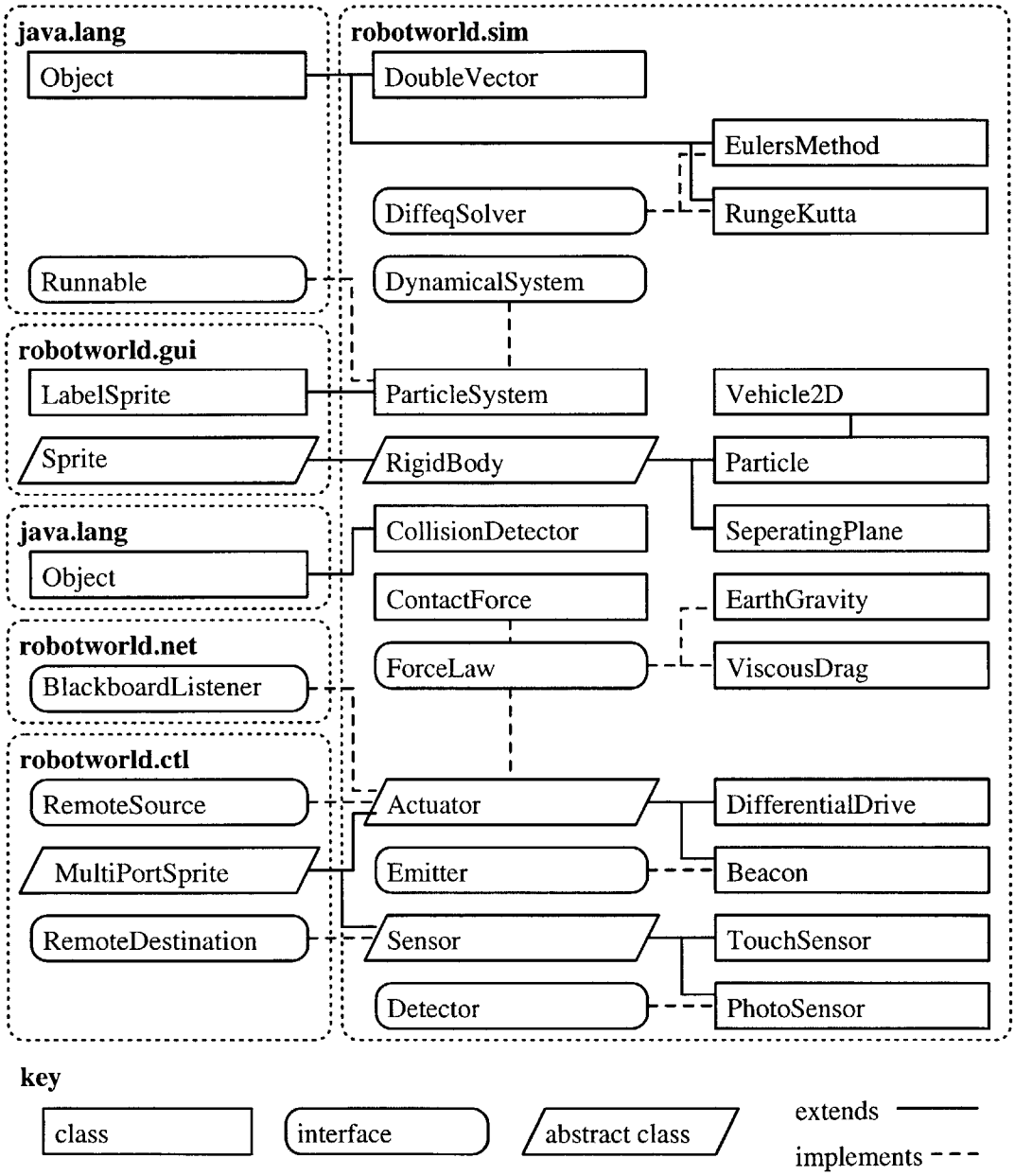


Figure 8-6: Class diagram of package robotworld.sim

playing field.

There are two concrete actuators so far, Beacon and DifferentialDrive. Beacon is a light source whose intensity is proportional to the actuator command value. DifferentialDrive is different in that is has two command values, one for the left wheel and one for the right wheel. It's responsible for implementing a proportional-gain controller to reach the commanded values over time as well as calculating the resulting forces and torques on the body, including the action of centripetal force.

There are two built-in optional force laws, EarthGravity and ViscousDrag. Each operates on the entire set of rigid bodies on the playing field when instantiated. EarthGravity simply applies a constant force vector to all bodies. ViscousDrag is slightly more complicated, but essentially resists the direction of motion proportionally to the body's velocity. A small amount of ViscousDrag helps stabilize the simulation.

Chapter 9

Putting It All Together

From reading Chapters 4–8, you should have a feeling for the package structure that makes RobotWorld possible. But how does this relate to the layers, and how do you decide which implementations to use at runtime? As it turns out, there are five core packages and five layers, but there is not a one to one correlation. Consider the robotworld.ide package. All five layers use this package for their user interface. The complete set of interrelationships is drawn out in Table 9.1. To emphasize their separation from the five core packages, the descendants of SpriteWindow and SpriteCanvas which implement the layers were moved into package robotworld, to reflect their presentational role in the application. See Figure 9-1 for a class diagram of these presentation classes. Now the only mystery left is how RobotWorld chooses particular implementations for its interfaces and abstract classes at runtime. Once this is understood, we can describe how to navigate from layer to layer.

An abstract factory makes the implementations available to classes, who simply need to present the name of the interface. For example, if the abstract factory was a coffeeshop worker, I could go up and ask for the FeaturedBlend (an interface). The coffeeshop worker looks up the mapping and finds CostaRican (an implementation), or whatever the day's blend is. So how exactly does this work?

Package / Layer	net	ide	gui	ctl	sim
Architecture	Canvas	Workspace	Canvas	-	Beanbag
Habitat	-	Workspace	Canvas	-	Beanbag
Morphology	Sprite	Workspace	Canvas	Canvas, Beanbag	Beanbag
Behavior	Sprite	Workspace	Canvas	Canvas, Beanbag	-
Code Editor	-	Workspace	-	-	-

Table 9.1: The relationship between the five layers and the five packages. Each cell of the table records what aspect of the layer is dependent on the given package, if at all. `Canvas` and `Workspace` indicate the appropriate classes in the class diagram (see Figure 9-1). `Beanbag` indicates that the beanbag for this layer contains beans from the given package. `Sprite` indicates that Sprites instantiated in this layer are dependent on the given package (used sparingly).

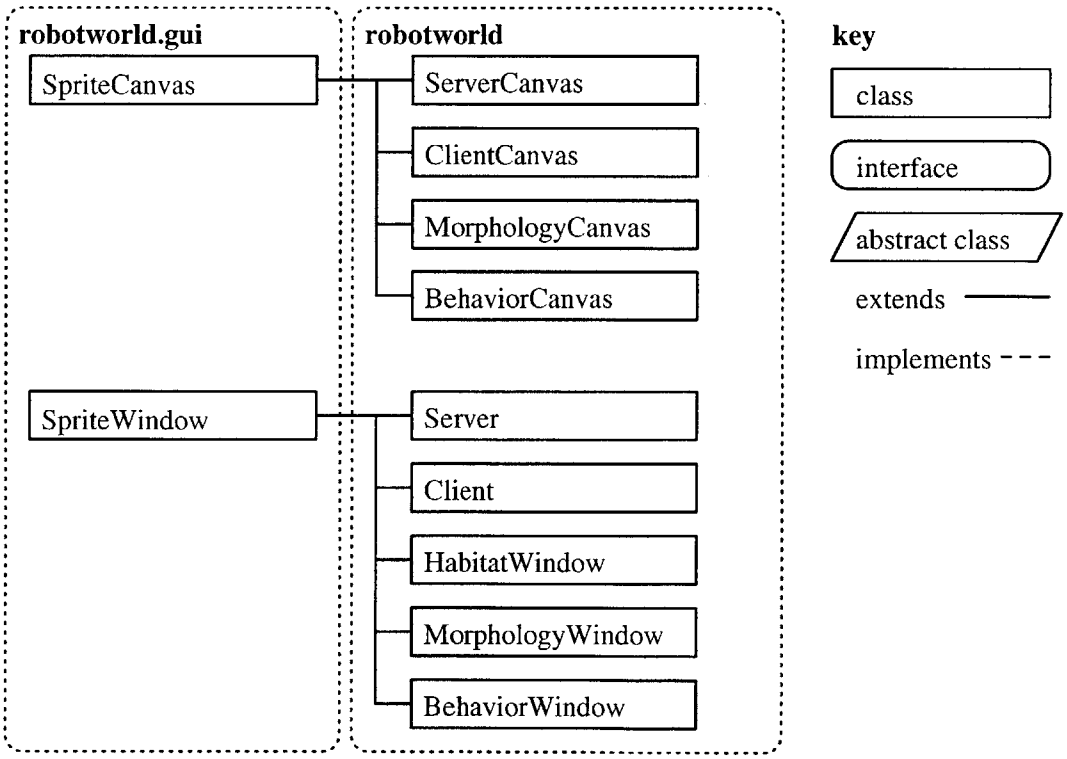


Figure 9-1: Class diagram of package robotworld

9.1 The Abstract Factory Mechanism

When RobotWorld is started up using robotworld.Server or robotworld.Client, the user supplies several command-line arguments. These are the names of properties files which reveal the interface-to-implementation mapping. There are two major tasks that RobotWorld.Main accomplishes: first, it iterates over the property files listed on the command line at startup calling loadProperties() on each. This method appends the standard .properties filename suffix to the name, look for the resulting name relative to the codebase, and if the file exists, attempts to add/update the system properties with those given in the file (in the form "name=value", one name/value pair per row. See the javadoc for java.util.Properties for more detail on proper file format.) In practice, this allows the setting of complex properties such as security policies and server aliases necessary to third party software such as Jini/JavaSpaces, as well as implementing object factories in RobotWorld.

Often times RobotWorld client code needs an implementation instance for a given interface or abstract class, and doesn't care/need to know which one is in use. This is especially important when entire 'product families' need to be swapped out due to a different operating environment (e.g. standalone shared-address blackboard vs. distributed JavaSpaced-based blackboard). The standard way to do this is to provide a name/pair mapping in a properties file corresponding to: fullyQualifiedInterface-Name/fullyQualifiedImplementingClassName. Note that this approach permits only one implementation per interface, which is fine in a 'factory' setting, but might not be suitable for other needs. This way, a client could simply ask for an implementation for a given interface by the use of Workspace.produce(), and supplying the Class object for the desired interface. This (the String) would then be used as a key to obtain the fullyQualifiedImplementingClassName as a String from the system properties that were loaded at startup. From this, a Class is obtained and an instance created, using the Reflection API. If all goes well, the implementation is then returned to the client class.

There are two other similar services that Workspace provides. One is getLayerFor(), and the other is getBeanbagForLayer(). When someone calls getLayerFor() and passes it a class which is understood to be a Sprite, it looks for a System property with the same class name. If a property is found, it is understood to be the class name of a Workspace, which is instantiated using the mechanism described above and returned. Otherwise, the name of the Sprite's superclass is examined for a System property. This process continues until a property is found, or the procedure walks off of the root of the class hierarchy, in which case null is returned. The result is a polymorphic property lookup, which is useful for defining a catch-all layer if nothing more specific can be found: CodeEditorWindow, mapped to Sprite itself. Assuming a Workspace was found, it can then configured to edit the Sprite by calling configureToEdit(). The other service provided, getBeanbagForLayer(), must do a little additional work to fill out the new Workspace's beanbag. It looks for a System property under the name of the Workspace. If found, the property is assumed to be a list of comma- or whitespace-delineated strings, each of which is a class name. These are added to a javax.swing.DefaultListModel, a Vector-like interface that integrates well with javax.swing.JList, which is used to represent the Beanbag. This is the same component that calls the SpriteCanvas's setCurrentBean() method when one of the beans is selected by the user. Now the mapping from Sprite to Workspace to Beanbag to Sprite is complete. Let's walk over the process step by step.

9.2 Sprites, Layers and Beanbags

Using this mechanism, we can traverse from one layer to another with a double-click. Let's start from the command line invocation of RobotWorld. This can be done from either Server or Client, both of which have a main() method. Let's assume it was the Server. First, it calls Workspace.loadProperties() and passes it all of the command line arguments. An appropriate command-line invocation might be:

```
C:> java robotworld.Server jdk singleuser layers
```

We can ignore the first two properties files for this discussion, and focus on layers.properties. The file is listed in Figure 9-2, we will be referring to it from time to time. Each of these properties is now loaded into the System properties.

Now, we return to the Server.main() method. It instantiates a Server, whose constructor then instantiates a ServerCanvas, and calls init() to initialize the Server GUI. This includes filling the beanbag with beans, which it obtains by calling Workspace's getBeanbagForLayer() method with itself as the argument. Now the Beanbag should have one item, ParticleSystem. Compare the Beanbag in the screen shot (Figure 9-3) with the Server listing in layers.properties (Figure 9-2). They should be identical. One final note: the last step in Server.main() is to create a LabelSprite, and call the Server instance's configureToEdit() method with the new Sprite as an argument. The Server will then call the ServerCanvas's setRootSprite() method. Everything we instantiate in the ServerCanvas will be a child of this LabelSprite. Since this LabelSprite has no parent, it will then become the "grandmother sprite," in other words, the root of the entire sprite tree. The choice of sprite to server as grandmother was not important, since it will never be directly displayed.

Now, moving on, the user can select the ParticleSystem bean. The JList should respond by calling the ServerCanvas's setCurrentBean() method. The user now instantiates a ParticleSystem on the ServerCanvas, which should look like the atomic symbol in the screen shot. When the user double-clicks on the ParticleSystem, the SpriteCanvas must first determine that the click landed on a Sprite, using find(). Then, since it was a double-click, it calls its own openSprite() method. This method asks Workspace.getLayerFor() to return a Workspace which knows how to the edit the ParticleSystem. Looking up the System property for ParticleSystem gives HabitatWindow. Check Figure 9-2 to see that this is true. The HabitatWindow constructor is very similar to the Server constructor. It instantiates a vanilla SpriteCanvas instead of a ServerCanvas, and calls init() to initialize the GUI. Workspace.getBeanbagForLayer() is called again, with the HabitatWindow as an argument. The result is a list of two

```
#
# MAPPING SPRITES TO LAYERS
# These properties establish the transitions between layers of RobotWorld.
# The mapping:
#    Foo = Bar
# Translates to:
#    When you double-click on sprite Foo, instantiate the layer Bar
#    to edit the Foo.
#
robotworld.gui.Sprite = robotworld.ide.CodeEditorWindow
robotworld.sim.RigidBody = robotworld.MorphologyWindow
robotworld.sim.ParticleSystem = robotworld.HabitatWindow
robotworld.ctl.CompositeController = robotworld.BehaviorWindow
#
# MAPPING LAYERS TO BEAN BAGS
# These properties establish the beans (sprites) available for
# instantiation on a particular layer.
# The mapping:
#   Bar = Foo1, Foo2, ... FooN
# Translates to:
#   Sprites Foo1, Foo2, ... FooN should appear in a bean bag within
#   layer Bar, allowing users to choose which sprite to instantiate next.
#
robotworld.MorphologyWindow = \
        robotworld.sim.PhotoSensor, \
        robotworld.sim.PhotoSensor.Left, \
        robotworld.sim.PhotoSensor.Right, \
        robotworld.sim.TouchSensor, \
        robotworld.sim.Beacon, \
        robotworld.sim.DifferentialDrive, \
        robotworld.ctl.CompositeController, \
        robotworld.ctl.RemoteConnection
robotworld.HabitatWindow = \
        robotworld.sim.Particle, \
        robotworld.sim.Vehicle2D
robotworld.BehaviorWindow = \
        robotworld.ctl.Controller, \
        robotworld.ctl.CompositeController, \
        robotworld.ctl.SignalSource, \
        robotworld.ctl.SignalDestination, \
        robotworld.ctl.Connection, \
        robotworld.ctl.SensorPort, \
        robotworld.ctl.ActuatorPort
robotworld.Server = \
        robotworld.sim.ParticleSystem
robotworld.Client =
robotworld.ide.UniformWorkspace.About = file:about.html
robotworld.ide.UniformWorkspace.Java_Reference = \
        http://java.sun.com/products/jdk/1.2/docs/api/index.html
robotworld.ide.UniformWorkspace.Textbook = http://www-cs101.ai.mit.edu/ipij/index.html
robotworld.ide.UniformWorkspace.Users_Manual = file:manual.html
```

Figure 9-2: A properties file for the application layers.

items: Particle and Vehicle2D. Compare the screen shot (Figure 9-4) with the listing in Figure 9-2. The thread of control is returned to openSprite(), which now calls the HabitatWindow instance's configureToEdit() method, and passes it the ParticleSystem instance that was clicked on in the first place.

Things are fairly straightforward from here. Clicking on any RigidBody (e.g. the superclass of Particle and Vehicle2D) from the HabitatWindow will pop up a MorphologyWindow. See Figure 9-5 for a screen shot. Clicking on a CompositeController in the MorphologyWindow will pop up a BehaviorWindow. Figure 9-6 shows a screen shot. But now a few complications set in. If you click on a CompositeController in a BehaviorWindow, you get another BehaviorWindow. See Figure 9-7. This makes sense, since the BehaviorWindow "knows how to edit" the CompositeController, no matter what layer it's embedded in. And finally, since there are no other entries for the sprite-to-layer mapping, double-clicking on any other sprite will take you straight to a CodeEditorWindow. See Figure 9-8. Note that this is obtained by the same mechanism, getLayerFor(), and configured using configureToEdit(). However, getBeanbagForLayer() is not called. This is simply because the CodeEditorWindow doesn't have a Beanbag. You have now seen the five core packages of RobotWorld, and the layering mechanism which makes it flexible and extensible. See the User's Manual (Appendix B) for details on how to install, configure, run, and extend the RobotWorld simulation environment. The rest of this thesis discusses where to go from here.

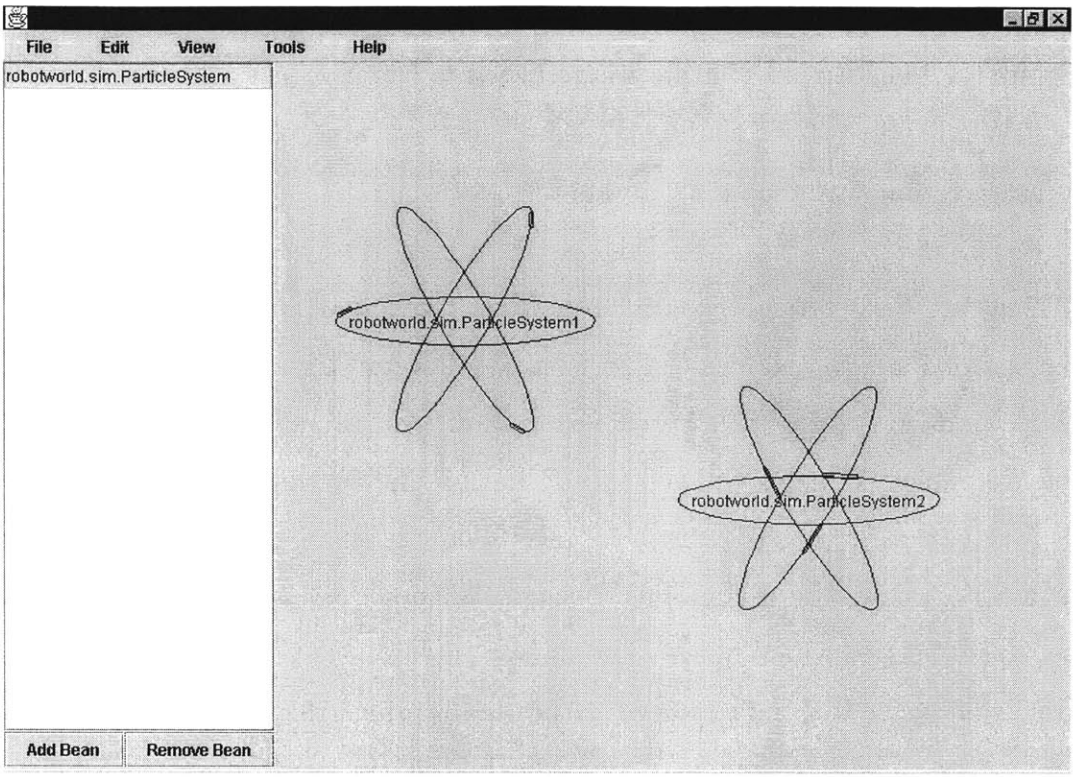


Figure 9-3: A screen shot of the Architecture Editor.

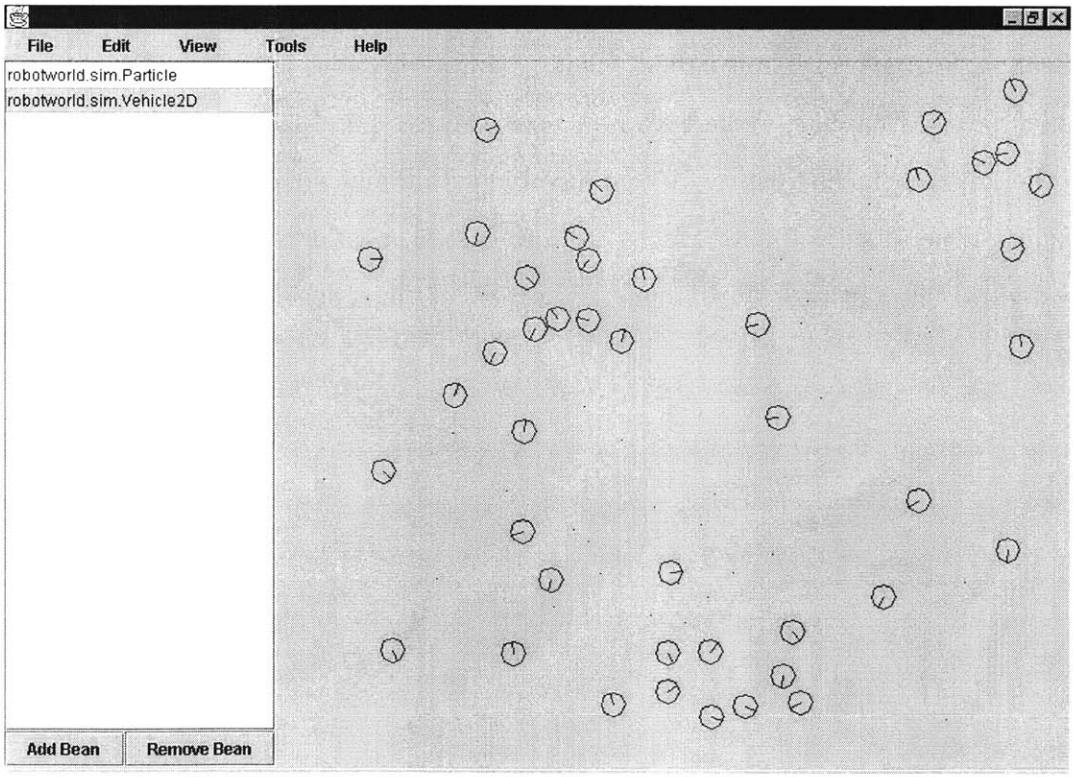


Figure 9-4: A screen shot of the Habitat Editor.

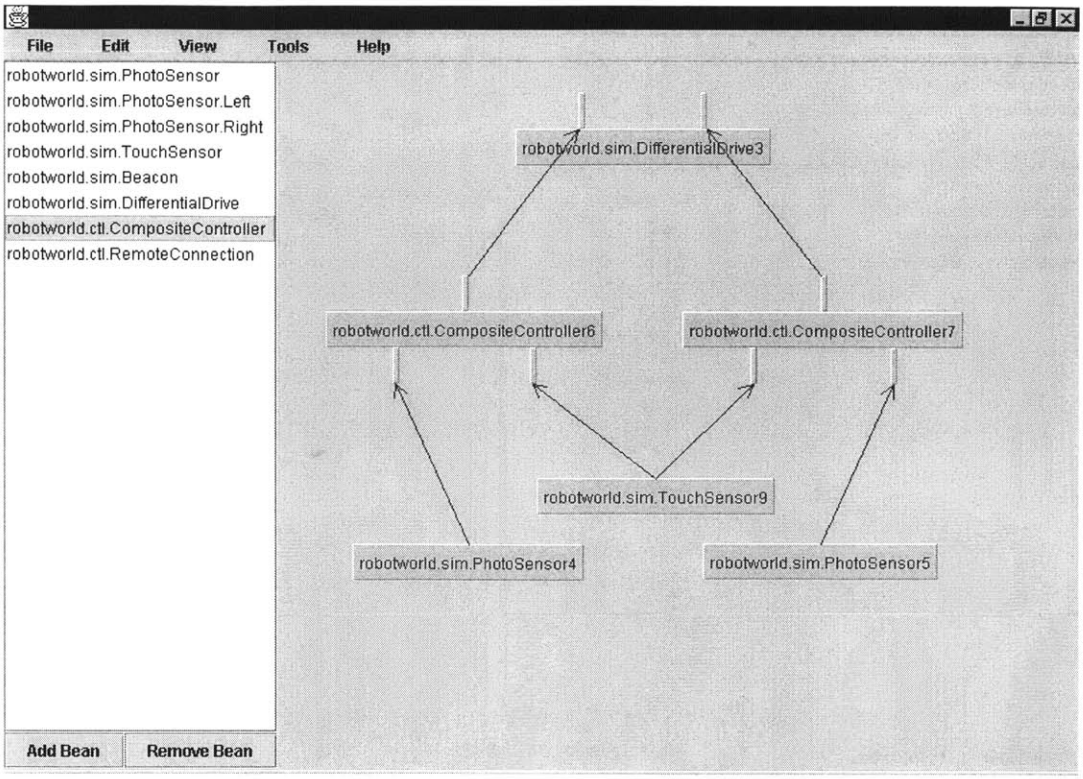


Figure 9-5: A screen shot of the Morphology Editor.

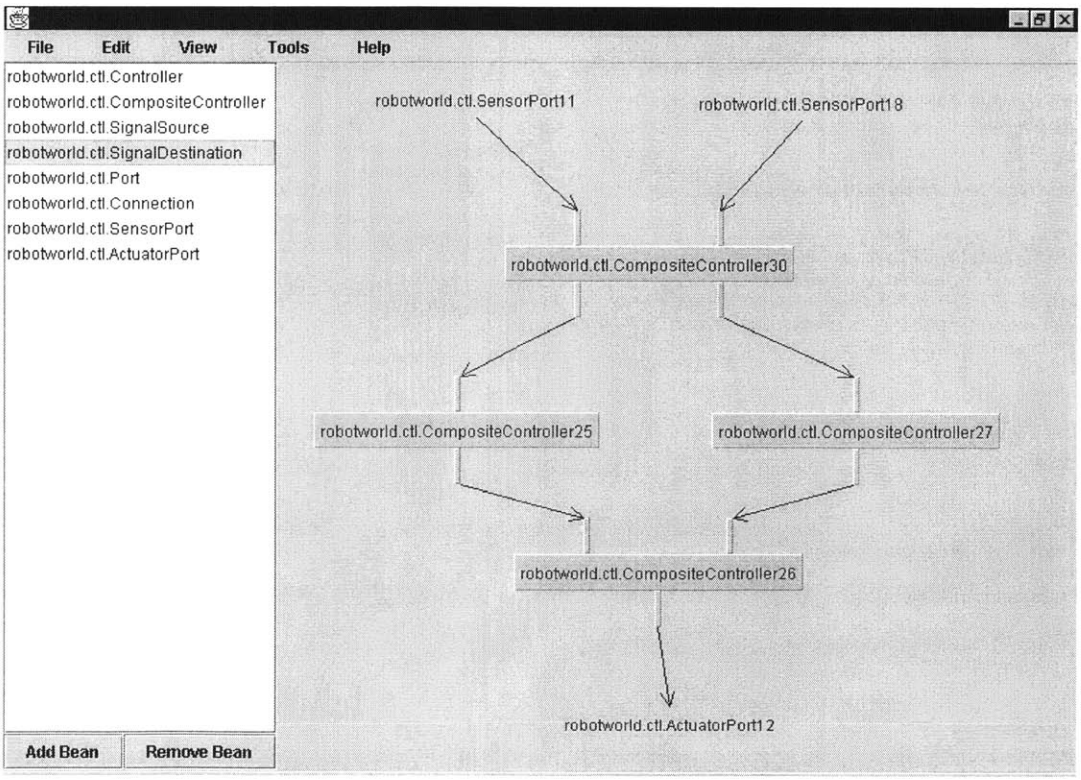


Figure 9-6: A screen shot of the Behavior Editor.

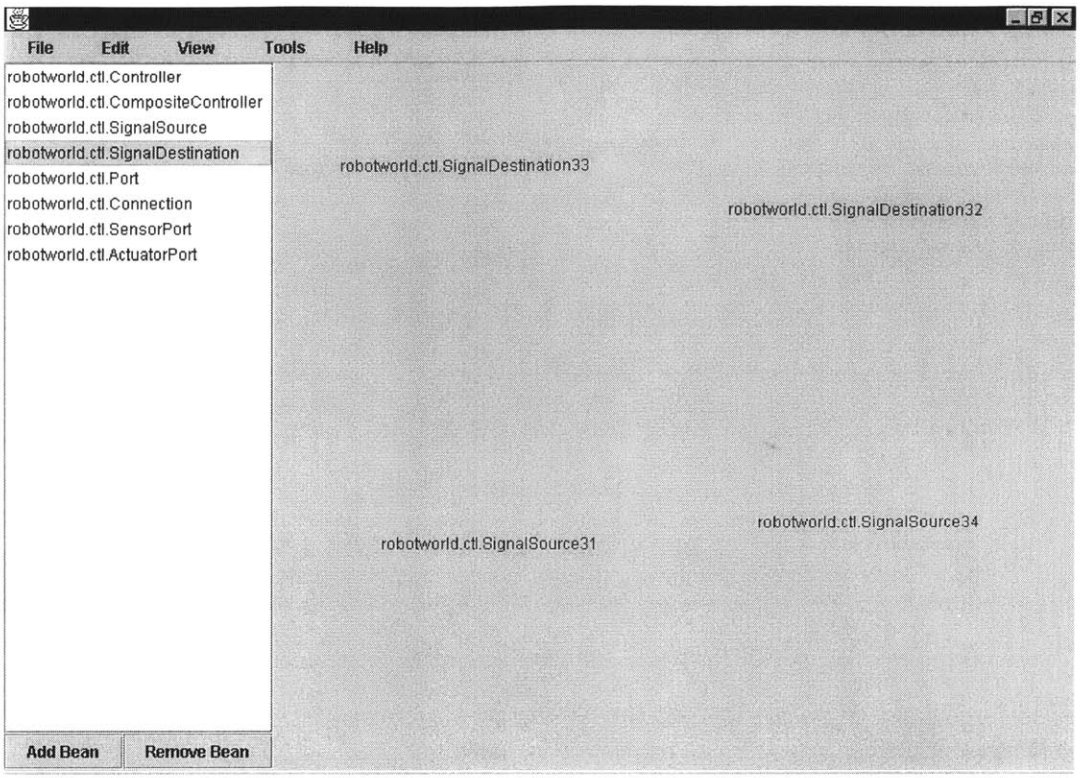


Figure 9-7: An alternate screen shot of the Behavior Editor.

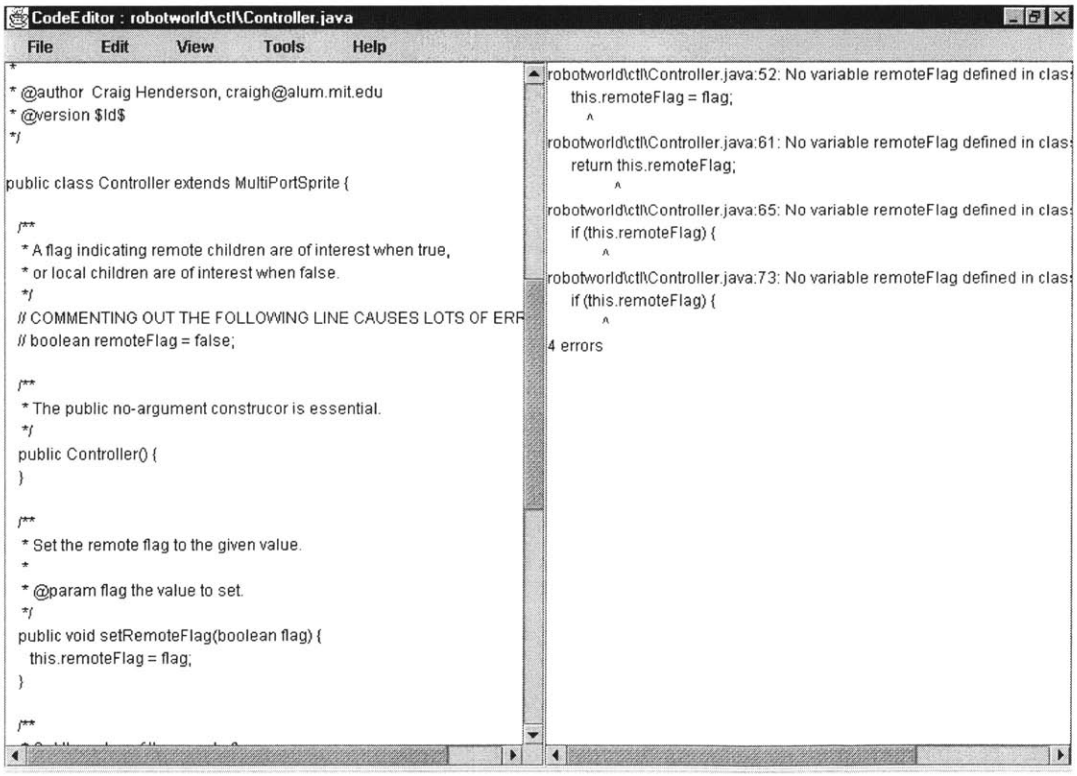


Figure 9-8: A screen shot of the Code Editor.

Chapter 10

Problem Set Examples

This chapter attempts to illustrate how RobotWorld could be used throughout the term of an introductory computer science class (CS101). Several examples follow, starting with observation of different vehicle behaviors in the visual interface, to writing short expressions and statements to drive a robot, to more involved programming activities.

10.1 Phototaxis

Based on the realization of Braitenburg's vehicles in software, the purpose of the first scenario is threefold: to build a system of interacting entities, to learn the anatomy of a self-animating Java class, and to experiment with different strategies that give rise to agent phototaxis and obstacle avoidance.

10.1.1 What are Braitenburg Vehicles?

Valentino Braitenburg [11] describes a series of thought experiments that reveal successively more complex patterns of behavior in some very simple hypothetical vehicles with elementary sensors and actuators. The first half of the book describes the thought experiments and paints the resulting behaviors in anthropomorphic terms,

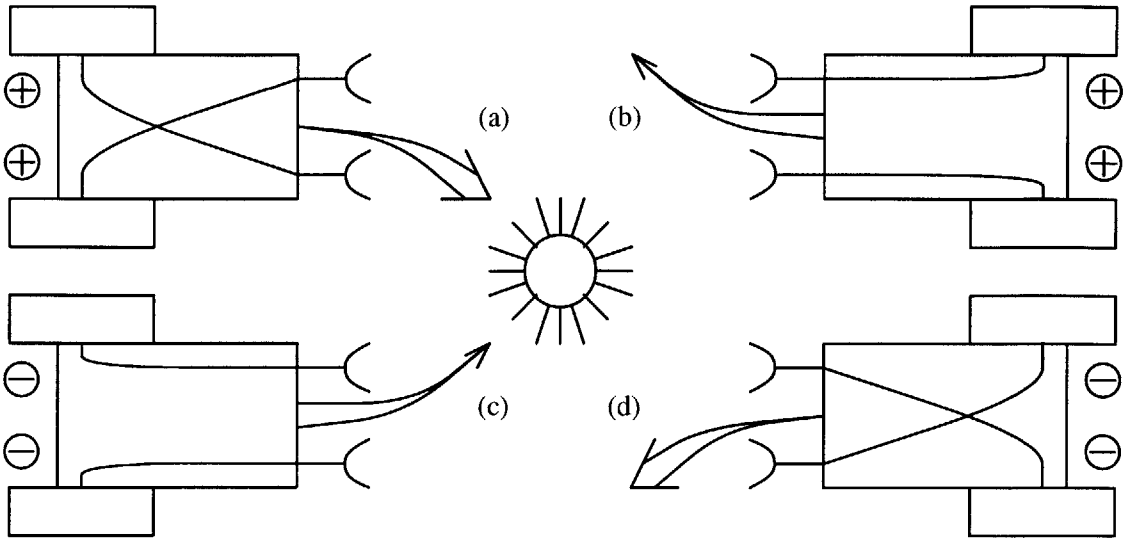


Figure 10-1: Four photo-sensitive vehicles: (a) Aggression, (b) Fear, (c) Love, and (d) Curiosity.

while the second half of the book provides more neurobiological background and suggests parallels in higher-level cognitive tasks. One might describe the central theme of the book to be the concept of "downhill synthesis and uphill analysis," meaning that a vehicular control system capable of a complex set of behaviors can be designed in a matter of minutes, while someone without access to its internals would postulate a tangled mass of interrelated mechanisms that would take months if not years to sort out. By presenting this kind of material, one would open the way for further exploration using a synthetic methodology. This could be used to study natural phenomena as described in Chapter 2, or one could use this as a way to study interactive programming. Some examples from the book will make this clear.

Imagine a flat terrain where these thought-vehicles live. See Figure 10-1. Each vehicle has two photosensors in the front, which measure the intensity of light incident. They also have two motors in the back on a differential drive mechanism which allows each motor to independently rotate a back wheel. The motors can move forwards or backwards. They also have passive coasters in the front, to handle whatever your concerns are about balance. We also have special devices which we can use to interconnect these sensors and motors. Some of these maintain monotonically

increasing [1] relationships between input and output currents (denoted with a $\oplus$), and some of them maintain monotonically decreasing [2] relationships (denoted with a $\ominus$) between input and output currents.

Fear has direct same-side connections. It quickly turns away from light sources, and slows down until it is hidden in darkness. However, if pointed directly at a light source, it will speed up until it collides.

Aggression has direct crossed connections. It seeks out light sources, driving faster and faster until it collides with them.

Love has inverse same-side connections. It seeks out light sources just like Aggression, but slows down until frictional forces bring it to a stop.

Curiosity has inverse crossed connections. It slows down in the presence of light sources, but turns away from them just like Fear. Eventually it leaves, speeding through the darkness until it finds another light.

As you can see, a strange and interesting repertoire of light-orienting behavior (phototaxis) is available from a very simple set of components and design principles. If you had encountered these devices, watching them cavort and interact with a number of light sources, you would most likely have projected a number of different attributes to the little critters: "Fear likes to hide in the darkness. Most of the time Fear just runs away, but sometimes it'll lash out at a light source in panic." And then: "Aggression loves nothing better than to bat around light sources." With a tool such as a Virtual Oscilloscope (described in Section 10.2), this observed behavior can be correlated with hard data. The synthetic approach can be supplemented with a little analysis to make the microworld more understandable.

---

[1] A *monotonically increasing* function $f(x)$ has the property that $f(x_1) \geq f(x_0)$ as long as $x_1 \geq x_0$. In other words, the derivative $f'(x)$ is never negative.

[2] A *monotonically decreasing* function $f(x)$ has the property that $f(x_1) \leq f(x_0)$ as long as $x_1 \geq x_0$. In other words, the derivative $f'(x)$ is never positive.

10.1.2 Exploring Fear and Aggression

Synopsis

The student begins by placing one vehicle and one light in the playfield. Nothing will happen yet. If the student launches the Brain Editor, they will see that there are no connections between the sensors and actuators. They should then connect one photosensor to a stepper motor on the same side. The vehicle will spin about. By connecting the other photosensor to the other stepper motor, the vehicle should arc away from the light until it hits a wall (this is Fear). If the student deletes the same-side connections and replaces them with crossed connections, the vehicle will suddenly arc towards the light and attempt to ram into it. The light can be thought of as a luminescent hockey puck: it will rebound elastically, and the vehicle should continue batting it around indefinitely (this is Aggression).

Choreography

1. Student types in "java robotworld.Server" at the console.
2. The Architecture Editor Window pops up. On the left is the BeanBag (a list of beans). On the right is the Canvas.
3. Student selects a ParticleSystem from the Beanbag.
4. Student clicks into the Canvas to create a ParticleSystem.
5. Student double-clicks on the ParticleSystem.
6. The Habitat Editor Window pops up. On the left is the BeanBag (a list of beans). On the right is the Habitat area where things can crawl around.
7. Student selects a Vehicle2D from the BeanBag.
8. Student clicks into the Habitat.
9. A Vehicle2D appears, but doesn't go anywhere.
10. Student double clicks on the Vehicle2D.
11. A Morphology Editor Window pops up. On the left is the BeanBag. On the right is a depiction of the arrangement of sensors and actuators inside the vehicle—currently empty.
12. Student selects and creates a Beacon atop this vehicle.

13. Student goes back to the Habitat Editor Window and creates another Vehicle2D.
14. Student double clicks on this Vehicle2D. A Morphology Window pops up.
15. Student selects and creates a DifferentialDrive. It has two input pins: one for each motor. The left pin goes to the left motor; the right pin goes to the right motor.
16. Student selects and creates a LeftPhotoSensor and a RightPhotoSensor.
17. Student selects RemoteConnection in the BeanBag.
18. Student clicks once on left photosensor and then once on left motor.
19. A RemoteConnection appears between those two components.
20. Back in the Habitat Editor Window, the vehicle begins to spin.
21. Student connects right photosensor to right motor.
22. Back in the Habitat Editor Window, the vehicle arcs away from the light.
23. Student clicks on the connection between left photosensor and left motor.
24. Student hits the DELETE key.
25. The connection disappears.
26. Student deletes the connection between right photosensor and right motor.
27. Student connects left photosensor to right motor.
28. Student connects right photosensor to left motor.
29. Back in the Habitat Editor Window, the vehicle should start "attacking" the light.

Notes

Currently one cannot directly edit the placement of sensors and actuators relative to the vehicle body. This would be the easiest if one could drag the components around to alter their positional attributes. Until then, direct manipulation of the vectors is required, by hardwiring relative positions into the constructor of a subclass. Thus LeftPhotoSensor is a PhotoSensor that places itself on the exterior of the vehicle, at 45 degrees counter-clockwise from the vehicle's heading. RightPhotoSensor is a mirror image of the LeftPhotoSensor. Also note that these are currently inner classes of PhotoSensor, namely PhotoSensor.Left and PhotoSensor.Right. However, the Class.forName() mechanism is unable to distinguish PhotoSensor as a class and

not a package, so this generates an error. Promoting these two inner classes to full classes is a trivial fix, but it is still not "the right thing" in the long run.

10.1.3 Turning Fear and Aggression into Love and Curiosity

Synopsis

At this point, the student should insert a Transformer node (which is a sort of interneuron) into the Brain Editor and double click on it. A Code Editing window should pop up, prompting them to write the body of the act() method, which is responsible for reading the node's input value, and writing the appropriate output value. First of all, they should just return a constant, then the fresh input value (obtained by a method call), and then the negative of the fresh input value. Each time, they should compile their code and instantiate the class into the Brain Editor. By connecting their new Transformer object between a photosensor and stepper motor (either same-side or crossed connections), they will observe several new behaviors, including Love and the Curiosity. They should feel free to experiment with more complex expressions and observe and explain the resulting behaviors.

Choreography

1. Student selects Transformer from the BeanBag in the Behavior Editor.
2. Student instantiates a Transformer into the Behavior area.
3. Student opens the Transformer by double clicking on it.
4. A Code Editor Window pops up. On the left is a list of members of the Transformer class (constructors, fields, and methods). A special entry named "All" is currently selected. On the right is the code for Transformer (read-only).
5. Student selects Edit → Extend This Class. A new Code Editor Window pops up.
6. Student selects "Override the act() method" from the member list.
7. Student types in a simple return expression.
8. Student selects Tools → Compile.
9. A "Save As..." window pops up asking for a name to save it under.

10. A message indicating the errors, or success, pops up. Click to dismiss.
11. Cycle until successful.
12. Back in the Behavior Editor Window, an instance of the subclass has been swapped for the Transformer.
13. Student connects left photosensor to the instance, and the instance to the left motor, or variations on this theme, and observes the resulting behavior.

Notes

The Transformer class will need to be designed for this scenario. It is a Controller with one input and one output. Edit → Extend This Class is an option that doesn't exist but would be nice, in lieu of it they can use File → Save As, come up with a new name, and then modify the class to extend the original Transformer. Also, items 4–6 above are based on a vision for the BeanSynthesizer, which has not been written (see Chapter 11, Future Work). It's just a fancy interface for a text area which masks all of a Java class except what a student really needs to see. For instance, all a student needs to see in this section is the return statement in the act() method. In lieu of such a tool, the instructor can provide the full source code and indicate the line to edit with comments, e.g.:

```
public double act()

   /*=========================================*
    * YOU NEED TO MODIFY THIS RETURN STATEMENT *
    *=========================================*/
   return 0;

}
```

10.1.4 Colliding: When Bad Things Happen to Good Robots

Synopsis

The student should have noticed that these vehicles have until now been indifferent to collisions. They will now design a Transformer2 which will monitor a TouchSensor

on the vehicle and take the appropriate actions to back it up. The Transformer2 is slightly different, in that the student will have to write the entire public void act() method. They should add a new input to the node which will connect to the output of their previously written Transformer. And since the TouchSensor delivers only zero or the magnitude of the force, they will have to use flow-of-control to decide what to tell the stepper motor. They will also need to provide state, in the form of a counter, which will allow a reversal of direction to continue for several cycles. (The field area should be exposed with the click of a checkbox.) Now they can test it out.

Choreography

1. Student selects Transformer2 from the BeanBag in the Behavior Editor.
2. Student instantiates a Transformer2 into the Behavior area.
3. Student opens the Transformer2 by double clicking on it.
4. A Code Editor Window pops up. On the left is a list of members of the Transformer2 class (constructors, fields, and methods). A special entry named "All" is currently selected. On the right is the code for Transformer (read-only).
5. Student selects Edit → Extend This Class. A new Code Editor Window pops up.
6. Student selects Edit → Disable Auto-Reflect. The method signature will now have to be written explicitly by the student.
7. Student selects "Override the act() method" from the member list.
8. Student types in a complete method body with a simple return expression.
9. Student selects Tools → Compile.
10. A "Save As..." window pops up asking for a name to save it under.
11. A message indicating the errors, or success, pops up. Click to dismiss.
12. Cycle until successful.
13. Back in the Behavior Editor Window, the old Transformer2 should be deleted and a new one created in its place.
14. Student selects the new Transformer2.
15. Student deletes connection from previously written Transformer to motor.

16. Student connects output of previously written Transformer to the Transformer2.
17. Student connects the TouchSensor to the other input of the Transformer2.
18. Student connects the output of the Transformer2 to the motor.
19. In the Code Editor, student enters a simple if-statement for the body of the act() method that returns one of two constants based on the value of the TouchSensor.
20. Student compiles and notes new behaviors of the vehicle. Cycle until vehicle goes forward when TouchSensor is not in contact, backward when in contact.
21. Student alters code so that when TouchSensor is not in contact, the motor current (computed by the previously written Transformer) is returned.
22. Student compiles and notes new behaviors in the vehicle.
23. Student selects "Add field" from the member list. A text field opens up.
24. Student enters (and possibly initializes) a field and attempts to compile it.
25. Cycle until success.
26. Student goes back to "Override the act() method" and alters logic to take advantage of the state in backing up the vehicle upon contact.
27. Student compiles and notes new behaviors in vehicle.
28. Cycle until satisfied.

Notes

This scenario requires a Transformer2 class, which is identical to Transformer except that it has two inputs instead of one. Also, note that the Edit → Disable Auto-Reflect is just one more planned feature of the BeanSynthesizer described above. By default, students shouldn't have to write the method signatures of methods they are overriding. It should be done automatically using the Reflection API (hence "Auto-Reflect"). Naturally, these "training wheels" should be disabled at the point at which students are ready to face the full complexity of method definition in the Java language.

10.1.5 Obstacle Avoidance: The Path to Self-Actualization

Synopsis

Finally, they will design from scratch a self-animating Transformer, which will detect obstacles ahead of time using time-to-contact, which is a sensor value that the vision system provides. Also known as τ, this seems to be one of those universal order parameters that govern the behavior of organisms with an optic system (hoverflies and gannets come to mind).

Choreography

1. In the Morphology Editor, student adds a TimeToContactSensor to their vehicle.
2. Student selects Transformer from the BeanBag in the Behavior Editor.
3. Student instantiates a Transformer into the Behavior area.
4. Student opens the Transformer by double clicking on it.
5. A Code Editor Window pops up. On the left is a list of members of the Transformer class (constructors, fields, and methods). A special entry named "All" is currently selected. On the right is the code for Transformer (read-only).
6. Student selects Edit → Extend This Class. A new Code Editor Window pops up. On the right is a blank text field—the student writes the entire class from scratch in this field. (This might be a great place to have a hook into an external IDE.)
7. Student attempts the following incremental designs:
   (a) A class that extends Transformer and compiles.
   (b) Implements Runnable, starts an animacy in its constructor, and provides a run() method.
   (c) While (true) loop that calls act(), and the act() method sleeps for some amount of time and then returns a constant.
   (d) The act() method returns their favorite expression based on the light level.
   (e) The act() method returns the former, unless the time-to-contact is greater than some threshold, in which case zero should be returned for the next few cycles (using a counter again).

(f) Extra credit: what does the difference between successive time-to-contact values indicate? You may want to monitor an Oscilloscope trace of the time-to-contact values, and associate points in the diagram with the activity of the vehicle. Now, how might you alter your vehicle design to seek perfectly soft landings instead?

Notes

This scenario requires a complex Sensor which uses an array of PhotoSensors to calculate the optic flowfield several image elements wide. The time-to-contact $\tau(t)$ is estimated using the PhotoSensors' distance $r(t)$ from the axis of motion, divided by the optic velocity $v(t)$ at each sensor. Several PhotoSensors can be packed close together to generate better estimates. See Kelso [44][pp190–192] for further discussion. The resulting TimeToContactSensor will output the τ value for student designs to handle.

10.2 Virtual Oscilloscope

Synopsis

A common way to introduce graphical user interfaces by writing a scribbler. This is just a Canvas which the student extends to add a mouse listener, a field to hold all the mouse click information, and a paint method to draw it on the Canvas. The result is a short Java class similar to the one described in Java in a Nutshell [23], or the Scribbler problem set on our project web pages [76].

A variant of the scribbler could serve as a scientific tool for understanding robot behavior—the oscilloscope. Rather than storing pairs of positions $[X, Y]$, the oscilloscope class would store pairs of state variables $[f(t), g(t)]$ of the robot. For instance, it could store velocity vs. time, velocity vs. acceleration, and so forth[3]. See Figure 10-2. This state information is continuously being updated on the blackboard. The stu-

[3]We have another problem set, Spirograph, which does something similar, except that students choose the functions $f(t)$ and $g(t)$ to draw.

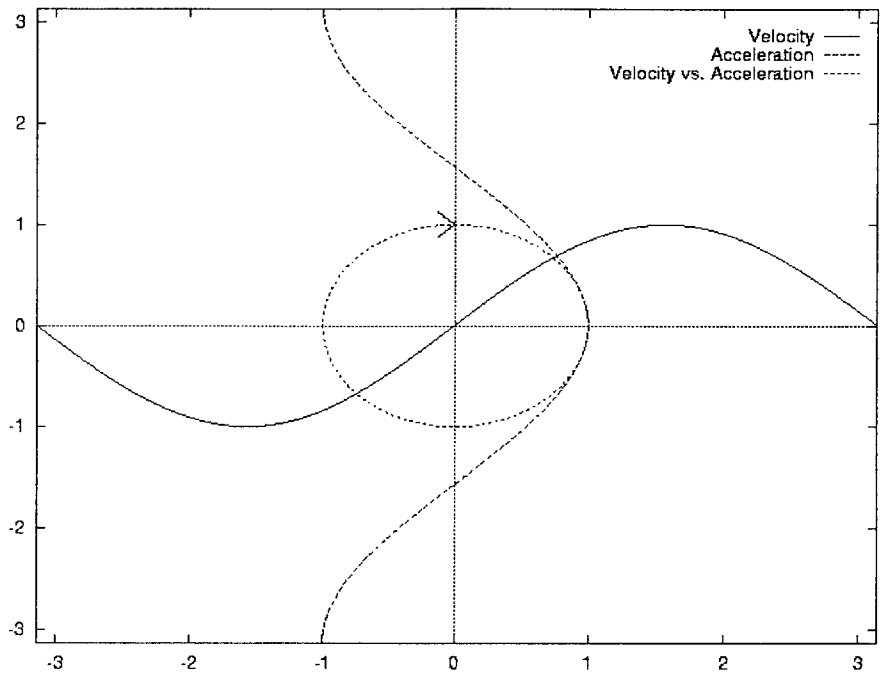


Figure 10-2: Three types of oscilloscope traces. Velocity is the horizontal input to the scope, and Acceleration is the vertical input. The arrow indicates the time origin and the direction of flow. This plot might describe an imaginary spacecraft with forward and reverse thrusters. It starts at zero velocity, with maximum thrust. The pilot backs off on the thrust, and when it reaches zero, the pilot starts up the reverse thruster. The spacecraft comes to a stop with maximum reverse thrust at the far end of the solar system, and ends up retracing its journey in mirror image all the way back to its starting point.

dent's job is to write a Canvas which listens to two or more Choice boxes, each of which displays the possible states for one axis of the plot. Based on these selections, they will need to read the appropriate data from the blackboard and store the pairs in a repository (or trace, as the electrical engineers say), and in their paint method draw all the $[f(t), g(t)]$. There is one more step needed to animate the Canvas. They need to implement Runnable, and kick off an animacy by creating and starting a Thread in the canvas constructor. The student can then have a while (true) loop in their run method that periodically redraws the canvas by calling repaint(). Side note: if the student extends javax.swing.JCanvas instead of java.awt.Canvas, double buffering is automatically taken care of. Now the fun part begins.

There are many directions the project can go from here. It all starts with a good abstract design for the repository, perhaps with the java.util.List interface from the JDK 1.2 API. This interface supports adding and removing elements, as well as clearing or iterating over the set. Now, if the student stores every pair of states in the repository, the trace will be either a continuous line as the robot "scribbles" its state onto the canvas, or a dust of points if the student's sampling rate is too low. If the student stores allows only one pair of states in the repository by calling clear() before every add(), the trace will be a single point, racing around the canvas as the robot moves around on the playing field. Other possibilities include storing only the latest N points for a streak effect, which could be very effective using a gradation of color, or only displaying every Kth point for a strobe effect (in the sense of strobing the data). Getting fancy, the students could mess with scaling and moving the plot, and add calibration and/or grid lines. For advanced students, storing the repository to a file or to the blackboard using serialization would also be a noble goal. The result of all this work is a tool which they can use to correlate the behavior of their robot with hard data, for example "seeing" the monotonically increasing or decreasing sensorimotor transfer functions described in Section 10.1 drawn out experimentally and in real time. Keep in mind that these don't have to be virtual robots. Since the information is being pulled from the blackboard, any real robot that knows how to write to our

blackboard can be used in place of the simulator. In fact, this Virtual Oscilloscope is probably useful enough that an implementation, complete with printing, loading and saving support should probably be made available to students from Day One, and then they can implement their own version at the point at which lectures and lab activities have prepared them for the programming challenge.

Choreography

1. Student begins with one of the four typical Braitenburg vehicles in the playing field.
2. Student double clicks to bring up the Morphology editor and adds an ObjectiveRealitySensor to the robot. This will make all robot state (e.g. velocity, orientation, ...) available to the Behavior subsystem, which is possible in a simulation but not on a real robot or organism. Later we will remove this crutch.
3. Student double clicks on the vehicle's Composite Controller, bringing up the Behavior Editor.
4. Student selects Scope from the Behavior Editor. Scope has two input pins ("probes"), and no output pins.
5. Student selects Edit → Extend This Class.
6. A dialog box pops up, asking for a file name. Student chooses one.
7. Student attempts the following incremental designs:
   (a) A class that extends Scope and compiles.
   (b) Implements Runnable, starts an animacy in its constructor, and provides a run() method.
   (c) While (true) loop that sleeps for some amount of time and then calls act().
   (d) The act() method reads data from the Horizontal and/or Vertical ports (SignalDestinations), depending on the value of getScopeMode(). This data is bundled into a Point or something similar, and the Point is stuffed into the repository via getRepository().add().
   (e) paintChildren() method is overridden to paint all the points onto the canvas.
   (f) Now, student selects their class from the beanbag and instantiates into the Behavior Editor.
   (g) Student connects one SensorPort to the Horizontal input, and another to the Vertical input.

(h) Student double clicks on their Scope instance. This brings up a window with their plot being continuously displayed!

(i) There are three modes to choose from in a box on the left of the plot: Horizontal vs. time, Vertical vs. time, and Horizontal vs. Vertical. Student should toggle between these modes to debug their painting code.

(j) They will also want to try connecting the probes to different SensorPorts and correlate this to events going on in the playing field.

(k) Extra credit: Experiment with streaking and strobing by ListIterating over the repository. Also, try changing the sleep time of the run() method to see how this changes the continuity of the plot. Finally, add some of the other advanced features such as calibration, scaling, moving, loading, saving, and printing. This will probably require digging into the canvas and window classes as well.

Notes

This scenario requires the following classes to be designed: ObjectiveRealitySensor, and Scope. The ObjectiveRealitySensor records the entire state vector of the vehicle, in a way that makes it easy to retrieve its component parts by name (linear and angular position and velocity). Scope is a MultiPortSprite which provides methods for getting and setting the Scope Mode, which can be one of three settings: Horizontal vs. Time, Vertical vs. Time, or Horizontal vs. Vertical. It also provides two input pins (SignalDestinations) built in, one labeled Horizontal, one Vertical. It should have a method for getting and setting the Repository, which should be a java.util.List.

10.3 Flocking

Synopsis

Based on the emergent properties of Reynolds' boids, the purpose of this scenario is also threefold: to design from scratch a collection of interacting entities, to learn the meat and potatoes of event handling and GUI design, and to compose simple local rules that give rise to collective flocking behavior.

"Stated briefly as rules, and in order of decreasing precedence, the behaviors that lead to simulated flocking are:

1. Collision Avoidance: avoid collisions with nearby flockmates.

2. Velocity Matching: attempt to match velocity with nearby flockmates.

3. Flock Centering: attempt to stay close to nearby flockmates." [66]

All of the rules above are to be implemented relative to the vehicle's perceptual system and not in some world coordinate frame. A simple implementation involves using two Sensors for the RobotWorld simulator. They generate lists of relative position and velocity vectors of all other agents on the playing field. All vectors are calculated with respect to the agent's frame of reference. Take, for example, three agents which are moving in the same direction and at the same speed. Each has to monitor two position vectors and two velocity vectors. The velocity vectors will remain zero so long as they all maintain the same speed. The position vectors have a magnitude equal to the inter-agent distance. The students' challenge is to design a system of interacting behavior modules (Controllers) that use this data to implement the three rules. They will need the math routines in RobotWorld.sim.DoubleVector or something equivalent to average the vectors.

A more complex, open-ended project would involve using only arrays of time-to-contact sensors to do the same thing using the techniques in Section 10.1.5. After all, geese don't use Cartesian coordinates. This would probably be worth a term paper in a class on artificial life. It would certainly be worth publishing if you could get it to work on real robots.

Choreography

1. Student begins with a Vehicle2D in the playing field.

2. Student double clicks to bring up the Morphology editor and adds a FlockPositionSensor and a FlockVelocitySensor to the robot. This will make relative flock state available to the

Behavior subsystem, which is possible in a simulation but not on a real robot or organism. Later we will remove this crutch.

3. Student double clicks on the vehicle's Composite Controller, bringing up the Behavior Editor.
4. Student selects Transformer from the Behavior Editor.
5. Student selects Edit → Extend This Class.
6. A dialog box pops up, asking for a file name. Student chooses one.
7. Student attempts the following incremental designs:
   (a) A class that extends Transformer and compiles.
   (b) Implements Runnable, starts an animacy in its constructor, and provides a run() method.
   (c) While (true) loop that sleeps for some amount of time and then calls act().
   (d) The act() method reads a signal from the input port. This signal is a list of DoubleVectors. Each DoubleVector is the relative position of a flockmate. The act method should also write a signal to the output port. This signal is a DoubleVector, which is a desired velocity vector. Thus, it is the student's challenge to implement a function which takes a list of positions, and generates a velocity command, such that Rule#1 is implemented (Collision Avoidance).
   (e) Now, student selects their class from the beanbag and instantiates into the Behavior Editor.
   (f) Student connects the FlockPositionSensor to their class's input, and the VelocityActuator to their class's output.
   (g) Student serializes their robot to a file.
   (h) Student instantiates two other identical robots into the playing field. Do they collide or not?
   (i) They will also want to implement Rule#2, using another Transformer class which will be connected between the FlockVelocitySensor and the VelocityActuator. Now, are there collisions or not? What happens if you disable the position monitoring Transformer?
   (j) Finally, implement Rule#3 using a third Transformer. What should it be hooked up to? How does it perform with or without the other two Transformers?
   (k) Extra credit: Purge the unrealistic sensors and actuators from the system. This includes FlockPositionSensor, FlockVelocitySensor, and VelocityActuator. Now, try to get the same behavior patterns by using only a set of time-to-contact sensors around the periphery of your robot, using a DifferentialDrive for steering and propulsion.

Notes

The following classes will need to be designed for this scenario: FlockPositionSensor and FlockVelocitySensor, described above, plus VelocityActuator. The VelocityActuator takes a desired velocity vector for the vehicle and generates an acceleration vector which will bring the actual velocity towards the desired velocity. The proportional gain controller used for DifferentialDrive will be sufficient.

Chapter 11

Future Work

11.1 For RobotWorld Developers

Let's face it, my wish list is long and life is short. Immediately following are the features which are necessary before I would consider RobotWorld to be at version 1.0 final, ready for export to other institutions for use in their curricula. The name of each feature is given in **bold**, followed by an explanation of the problem, the benefits of solving it, and a rating of *difficulty* of the implementation and the *payoff* of the result. The rating R runs on a logarithmic scale from 1 to 10 (like the Richter scale). Thus a difficulty rating of 2 is an order of magnitude more difficult than a rating of 1. The criteria used to apply a difficulty or payoff rating were not picked from a hat, they were picked from Table 11.1. For example, a change that involved only one class, for the purpose of runtime efficiency, would have a Difficulty of 1 and Payoff of 2. The desired features for the final version of RobotWorld are sorted by Payoff level for the reader's convenience.

R	Difficulty
1	One class affected.
2	Search and replace a programming idiom.
3	Design or re-design affecting one package.
4+	Design or re-design affecting many packages.

R	Payoff
1	Consistency
2	Efficiency
3	Transparency
4+	Flexibility

Table 11.1: Difficulty and Payoff rating systems

11.1.1 Consistency

The world is in a state of flux. My brain is in flux, my blood caffeine level is in flux, the Java language is in flux. As a result, there are certain programming conventions which were not adhered to consistently across RobotWorld. This may sound like a cosmetic change, but in actuality this may result in anything from an unresponsive GUI to conceptually distinct classes being hidden inside other classes. Fortunately, these changes are relatively easy to make.

SpritePropertyEditor Complete migration to Swing. Only one class is still based on AWT, robotworld.gui.SpritePropertyEditor. The change is simple, but the reason for the migration is less so. Ninety nine percent of RobotWorld is based on Swing. Applications must be all AWT, or all Swing. AWT is based on heavy-weight components, which require system resources (windows/frames), whereas Swing is based on lightweight components, which are pure Java. Unpredictable bits of GUI functionality are lost when the two frameworks are mixed. This is all explained on the Swing Connection [27]. There is another problem with this class. It's hard wired to edit only one Sprite property: its label. We need to be able to edit any of its properties, by looking for its setter and getter methods—a Java bean convention—using reflection. See Chapters 10 and 12 of Java in a Nutshell [23] to learn more about beans and reflection. BeanAnalyzer will help immensely. *Difficulty: 1, Payoff: 1.*

Proportional Gain Controller code needs to be promoted to its own class. Currently the code is embedded inside the apply() method of DifferentialDrive. It should include getter and setter methods for the gain constant, as well as a property which tells it whether to be OPEN_LOOP or CLOSED_LOOP. This will allow for student experimentation. *Difficulty: 1, Payoff: 1.*

RadiationModel code needs to be promoted to its own class. Currently the code is embedded inside the computeForces() method of ParticleSystem. See Chapter 8 for details. *Difficulty: 1, Payoff: 1.*

CodeEditor provides auto-indent and brace matching functionality. Unfortunately, the Swing libraries have been in flux, and what used to work no longer works. This class probably needs to become a javax.swing.event.CaretListener, and adhere to the final Swing API. Read up on the Java Developer's Connection for the latest details on Swing programming. *Difficulty: 1, Payoff: 1.*

11.1.2 Efficiency

These are not just random chunks of code that could be optimized. They are listed because they could potentially cause sluggishness or bloat in the system. All of these areas, if left untreated, will cause a major headache as the system scales up and resource spending becomes critical to RobotWorld's usability.

Numerical Stability *This is a showstopper!* At first I thought the problem was robotworld.sim.RungeKutta. Now I believe the problem is in the size of the time step. What complicates this is that as the time step decreases, so does the "playback rate" of activity on the playing field. After a certain point, you may not have the patience to watch. Large time steps seem to cause a robot to goes unstable. For instance, Aggression will start to approach a light source, and then suddenly start accelerating in reverse. It never seems able to recover. It's entirely possible that the math in any of the Beacon, PhotoSensor, DifferentialDrive, or RadiationModel classes is not quite right. It might be worth checking over. The testing process can be sped up by fixing `SpritePropertyEditor` first. This is because part of the problem is mucking with time steps, the sleep time of the simulator, etc. If these are made into Bean-like properties, then they can be modified during runtime. Otherwise the experimenter will be forced to recompile for every tweak of a constant... Not much fun, if you ask me. *Difficulty: 2, Payoff: 2.*

AbstractEntry Solve the BlackboardEntry inheritance explosion problem. Here is the crux of it: the JavaSpaces specification requires everything on the blackboard to implement the net.jini.entry.Entry interface. We wish to use Java-

Spaces, however, we do not want to lock this interface into the RobotWorld design and force it on our users. Also, what if we change our minds and want to use a different implementation, perhaps IBM's TSpaces or Objectspace's Voyager? We want the RobotWorld community to have independence from commercial software, as well (e.g. SimpleBlackboard). So how do we pull off this feat? Currently a very ugly thing happens. We require $O(NK)$ classes, where N is the number of Blackboard entry implementors (e.g. JiniEntry, SimpleEntry), and K is the number of entry abstractions (e.g. WorkspaceEntry, SignalEntry). The optimal solution uses the Bridge pattern, which requires only $O(N + K)$ classes. The basic idea is to separate the Implementors and Abstractions into two inheritance hierarchies which communicate only via their interfaces. The tricky part will be preserving as much of the JavaSpaces functionality as possible: wildcards, type matching, bytestream equivalence, etc. Read the JavaSpaces specification for all the details. It may require some judicious work on the part of the programmer to decide what is worth preserving. The brains of the whole operation will be in the root of the Abstraction hierarchy, most likely located within equals() and hashCode(). See SimpleEntry for an idea of what this might entail. Also, it would be entertaining to write a networked version of a blackboard that didn't depend on JavaSpaces, although there might be significant effort duplication. Still, it might be a worthy project. It would need a better data structure for quick lookups on large spaces: perhaps a tree-based or hashing implementation. *Difficulty: 2, Payoff: 2.*

Collections Migration Complete migration to Collections. Currently there is a mix between Collections and the more traditional Vectors and Hashtables. The migration should speed things up a bit, since Vectors and Hashtables are synchronized by default, whereas Collections are not. You can use a synchronizing wrapper on a Collection if you really need it. This will also have the side effect of consistency in programming style, which is important for those who need to work with the source code. *Difficulty: 2, Payoff: 2.*

Optimize Network The entire Sprite tree is being serialized over and over to the blackboard, rather than incremental changes. This is because there is no separation between the sprite tree and sprite state. If state were captured in a separate class, then the sprite tree could be serialized in response to user events, while the sprite state could be serialized on an ongoing basis. See `Model-View Separation` to see how this fits into a larger pattern of changes. *Difficulty: 3, Payoff: 2.*

11.1.3 Transparency

Intuitive Graphics RobotWorld desperately needs graphics to reflect the sensor-actuator geometry. A look at the Morphology Layer of Figure 3-1 should give a clue of what I mean: the radial boundary of the robot, the placement and orientation of the sensors relative to the center of mass, the length of the axle, radius of the tires, and so forth. All of these should be editable via `SpritePropertyEditor` by double clicking on any of the components. This way, students can immediately tell if their robots are cross-eyed, identify its blind spots, etc. *Difficulty: 2, Payoff: 3.*

Blackboard Interaction This is actually pretty simple, and consists of two changes. First, the SimpleBlackboard implementation preserves object identity, when the Blackboard interface says that blackboards do *not* preserve identity, they operate on copies. This inconsistency could mislead users who write code using one implementation and then switch. Currently, SimpleBlackboard stores the entry references themselves on a write() or notify(). What it needs to do is to create a java.rmi.MarshalledObject with the entry as a constructor argument, and store that instead. The matches() method should be rewritten to operate on MarshalledObjects—take a look at the Blackboard javadoc for the bytestream equivalence test. Finally, when returning an entry as a result of read() or take(), the entry should be reconstituted using MarshalledObject.get(). That was the first change. The second change involves the way listen() and notify() are being

used in the RobotWorld application. Read the comments embedded within the JiniListener class to find out more. *Difficulty: 2, Payoff: 3.*

Interconnectivity This is a set of issues I was grappling with even as I was checking the final version of the code into CVS. First, the two inner classes of PhotoSensor (PhotoSensor.Left and PhotoSensor.Right) cannot be instantiated from the Beanbag. It seems that the Class.forName() mechanism regards PhotoSensor as a package. A quick fix is to promote these two inner classes to full classes, LeftPhotoSensor and RightPhotoSensor, and then update layers.properties to list the full classes under MorphologyWindow rather than the inner classes. However, the need for these two classes will disappear when `Intuitive Graphics` is implemented. Next, CompositeControllers in the MorphologyWindow always present their pins inverted from the norm established in the BehaviorWindow and elsewhere: input pins on top, output pins on the bottom. See the screen shots for an example of this problem. Somehow the remoteFlag needs to be set within the constructor, but only if the CompositeController has been instantiated in a MorphologyWindow. The only remedy which I can see at the moment is a new subclass, RemoteController, which is listed in the layers.properties entry for MorphologyWindow rather than CompositeController. While we're at it, we may want to rewrite the isSource() and isDestination() methods of both CompositeController and RemoteController, so that RemoteController *only* creates pins for ActuatorPorts and SensorPorts, and a CompositeController *only* creates pins for SignalDestinations and SignalSources that are *not* SensorPorts and ActuatorPorts. In fact, it might be prudent to prevent SensorPorts and ActuatorPorts from being instantiated in a CompositeController. Otherwise users will not be able to distinguish between pins which are locally connectable and pins which are remotely connectable. Finally, upon analysis, the one-sided roles of SignalDestination and SignalSource unnecessarily restrict the Behavior Layer to two layers of recursion. This is not what was intended. CompositeControllers should be recursively nestable to any degree. The problem lies in the

Source/Destination duality: currently we have two objects, which are passive. They are externally connectable, but not internally connectable. Or rather, internal connections conflict with the external ones. (Take a look at the Behavior Editor screen shots. Connect the SignalSources to the SignalDestinations. Now follow the flow of data. It doesn't form a loop like one would expect. The reason is that these data transfer elements need to be both Sources and Destinations, but they present one face externally, and another internally). In terms of an implementation, we need a single object, a Port, which is animate. It reads from a single incoming connection, and outputs to all outgoing connections. The question is which face it presents externally (as a Pin), and which face it presents internally (as itself). Perhaps two subclasses of Port will do the trick. The benefit of writing Port is to draw together the commonality between SignalSource, SignalDestination, ActuatorPort, SensorPort, Actuator, and Sensor. They could all then subclass Port. These classes are all very similar, except that how they get their input, and how they deliver their output, varies. This can be captured by two methods of Port, `public Object getIncomingSample()` and `public void setOutgoingSample(Object o)`. Port should also be animate, with a while-true loop in its run() method. run() should call a method which does all the work and can be overridden, called act(). By default, act() just calls `setOutgoingSample(getIncomingSample())`. When this overhaul is complete, students will be able to interconnect whatever they see on the screen, with immediate results. This would make the interface much more intuitive and useful. *Difficulty: 3, Payoff: 3.*

Toys and Obstacles Currently the vehicular agents of RobotWorld can only react with each other and with the four walls of the playing field. Fortunately, the walls are rigid bodies in their own right (SeperatingPlane), rather than special-case code hard-wired into the CollisionDetector. The normal vector to the plane determines how other objects will react when colliding. A polygon or irregular wall could be composed of several SeperatingPlanes, which could allow

for a much more interesting playing field. Setting the inverseMass field to be nonzero (finite) would allow these objects to be pushed around the playing field. Also, programmatic sprite addition and deletion (see `Saving Your Work`) would allow interesting effects such as "consumption" of food pellets or "deposition" of pheromones. Wraparound—where sprites drive off one side of the playing field, to appear on the opposite side—can be implemented using this effect as well. Two opposing walls would need to collaborate such that if either was notified of a collision, it would move the colliding sprite to an appropriate spot tangent to its partner. Then, of course, there are the toys. These should come packaged with RobotWorld as a "kit" of parts which can be used e.g. in Section 10.1.2, when the student has not yet learned how to design their own. This I will leave up to the imagination of the reader. Re-read Chapter 2 if you need some ideas. *Difficulty: 3, Payoff: 3.*

Saving Your Work *This is a showstopper!* Currently, everything in RobotWorld is transient, with the exception of any files modified using the CodeEditorWindow. It allows users to load and save files, but it does not remind the user to save their work before a compile, or after the user closes the window. The bean bag allows one to add and remove beans, but the changes are lost when the application shuts down. If a student instantiates and interconnects a bunch of sprites in the visual GUI, they should be able to serialize this work to a file and be able to load it on a subsequent visit to RobotWorld. Currently, this work will be lost when the application shuts down. Also, students should have programmatic control over all tasks that can be performed in the visual GUI (at least adding and deleting sprites, and interconnecting them). Right now there are some critical bits of code in the canvas classes—which sprites don't really have access to. This will allow students to write a Java class corresponding to each of the four Braitenburg vehicles, which they can then add to the BeanBag. Now, for subsequent programming assignments, all they have to do is click on Love and instantiate it in the playing field! (The vehicle's constructor can

do all the component instantiation and wiring they previously did by hand.) On a related note, RobotWorld's blackboard implementation—JavaSpaces—has been set up to be transient, simply because it was the easiest. An equivalent to server.bat and multiuser.properties must be written to invoke the persistent version of JavaSpaces. Also, documentation must be added to the Users Manual regarding any differences in operation as a result of moving to the persistent version. *Difficulty: 3, Payoff: 3.*

Bean Analyzer/Bean Synthesizer : given a class, break it down by anatomical structure and report this to the user. Or, given fragments, synthesize an entire class file. See some of the SIGCSE literature [68, 21], they may give you some ideas. Also see the discussion of BeanAnalyzer in Chapter 5 and the gradual unveiling of class structure in Section 10.1 to see how this was meant to be used. *Difficulty: 3, Payoff: 3.*

Virtual Oscilloscope It would be beautiful to have a tool which can scribble the robot's state onto a canvas, generating time series or even phase portraits of the dynamically changing signals. See Section 10.2 for a detailed discussion of this tool. *Difficulty: 3, Payoff: 3.*

Info Bar It would be nice to have a small text window which prints the name and class of any component or pin that is clicked on (using getLabel() any other relevant information). This should be fixed in conjunction with `BeanBag`. That way you only have to touch SpriteWindow and SpriteCanvas. The Info Bar should be something small and nonintrusive at the bottom of the interface, kind of like the Emacs info bar that tells you the file you're editing, the line number, etc. *Difficulty: 3, Payoff: 3.*

Math Routines Can we have a consistent way of accessing (naming), updating, and operating on vectors scalars, and other mathematical objects? A hint from the pattern world: Iterator with Visitor, and perhaps Composite. The JDK 1.2 Collections framework may help here, as will any experience with lambda ex-

pressions in Scheme or Lisp. The problem is, scalar quantities are usually stored in fields, and are modified by using the + = assignment operator, while vector quantities are modified by calling methods such as add(). This will become a nightmare if the number of dimensions of the state vector changes. For instance, if someone wanted to extend RobotWorld vehicles to have arbitrary 2D morphology, inertia will need to upgrade from a scalar to a 2x2 matrix. In order to extend RobotWorld to 3D, orientation will have to upgrade from a scalar to a 3x3 matrix or a 4x1 quaternion. And so forth. If there is an abstract MathObject, and a way to compose a tree of those objects using abstract MathOperations, one could theoretically swap these implementations in and out using an AbstractFactory. *Difficulty: 3, Payoff: 3.*

Force Editing We need a dynamic way to insert, delete and modify ForceLaws at work in the playing field. Modification is easy once a `SpritePropertyEditor` has been implemented. It would probably be easiest to render ForceLaws as sprites, use your imagination. What does gravity look like, an apple? Keep in mind that there are some forces that should subclass ConnectorSprite, for instance Hook's Law for springs. So now the only question is, where is this force-editing area? Should it be imposed on top of the Habitat Editor, or the Morphology Editor? How do you distinguish the vehicles/sensors/actuators from the forces that act on them? Another solution is to split the window, or pop up a separate window, when the user selects an "Edit ForceLaws" option from the Workspace menu. The problem is, some forces like gravity are "invisible" and some, like Actuators or the hypothetical springs, are not. I dunno. Be creative. *Difficulty: 3, Payoff: 3.*

11.1.4 Flexibility

Empower Clients *This is a showstopper!* Client edits to the sprite tree are discarded without requesting the corresponding changes from the Server. (By the way, the edits die with the window. Open a different window, and you might see

the "same" sprite in a completely different state. Imagine seeing two completely different values of your bank account on your screen at the same time—one from before, and one from after making a transaction. This is exactly what is happening.) This is a bummer. The solution is to apply the Command pattern in conjunction with the second protocol suggested in Chapter 3. Until this is changed, only the person running the Server can make any "true" changes to anything! The code that has to be touched is currently in ServerCanvas, which serializes the sprite tree along with all sprite state, and ClientCanvas, which deserializes a copy, paints it, and then throws it away! This is bad, bad, naughty hacker. This fix may be best worked alongside `Optimize Network` alone, or as part of the larger pattern of changes described in `Model-View Separation`. *Difficulty: 3, Payoff: 4.*

BeanBag If every Sprite has an associated BeanBag of child classes, then it will be much easier to save, load and share ongoing work. A BeanBag is simply a list of classes which are permitted to be instantiated and added as children to a particular sprite. In terms of the interface, the BeanBag classes are enumerated in the list on the left of every RobotWorld window. The user selects a class, and clicks inside the window. The class is instantiated, and added as a child to the parent sprite which "owns" the window. The BeanBag should be serializable, so that they can be submitted to the blackboard and shared with other users. Whether a BeanBag is a list of filenames (Strings), a list of files (monolithic Strings) or a list of Classes is a security issue which should be given sufficient thought. Do we want to have students freely sharing code, or not? It probably depends on the classroom setting and collaboration policy. It may be prudent to consider groups and permissions, and the larger authentication and authorization issues. You may want to keep an eye on two newly minted Java APIs: JAAS and JSDT. JAAS is the Java Authentication and Authorization Service, which is a newly proposed extension to the security architecture. There is also JSDT, the Java Shared Data Toolkit, which is 100% pure Java

and can be downloaded as of the time of this writing. It provides classes for general purpose collaborative communication with support for sessions, groups, authentication, multicast, and more. Bon appetit! *Difficulty: 3, Payoff: 3.*

Rethinking IDE I'm referring to both CompileManager and EditManager, and the larger set of use cases including Extend This Class. The question is whether a more general framework is needed for controlling external applications besides editors and compilers. Also, the ExternalIDEManager should be examined for thread-safety. It would also be prudent to have it be self-animating, so that compiling doesn't have to occur in the GUI thread of the Java Virtual Machine (potentially bad). *NOTE:* The remainder of this Desired Feature is not necessary for the RobotWorld 1.0 distribution. It would also be great to have a custom virtual machine for RobotWorld that would allow you to auto-magically update all runtime instances when a class gets re-compiled. SuperCede offers this capability; I wonder how long it would take to implement? While we're on the subject, it would be nice to have a compiler written in 100for the purposes of interactive editing and compiling. It should interface smoothly with the class loader of the custom virtual machine. It should also have a spell checker that lets you know when you misspell class names (like "SeperatingPlane," which should be "SeparatingPlane"), field names, method names, variable names or even comments. But this is getting a little fancy. *Difficulty: 3, Payoff: 4.*

Multiple Sim Threads The sim package is littered with classes that ought to be animate but aren't. Some of these will need to implement DynamicalSystem in order to request service from a DiffeqSolver. This will in effect make for a multithreaded simulation. Currently you can tell that it's single-threaded when all agents are moving quickly, and they all pause simultanenously when the thread gets time-sliced. Search for the string "HACK" within the sim package. Also, act() should always be called by a sprite's own animate while-true loop. In the end, ParticleSystem's animacy could probably be revoked—if all of its particles, forceLaws, etc, were all animate. *Difficulty: 3, Payoff: 4.*

Model-View Separation Much nastiness and complexity arises from the fact that there is no model/view separation in RobotWorld. Classes simply extend one of the Sprite classes to obtain those properties. An alternate approach, offering flexibility and simplification of design, would be to have Sprites encapsulate the data model in a separate abstract class or interface. There are some hints in George Reese's book on JDBC [63]. He offers a persistence framework with use of the Observer interface to keep data current. There is also an article on the Swing Connection[26], which describes the Model/View pattern in detail. This would then allow all sprites to be views, responsible only for rendering state (data models) that have been read off the blackboard. It could be solved simultaneously with `Optimize Network`, `Math Routines`, and `Multiple Sim Threads`. Why? Assume the data model is a MathObject that satisfies the `Math Routines` criteria. Then sprites are simply the consumers of this model, and the DynamicalSystems are the producers. The high-bandwidth and relatively constant sprite tree data models have become decoupled from the low-bandwidth ever-changing sprite state data models, resulting in less bandwidth consumption. This solves `Optimize Network`. On the producer side, each particle is free to become a self-animating DynamicalSystem, satisfying `Multiple Sim Threads`. After all this is up and running, it will pave the way for `Empower Clients` and `Architecture Layer`. *Difficulty: 4, Payoff: 5.*

Chapter 12

Conclusions

The RobotWorld system allows students to come to understand how an ensemble of objects can interact to accomplish a desired goal. Whether a student merely uses the GUI to design some Braitenburg-type vehicles, extends some of the controller code to implement a wall-following, obstacle-avoiding agent, or extends the RobotWorld system with a new Blackboard implementation or a virtual oscilloscope, students will have had interactive, hands-on experience in a concrete, accessible programming environment. By centralizing the theme of a situated agent or ensemble of agents, a wide range of concepts in computer science at the freshman and even upperclass level can be made more concrete, more compelling, and more intuitive. If one takes the constructionist approach, agents can serve the crucial role as a bridge between the student's previous experiences and cognitive structures and the possibly abstract concepts in computer science.

The RobotWorld system, then, provides Ben-Ari's accessible ontological reality [9]. It is then up to the instructor to present models and metaphors which illuminate the concepts of computer science within the context of the laboratory, where the student's hypotheses, experiments, and analyses will, with the instructor's help, tease out any misconceptions and point the way towards more viable models. As a result of this process, misconceptions will be caught early, and students will be more committed and will be able to learn, retain, and explain the core concepts more effectively, improving

their ability to plan, coordinate, and critique software designs in an academic or professional setting. In summary, RobotWorld provides a vehicle for independent discovery, which, when combined with some helpful direction from an instructor and one's peers, can head the student of computer science down a path towards academic and professional success.

Appendix A

Key to UML Notation

Figure A-1 shows the subset of the Unified Modeling Language (UML) used in this thesis. UML provides a standardized graphical notation for object-oriented analysis and design. After *UML Distilled* [28].

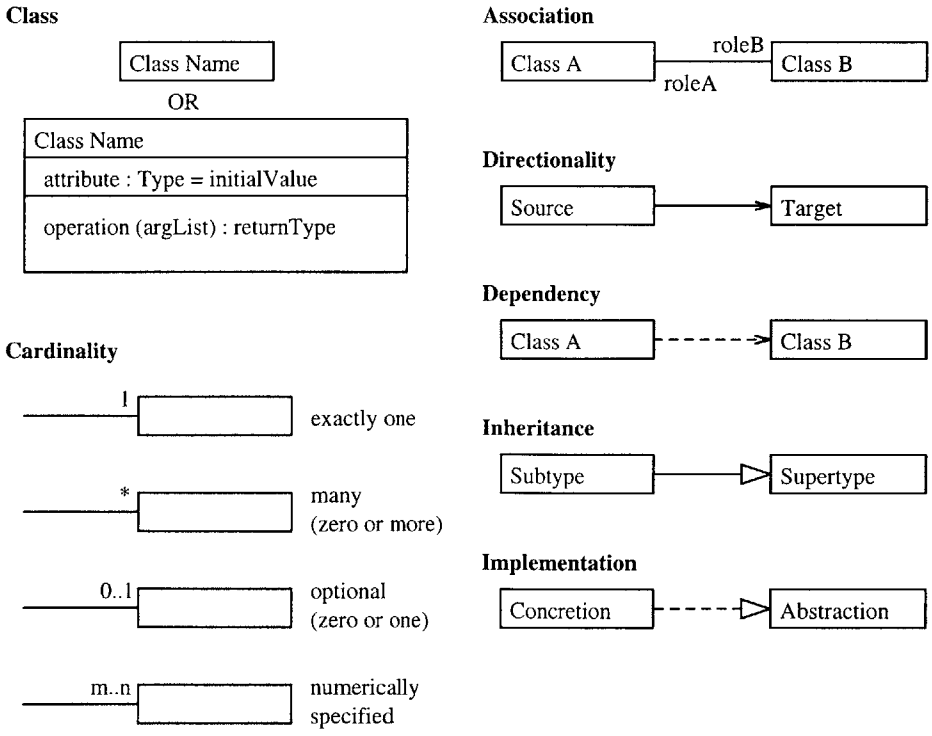


Figure A-1: Key to UML notation.

Appendix B

User's Manual

The purpose of this document is to explain how to install, configure, and run the RobotWorld simulation environment. It also explains how to add your own extensions to the framework. Different sections of this manual assume different needs, and different levels of exposure to the Java technology:

- **A student** will be most interested in What is it? (B.2), Starting Up (B.5), Using the Workspace (B.6), and The Five Layers of Observation (B.7).
- **An instructor** will be potentially interested in all sections of this manual. What is it? (B.2) should give you an idea of the purpose of RobotWorld. Installing (B.3) and Custom Configuration (B.4) show you how to set up RobotWorld for your system. Future Directions (B.8) indicates some of the improvements we'd like to make to RobotWorld, most notably Saving Your Work (B.8.3).
- **A developer** who would like to add her own extensions to the application framework should go to Extending RobotWorld (B.8.2). This involves programming to the RobotWorld API in order to extend or replace the components which constitute the RobotWorld application. For example, you might want to add a new layer of observation, or swap out the current blackboard implementation with your own.

B.1 Table Of Contents

(a) JDK Configuration (B.4.1)

(b) NT Emacs Configuration (B.4.2)

4. Starting Up (B.5)

   (a) Singleuser Startup (B.5.1)

   (b) Multiuser Startup (B.5.2)

5. Using the Workspace (B.6)

   (a) The Beanbag (B.6.1)

   (b) The Canvas (B.6.2)

   (c) Sprites and Layers (B.6.3)

   (d) The Menu Bar (B.6.4)

6. The Five Layers of Observation (B.7)

   (a) The Architecture Layer (B.7.1)

   (b) The Habitat Layer (B.7.2)

   (c) The Morphology Layer (B.7.3)

   (d) The Behavior Layer (B.7.4)

   (e) The Code Editing Layer (B.7.5)

7. Future Directions (B.8)

   (a) Example Problem Sets (B.8.1)

   (b) Extending RobotWorld (B.8.2)

   (c) Bugs and Features (B.8.3)

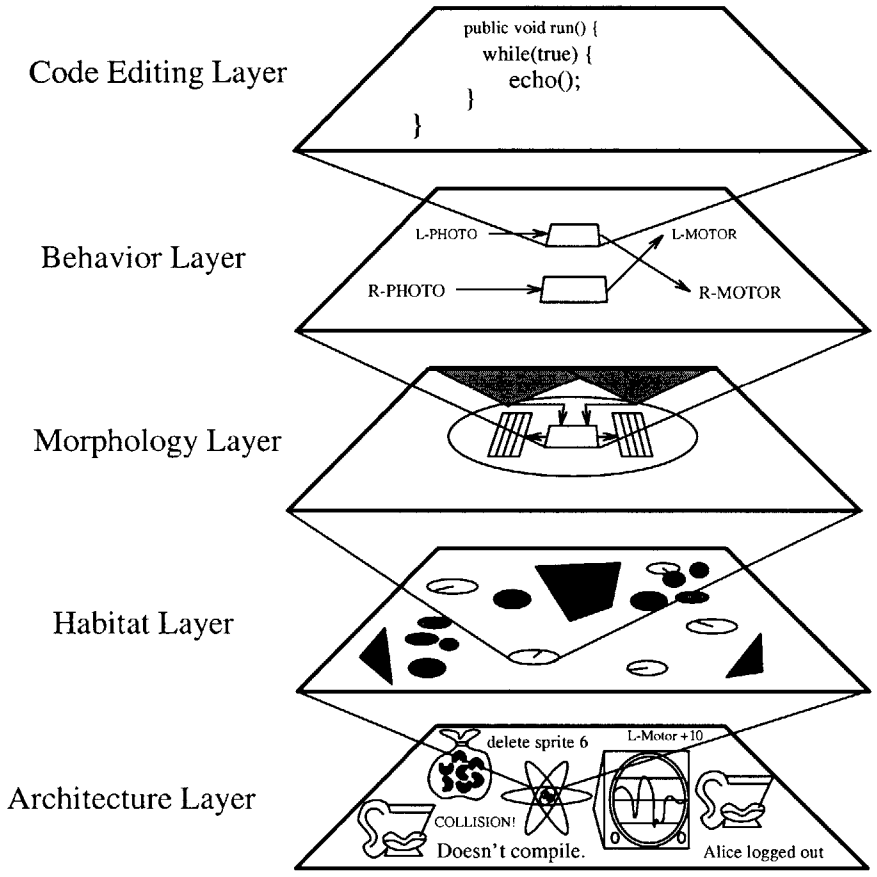
   (d) Finding Out More (B.8.4)

B.2 What is it?

RobotWorld is a robot simulation environment for introductory computer science education. The goal is to establish a link between computation and behavior by providing an interactive learning laboratory, in which students can:

- wire up a robot brain
- program a community of agents
- design interactive displays
- experiment with the network
- replace or extend system components

RobotWorld is a Java application that uses a distributed, asynchronous communication system called a blackboard to form a collaborative workspace. Students can control their dynamically simulated robots by interconnecting behavior nodes in a visual GUI, or they can edit the Java classes interactively. Here's how everything fits together. (Click for a larger image.)



B.3 Installing

There are two options for installation. The Basic Installation provides the shortest path to starting up the application, but is restricted to single-user operation. This might be appropriate for a quick evaluation before adding the packages for networked, multiuser operation.

B.3.1 Basic Install

Download the robotworld basic distribution and unzip it into the directory of your choice. Let's assume this is the directory `/rw/`. These are the files you should see:

- `about.html`: About RobotWorld.
- `manual.html`: This User's Manual.
- `makefile`: allows you to compile part or all of RobotWorld from the source code.
- `layers.properties`: sets up the five layers of observation.
- `singleuser.properties`: sets up a singleuser implementation.
- `jdk.properties`: sets up RobotWorld to use a pure Java editor and the JDK for compiling. Meant to be used with JDK Configuration (B.4.1).
- `ntemacs.properties`: sets up RobotWorld to hook into NT-emacs for editing and compiling. Meant to be used with NT Emacs Configuration (B.4.2).
- `.emacs`. Optional Emacs initialization file. Meant to be used with NT Emacs Configuration (B.4.2).
- `javadoc/`: the RobotWorld API Documentation.
- `robotworld/`: the RobotWorld source and classes.

If the Basic Install is all you require, you will now need to perform a Custom Configuration (B.4) so that RobotWorld can find a Java compiler on your system.

B.3.2 Multiuser Install

The Multiuser Install is an extension to the Basic Install that allows many users to interact across the network. Here's what you have to do:

- Perform the Basic Install first.
- Log in to Sun's Java Developer Connection (`http://developer.java.sun.com/developer/`) (requires free registration).
- Visit the Jini Download web site (`http://developer.java.sun.com/developer/products/jini/`), and click on Product Offerings. Download and install the Jini Starter Kit. (You won't be needing the TCK, which is a compatibility kit.) Let's assume you installed it in the directory `/jini1_0/`.
- Download and install the Evaluation Binaries for JSTK (JavaSpaces) into the same directory.
- Download the robotworld multiuser distribution and unzip it into your RobotWorld directory, `/rw/`. These are the new files you should see:
  - `multiuser.properties`: sets up a multiuser implementation.
  - `server.bat`: fires up the three required server processes for the multiuser implementation. For reference, there are three lines in the file, one for each process. They are the RMI Registry (the rmiregistry line), the HTTP server (the tools.jar line), and the JavaSpaces server (the transient-outrigger.jar line).

  In addition, you will see new source code, classes, and documentation all corresponding to the package `robotworld.net.jini`, which provides a Jini/JavaSpaces implementation of a blackboard for RobotWorld.

- You will need to edit `multiuser.properties` and `server.bat` to bring RobotWorld into harmony with Jini/JavaSpaces. The changes will be listed by option:
  - `/jini1_0/`: wherever you see this in `server.bat` or `multiuser.properties`, change it to the full path your own Jini install directory.
  - `java.rmi.server.codebase`: the URL following this needs to contain the Internet address of the machine that will act as the RMI Server (the one that will run `server.bat`). Both `server.bat` and `multiuser.properties` must be edited to reflect this Internet address. The port should remain 8080, as that is the default provided by tools.jar, which supplies the HTTP server. Note that everything below the `/jini1_0/lib/` directory will be web-visible while the HTTP server is running.
  - `com.sun.jini.outrigger.spaceName`: an identifier for the JavaSpaces instance is provided. If you wish to change it, the spaceName can be anything, but it must be exactly the same in both `server.bat` and `multiuser.properties`.
  - `java.security.policy`: for ease of installation, a wide-open security policy file is made available with the Jini distribution. If this is not acceptable, write your own and modify both `server.bat` and `multiuser.properties` to point to your file instead.
  - If security is an issue, there are many other possible improvements, such as: changing the RMI registry port from 1099 (the rmiregistry default), changing the HTTP port from 8080 (the tools.jar default), and disabling the tools.jar process to use a more discriminating HTTP server.
  - If and when RobotWorld switches to the persistent version of JavaSpaces, `server.bat` and `multiuser.properties` will both need to be updated.

You will now need to perform a Custom Configuration (B.4) so that RobotWorld can find a Java compiler on your system.

B.4 Custom Configuration

RobotWorld was designed to be highly configurable. This section describes how to customize RobotWorld to hook into your favorite text editor, compiler, or integrated development environment (IDE). A later section (Extending RobotWorld (B.8.2)) describes how to program to the RobotWorld API and replace the current implementation classes with your own.

B.4.1 JDK Configuration

This is the simplest possible configuration. RobotWorld comes with a built-in code editor, but you will need to tell it where to find a Java compiler.

- If you don't already have it on your system, download and install the Java 2 SDK Standard Edition (`http://java.sun.com/products/jdk/1.2/`). Let's assume you have installed it in the directory `/jdk1.2/`.

- Note that you no longer have to set the `CLASSPATH` environment variable.
- You should, however, set the `PATH` environment variable to include the `/jdk1.2/bin/` directory.
- *Optional:*If you do not wish to do change your `PATH`, you will need to edit the `jdk.properties` file that came with RobotWorld, and change the line:

```
robotworld.ide.ExternalIDEManager.COMPILE   = javac {0}
```

  to the following:

```
robotworld.ide.ExternalIDEManager.COMPILE   = /jdk1.2/bin/javac {0}
```

- Go ahead and start up RobotWorld (B.5). Now, and whenever you start up RobotWorld, specify `jdk` as one of its command-line arguments. This causes the `jdk.properties` file to be loaded, so that RobotWorld knows what to do.
- *Optional:* Keep in mind that if you don't like the supplied code editor, you can hook into an external editor like notepad. Let's assume that you're on a Windows machine, and you've decided to use notepad instead. Then you can simply edit the `jdk.properties` file that came with RobotWorld, and uncomment the line:

```
# robotworld.ide.ExternalIDEManager.EDIT       = notepad {0}
```

  If notepad isn't on the `PATH`, either add it, or edit the line above to have the full directory path to the notepad executable. This technique can be used to invoke any command-line accessible executable in order to edit or compile a particular file. Note that the `{0}` will be replaced by the filename to edit or compile.

B.4.2 NT Emacs Configuration

This requires quite a bit of installation, but if you're a diehard Emacs fan in a Windows NT world, it'll be all worth it.

- Perform all the steps in the JDK Configuration, except that wherever `jdk.properties` is specified, use `ntemacs.properties` instead.
- Download and install Emacs for Windows NT/95/98 (`http://www.cs.washington.edu/homes/voelker/ntemacs.html`). Let's assume you installed it in the directory `/ntemacs/`. Be sure to add `/ntemacs/bin/` to your `PATH`. Otherwise setting up gnuserv (see below) will be more difficult.
- Download and install Cygwin Bash or some other shell (`http://www.cs.washington.edu/homes/voelker/ntemacs.html#shell-bash`), so that executables can be invoked from within Emacs. This allows for the use of makefiles as well as debugging features such as jump-to-error. Jump-to-error allows one to click on a compiler error to jump to the offending line in the source file.
- Download and install gnuserv/gnuclient (`http://www.cs.washington.edu/homes/voelker/ntemacs.html#assoc`), which allows Emacs to be used as a server. This means that you can pass commands to a running version of Emacs to open and compile files. The `readme.nt` installation instructions are somewhat cryptic for the un-initiated, hopefully this helps. Let's assume that gnuserv was installed in directory `/gnuserv/`. Then the emacs `load-path` variable must be pointed towards `/gnuserv/`. You'll have to consult the emacs Help menu: **Help > Describe > Apropos Variables**. Then type `load-path`, and hit return. At the bottom

of the buffer is the word **Variable**. Single-click on it, and hit return. A buffer will open. At the bottom of it is the word **customize**. Single-click on it, and hit return. You will see a list of directory names, go to the bottom of the list and click on **[INS]**. Click on **[Current dir?]** and select `Directory`. Now type `/gnuserv/`, your gnuserv directory name, into the text area. Be sure to click on **[State]** and select `Save for future sessions`. You must now load the file `gnuserv.el` into Emacs, and byte-compile it. Do this by selecting the Byte-Compile option under the Tools Menu. The resulting .elc file should be placed in `/gnuserv/`. Finally, add `/gnuserv/` to the `PATH`, or edit `ntemacs.properties` to include the full path to the executables `gnuclient.exe` and `gnudoit.exe`. Whew! Did you break a sweat?

- Be sure to examine the `.emacs` initialization file provided with RobotWorld. If you already have an `.emacs` file, you may want to copy the relevant bits of code to interface with Bash and Gnuserv. There are also options for syntax coloring (font-lock), highlighted brace matching, and other conveniences. If you don't have an `.emacs` file, you're welcome to use the provided one. You will probably want to scour the NT Emacs pages for tips and tricks on printing, CVS support, and other useful additions that are too numerous to mention.
- Go ahead and start up RobotWorld (B.5). Now, and whenever you start up RobotWorld, specify `ntemacs` as one of its command-line arguments. This causes the `ntemacs.properties` file to be loaded, so that RobotWorld knows what to do.

B.5 Starting Up

B.5.1 Singleuser Startup

To start up RobotWorld in singleuser mode, enter the following at the command prompt:

```
java robotworld.Server singleuser layers [ide_option]
```

where [ide_option] is either `jdk`, `ntemacs`, or an option provided by the instructor.

B.5.2 Multiuser Startup

This option is not available unless a Multiuser Install has been performed. To start up RobotWorld in multiuser mode, enter the following at the command prompt:

```
server.bat
```

This will launch three processes: one to set up an RMI registry, one to set up a simple HTTP server for class portability, and one to set up a JavaSpaces server. If the Multiuser Install went smoothly, you don't need to understand what these do. However, if you forget to launch the server processes, the next part won't work. Now enter this at the command prompt:

```
java robotworld.Server multiuser layers [ide_option]
```

where [ide_option] is either `jdk`, `ntemacs`, or an option provided by the instructor.

Now that a RobotWorld server is running, any number of client machines can start up a RobotWorld client. It is assumed that a Multiuser Install has been performed on these clients, and their `multiuser.properties` files are identical to that of the server's. Then, all a client has to do is type the following at the command prompt:

```
client.bat
```

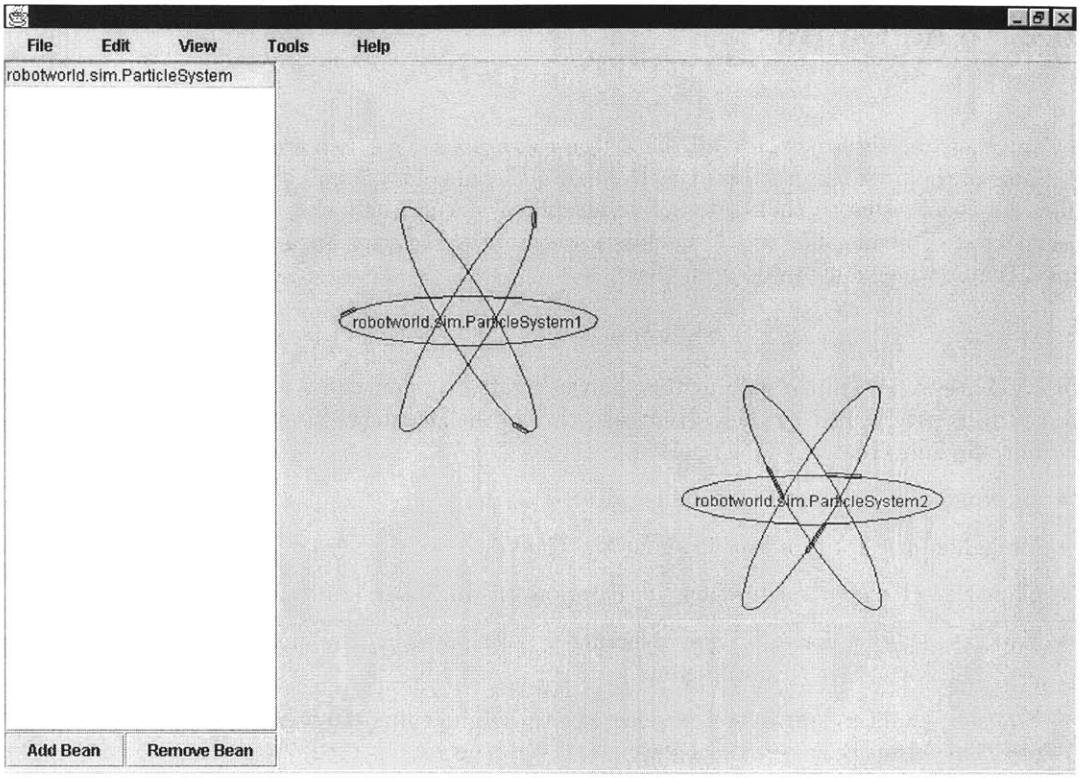
This will launch the RMI registry, which will allow the client to locate the server (this action is superfluous when the client and server are being run on the same machine). Now the client application can be started up by typing the following:

```
java robotworld.Client multiuser layers [ide_option]
```

where [ide_option] is either `jdk`, `ntemacs`, or an option provided by the instructor.

B.6 Using the Workspace

When you start up RobotWorld, a window similar to this one will appear. There are two major panels, one on the left (the Beanbag), and one on the right (the Canvas).



B.6.1 The Beanbag

The Beanbag, on the left, controls what can appear in the Canvas. The names listed in the Beanbag are Java classes (beans). Each one has different kinds of properties and behaviors.

- You can select the class of bean you'd like to create by single clicking on one of the choices in the Beanbag. It should now be highlighted.
- Click on an empty spot in the Canvas to create (instantiate) the bean.
- You can add a new class of bean to the Beanbag by clicking on the button marked **Add Bean**. You will be prompted for the bean's name. You won't be needing this option until you have written your own Java class, and want use it within RobotWorld (see Chapter 10 of this thesis).
- You can remove the currently selected bean from the Beanbag by clicking on the button marked **Remove Bean**. You might want to use this to remove beans that you added earlier, but don't need anymore. *Beware!* Don't remove the beans supplied with RobotWorld, unless you know what you're doing! Otherwise, you'll have to restart RobotWorld to get them back.

B.6.2 The Canvas

The Canvas on the right is where all the action takes place. You can create many kinds of objects here. Some of them will interact with each other, and some of them will only react when you directly manipulate them. Due to their interactive, graphical nature, let's call them *sprites*. They're just a special kind of bean (they're all subclasses of `robotworld.gui.Sprite`, if you know what that means). Here's what you can do with them:

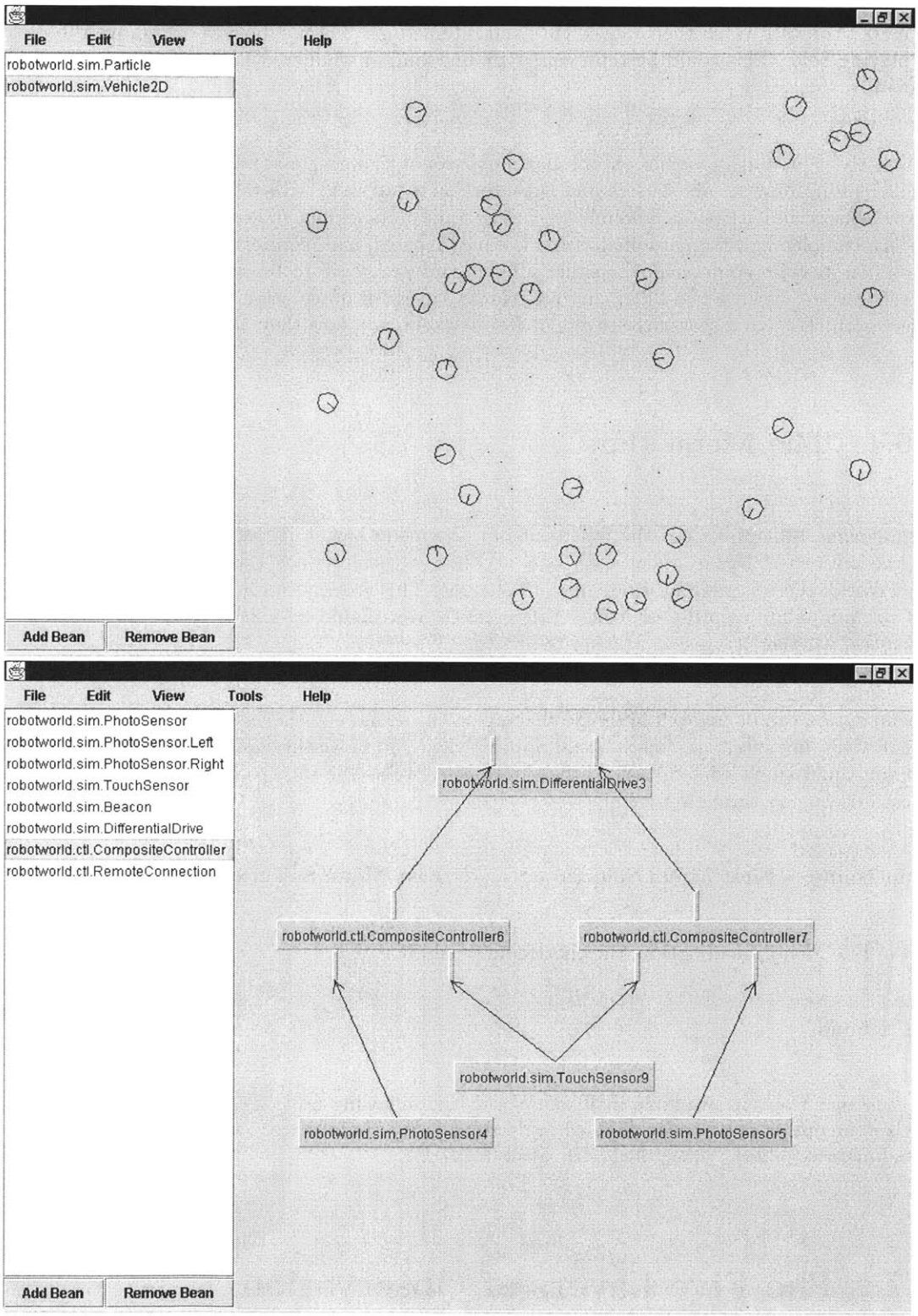
- Clicking on an empty spot in the Canvas creates (instantiates) a sprite there. The kind of sprite is determined by the current selection in the Beanbag. If nothing is currently selected, nothing gets created.
- Clicking once on a sprite selects (or deselects) it.
- Dragging on a sprite moves it about the Canvas.
- Clicking twice on a sprite opens it (more on this in Sprites and Layers (B.6.3).)
- You can delete a sprite by first selecting it, then invoking the menu option **Edit** > **Delete** (see The Menu Bar (B.6.4), below).
- You can rename a sprite by first selecting it, then invoking the menu option **File** > **Rename** (see The Menu Bar (B.6.4), below).

Keep in mind that some of the windows override this behavior so that you can do more complicated things, like connect one sprite to another one. The section called The Five Layers of Observation (B.7) describes exactly which windows act this way, and when they switch from the normal behavior to a more specialized one.

B.6.3 Sprites and Layers

If you've ever used a visual editor such as Visual Basic before, you should find the RobotWorld interface somewhat similar. You select the kind of thing you want from a menu, and you create it in a nearby space. If you're interested in that object's properties, you double click on it, and a window pops up. In Visual Basic, this might contain a list of properties such as the size and color of the object, which the user can alter using sliders and menu buttons.

Here's where RobotWorld is different: when you double click on a object, a new window pops up that *decomposes* the original object into another window full of sprites—its component parts. In other words, sprites can be recursively nested, just like Russian dolls. In the original window, you can see the sprite from the outside—as a whole. In the new window, you can see the sprite from the inside—as many interacting parts. The two diagrams below should help make this clear. On the top is a playing field full of robots—seen as wholes. On the bottom is a close-up of one particular robot—seen as many parts. Keep in mind that the other robots might have completely different looking close-ups. *Note to the instructor: Each window represents a particular* instance *of a sprite object. There is currently no way to represent a Java* class *in RobotWorld, except as a name displayed in the Beanbag.*



The new window that pops up is not always the same kind as the old one. This is because when you zoom in on a sprite's internals, the rules of interaction might be completely different. The details that emerge constitute a new *layer of observation*. For example, imagine a playing field full

of robots. You can watch their movements and interactions, and maybe add or remove robots from the playing field. This could be represented by one kind of window, where the beans are all kinds of robots.

Alternately, you could zoom in on one particular robot, ignoring everything else, and monitor the signals passing between its sensors and actuators as it interacts with the world. You could add or remove sensors and actuators, and maybe change some of its wiring, and see how its reacts differently. This is a completely different domain from the playing field, and requires a new kind of window that knows how to wire components together. The beans are going to be sensors, and actuators, and wires. You are viewing two different, but related, layers of observation. Visit The Five Layers of Observation (B.7) to discover the layers of RobotWorld, and how they all interrelate.

B.6.4 The Menu Bar

The following table describes the five menus on the menu bar: File, Edit, Tools, View, and Help, and the options available under each one. Although the interface is similar for all windows in RobotWorld, there is a slight difference in behavior. This is because the same menu option can be used to manipulate a sprite, or a file. For instance, you should be able to print the contents of a workspace, whether it consists of a sprite or a file.

A menu option can be invoked by single clicking on the menu, and then single clicking on the option. If there were any submenus, they would appear when you move the mouse pointer over them, and then you could single click on any of their options as well. The shorthand notation will be used to indicate traversing through a menu:

Menu Name > First Menu Selection > ... > Last Menu Selection

For the File Menu, Load option, the shorthand would be:

File > Load

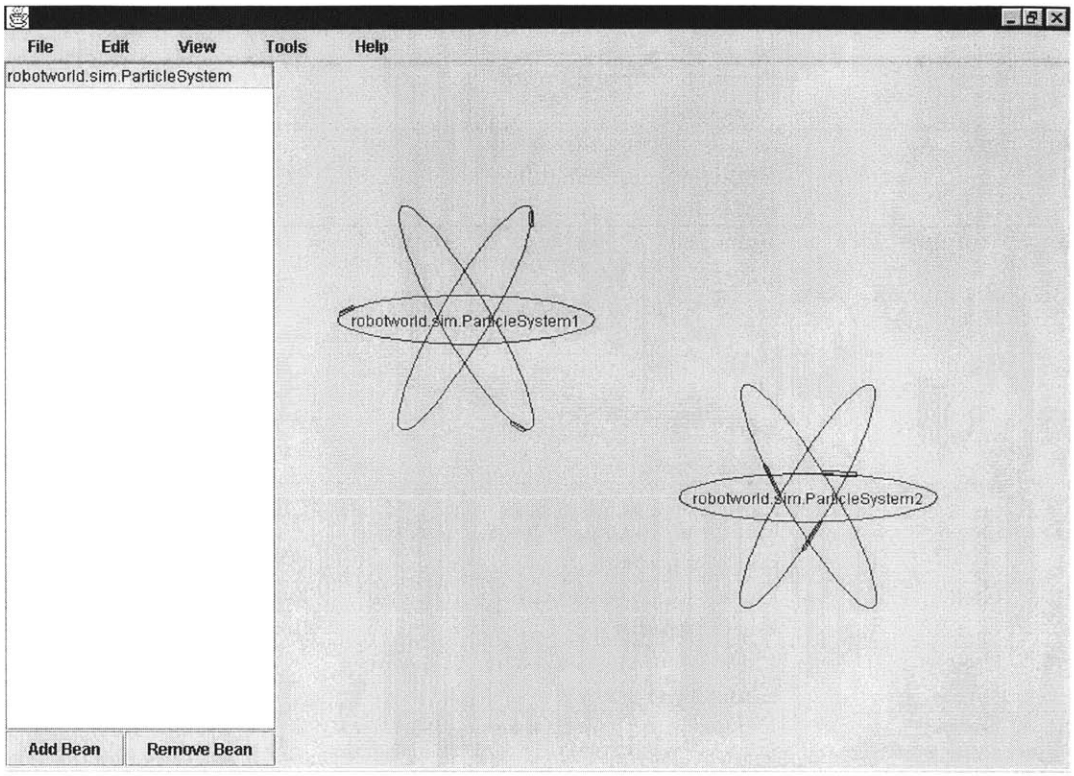
For each menu option available in RobotWorld, the following table lists a short description, and whether the option is currently enabled or disabled in the Code Editor (which deals with text files), and all other windows (which deal with sprite objects).

B.7 The Five Layers of Observation

This section assumes you have just completed Starting Up (B.5), using either the singleuser or multiuser version. If you are running the singleuser version, you can ignore the description of the events happening in the Client, because you are running the Server.

Menu Option	Description	Action on File	Action on Sprite
File > Clear	Clears the workspace	Enabled	*Disabled*
File > Load	Loads content into the workspace	Enabled	*Disabled*
File > Save	Saves the contents of the workspace	Enabled	*Disabled*
File > Save As	Saves the contents of the workspace under a different name.	Enabled	*Disabled*
File > Rename	Renames a workspace item.	*Disabled*	Prompts for a new name for the currently selected sprite
File > Print	Prints the contents of the workspace.	*Disabled*	*Disabled*
File > Exit	Exits the RobotWorld application.	Enabled	Enabled
Edit > Copy	Copies a workspace item to the clipboard.	*Disabled*	*Disabled*
Edit > Cut	Cuts a workspace item to the clipboard.	*Disabled*	*Disabled*
Edit > Paste	Pastes a workspace item from the clipboard.	*Disabled*	*Disabled*
Edit > Delete	Deletes a workspace item.	*Disabled*	Deletes the currently selected sprite.
View > Code	Pops up all Code Editor Windows.	*Disabled*	*Disabled*
View > Behavior	Pops up all Behavior Windows.	*Disabled*	*Disabled*
View > Morphology	Pops up all Morphology Windows.	*Disabled*	*Disabled*
View > Environment	Pops up all Habitat Windows.	*Disabled*	*Disabled*
View > Architecture	Pops up all Architecture Windows.	*Disabled*	*Disabled*
Tools > Beanbag	Organize, load, or save the user's beanbags.	*Disabled*	*Disabled*
Tools > Compile	Compiles the currently selected file.	Enabled	*Disabled*
Help > About_RobotWorld	Find out more about RobotWorld.	Enabled	Enabled
Help > Java_Reference	Browse the JDK 1.2 API.	Enabled	Enabled
Help > Textbook	Browse an online CS101 textbook.	Enabled	Enabled
Help > Users_Manual	Browse this User's Manual.	Enabled	Enabled

B.7.1 The Architecture Layer



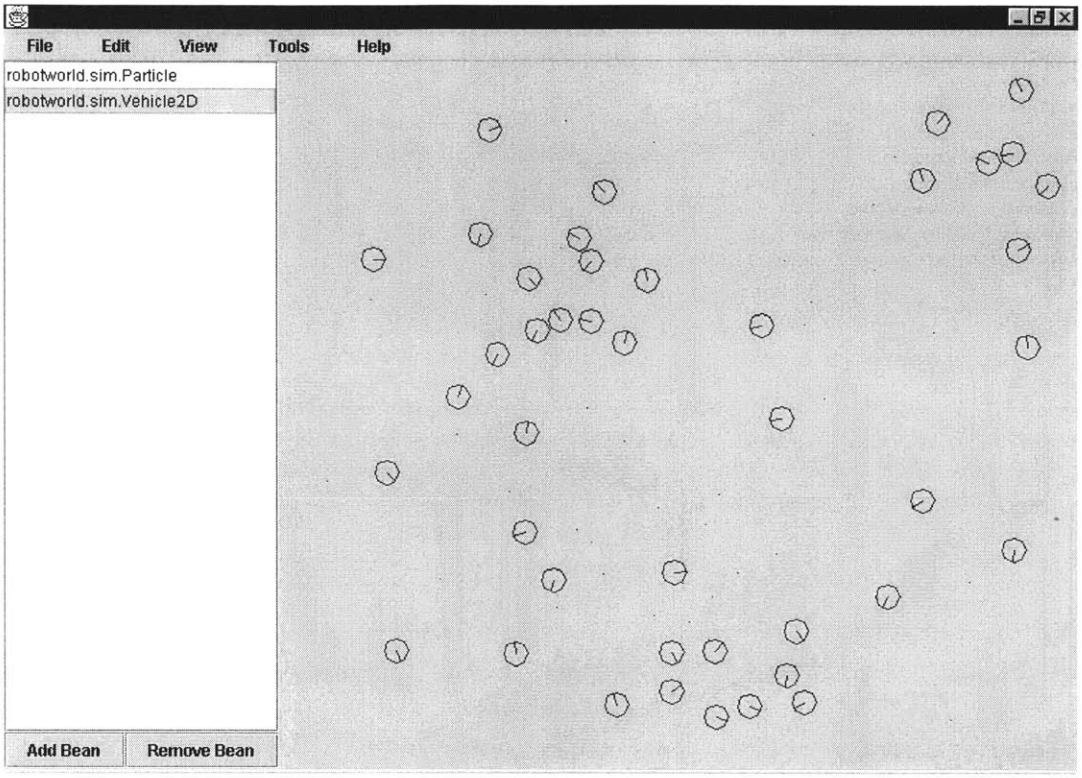
We arrived at this Server Window when RobotWorld started up. The Client Window is identical except that the Beanbag is empty.

Whenever you start up RobotWorld, you are in the Architecture layer. This conceptually covers both the Client Window and the Server Window–they both paint the same picture. However, the Client's beanbag is empty, reflecting the fact that it cannot create a session, it can only browse existing ones.

The user running the server should select `robotworld.sim.ParticleSystem`, the one and only bean in their Beanbag, and create one by single-clicking in the Canvas. A large animated atomic symbol will appear in the server's Canvas. Shortly thereafter, it will also appear in the Client's Canvas. The user at the server end should drag the ParticleSystem around the screen. You will notice that all changes made at the server end will be reflected at the Client end. (Currently the Client can only browse; all of their changes are discarded. This should be fixed in a future release.) This is the purpose of the Architecture layer: to coordinate the activities of client and server. Note that since objects can be recursively nested, everything inside an object placed in this layer will also be available to others. In the future, there may be other objects you may wish to place here, at the root, for coordination. For now, we will keep things simple.

The job of the ParticleSystem is to ensure that all of the sprites it contains move as if they were subject to Newton's laws of motion. Double-click on it to see it in action (both the client and server can do this). A new window should pop up, bringing you to...

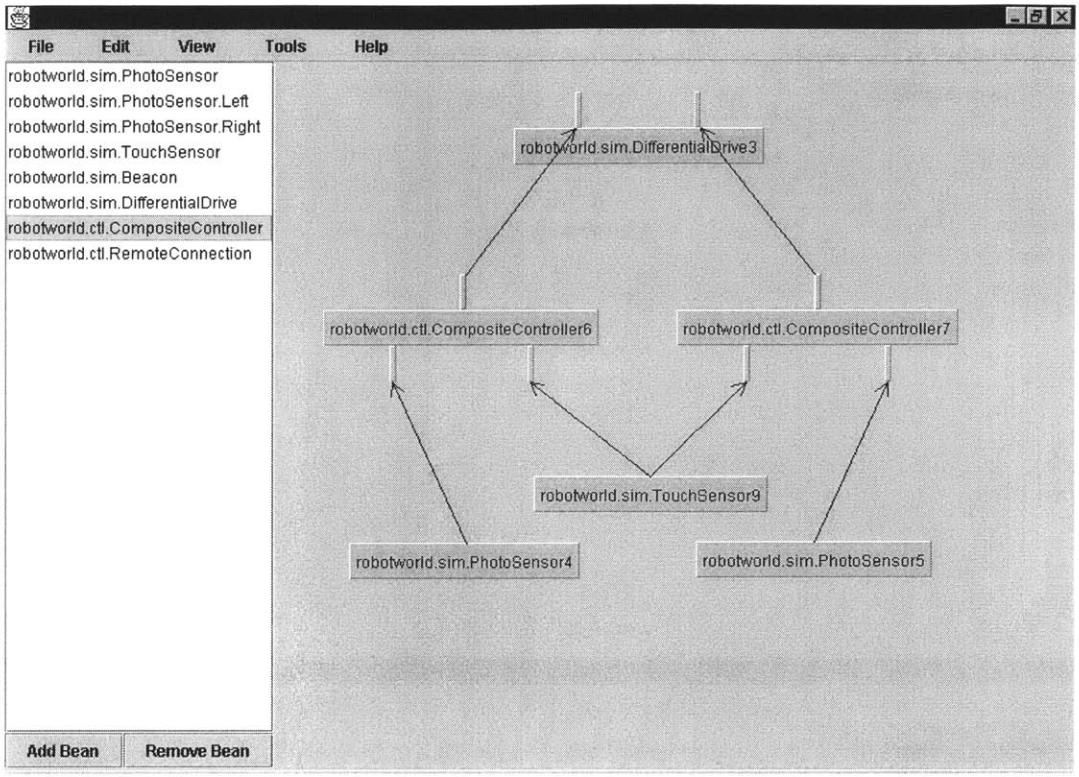
B.7.2 The Habitat Layer



We arrived at this Habitat Window by double-clicking on a ParticleSystem in the Architecture Layer.

This area is a two-dimensional playing field where your robots can drive around and interact. You can think of yourself as looking down on the scene from above. There are two beans available here: `robotworld.sim.Particle` and `robotworld.sim.Vehicle2D`. Create a few of each in the canvas. Notice that the thin line inside the Vehicle2D indicates its heading. If it could drive on its own, this is the direction it would go. However, these are no more than billiard balls at the moment, subject to the whims of your mouse pointer... Try dragging them around with the mouse. The particles have an invisible radius of mouse sensitivity; you don't need to be exact. Now try dragging a Vehicle2D such that it bumps into several other vehicles. They should get pushed around in a reasonable way, although the collision routines aren't perfect. You get the general idea. Now, let's add some reactivity to these vehicles. Double-click on a Vehicle2D to enter...

B.7.3 The Morphology Layer



We arrived at this Morphology Window by double-clicking on a Vehicle2D in the Habitat Layer.

The name refers to the geometric form and structure of the vehicle, including the placement of sensors and actuators. These factors will determine to a large degree what your vehicle is capable of doing. A quick note: this layer could benefit from a layout which mimics the vehicle's "real" morphology, but for now, abstract labeled boxes will have to do. Because of this, the explanation that follows will be rather detailed.

Notice that many more beans have been supplied here:

- `robotworld.sim.PhotoSensor`, which measures the intensity of any emitted radiation in the playing field. It is most sensitive to light sources in front of it and aligned with its central axis, and less so to light sources at an angle. It has almost no response to light sources behind it. By default, it is placed on the nose of the vehicle, with the same heading. Its cone of sensitivity is roughly 45 degrees to either side.
- `robotworld.sim.PhotoSensor.Left`, which is identical to the plain PhotoSensor, but is placed differently. By default, it is placed on the exterior of the vehicle, 45 degrees to the left of the vehicle's nose. Its central axis is aligned to be parallel with the vehicle's heading. The cone of sensitivity is the same.

- `robotworld.sim.PhotoSensor.Right` is a mirror image of PhotoSensor.Left.
- `robotworld.sim.TouchSensor` models a unidirectional bump skirt: when the robot contacts an object at any point along its exterior, the resulting force on the vehicle will be measured.
- `robotworld.sim.Beacon` is an actuator. It can accept commands to glow at any given intensity. You will need at least one Beacon on some vehicle in the playing field before the PhotoSensors will pick up anything. (The playing field has a small but constant ambient light level.) By default, it just glows at a fixed intensity.
- `robotworld.sim.DifferentialDrive` is a pair of motors that can run independently. The DifferentialDrive sprite presents two pins—one goes to each motor. The left pin goes to the left motor, and the right pin goes to the right motor. Each motor is connected to a wheel, such that when both motors turn forwards, the vehicle moves forwards. When both motors turn in reverse, the vehicle backs up. If the motors turn in the same direction but at different speeds, the vehicle will drive in an arc. If one motor turns forwards while the other turns in reverse, the vehicle will spin in place. The inputs to the motor pins are the rates at which the motors (and hence the wheels) should turn. They will try to reach these values, but they are limited as a real system might be so that there is a maximum driving velocity.
- `robotworld.ctl.RemoteConnection` is a special sprite. It allows you to interconnect sensors, actuators, and other components. *Important: when you select this bean in the bean bag, the standard user interface is disabled.* So long as this bean remains selected, the window is in *Special Mode* and this following user interface applies:
  - Clicking on a remote source (like a sensor), and then a remote destination (like an actuator), creates a RemoteConnection between the two. It appears as a directed arrow from the source to the destination. *Rule:* A source can have many destinations, but a destination can have only one source.
  - Clicking on a RemoteConnection selects (or deselects it).

  This means that RemoteConnections and only RemoteConnections can be created, selected, deleted, and opened in this particular mode. The "connection" part should make intuitive sense, you want to be able to connect different parts of the vehicle together. But what does the "remote" part refer to?

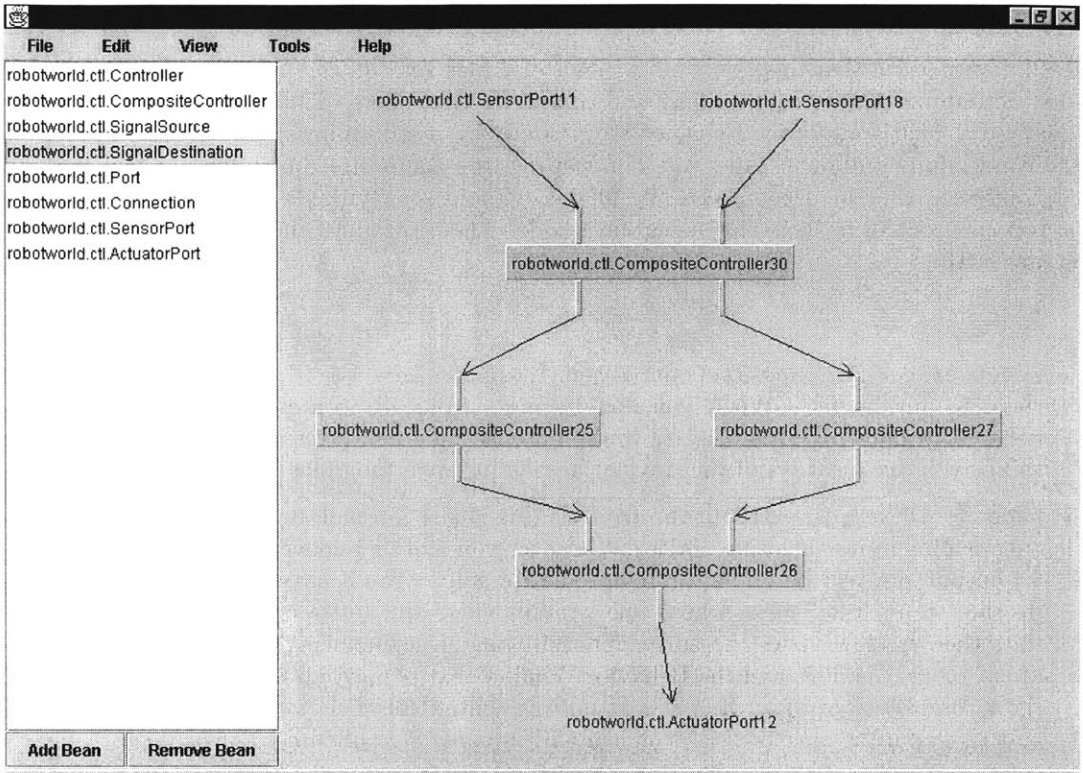
  This layer has been specially designed to bridge multiple machines on the Internet. What this means is that the ParticleSystem, with all of its vehicles and all of their sensors and actuators, can be running on one machine. Let's consider this system of entities to provide a service, which simulates Newtonian motion. In this sense, it is a Server. Then, a Client of this service, on a different machine, can visit this layer and create an object that encapsulates wall-following behavior. This object would stay on the Client machine, but its would be able to communicate with all the sensor and actuator objects on the Server by using RemoteConnections. (To the observer, however, they will all appear to be running locally—the graphics don't attempt to distinguish between local and remote objects.) This mechanism is general enough to allow a whole system of objects to communicate across a network; whether they are servers and clients depends on who is providing the service and who is requesting it. (Keep in mind that Clients' changes are currently discarded. This should be fixed in a future release. For now, all objects are local to the Server machine, but they *are* communicating using the network.)
- `robotworld.ctl.CompositeController` is what you would use to encapsulate local behavior, such as the wall following described above. You would use this class when a direct connection between sensors and actuators isn't enough, and you want something more complex to control your vehicle. They can be recursively nested, as the name suggests. CompositeControllers are designed to look like computer chips, with pins on the top (for input) and pins on the bottom (for output). They start out with no pins at all. As on a real chip, these pins are just external

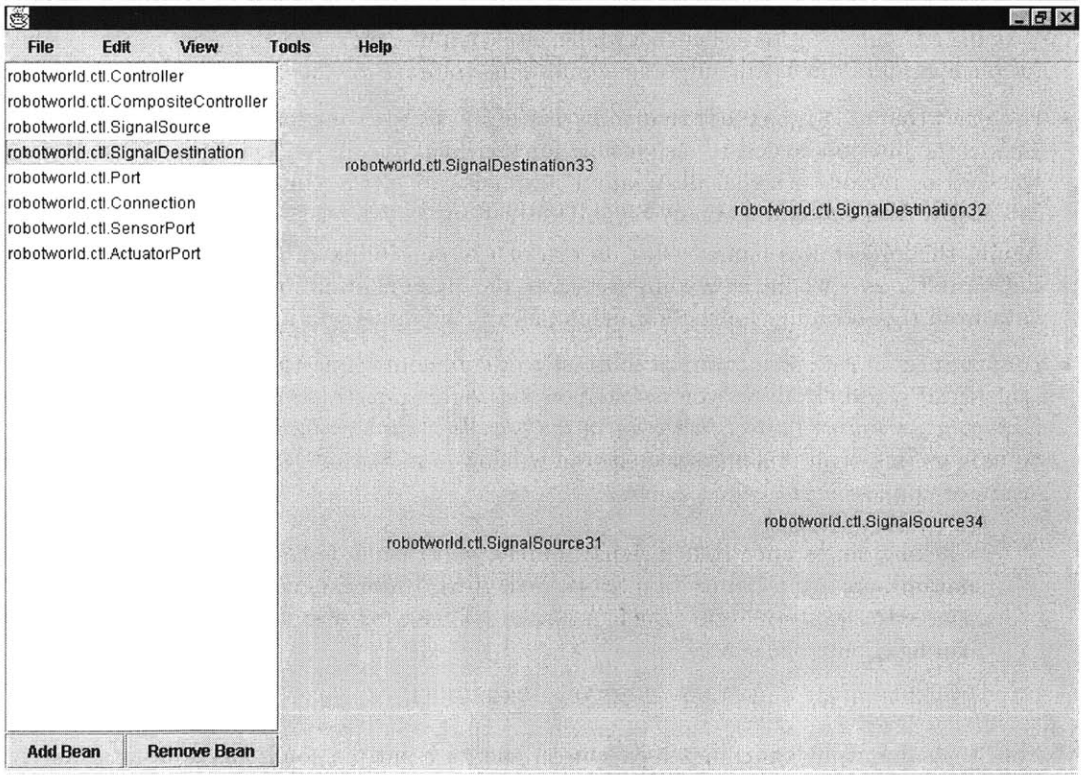
manifestations of internal structures. You will have to go inside the CompositeController by double-clicking on it—this will take you to The Behavior Layer (B.7.4)—in order to add the inputs and outputs which will pop out the pins that allow it to interface with the rest of the Morphology layer.

All right! Now that you have an idea of the capabilities of this layer, it's time to create something. Try the following for starters:

- Select DifferentialDrive from the Beanbag and create one.
- Select PhotoSensor.Left from the Beanbag and create one.
- Select RemoteConnection from the Beanbag. The window is now in *Special Mode.*
- Click first on the PhotoSensor.Left, and then on the *right* motor pin of the DifferentialDrive. This will create a RemoteConnection between the two. This means that the signals being generated by the photosensor are being fed to the right motor. There is a small amount of ambient light in RobotWorld, so if you look back in the Habitat Window, you should see the vehicle start to spin.
- Select PhotoSensor.Right from the Beanbag, and create one.
- Select RemoteConnection from the Beanbag. The window is now in *Special Mode.*
- Click first on the PhotoSensor.Right, and then on the *left* motor pin of the DifferentialDrive. This will create a RemoteConnection between the two. Since the light is even for both photosensors, the vehicle should straighten out back in the Habitat Window.
- Now add a Beacon to the vehicle. It glows at a fixed intensity, so you don't need to connect anything to it at this point.
- Now double-click on one of the vehicles in the Habitat Window and set it up using exactly the same components as the first vehicle. However, wire it a little differently. Try connecting the left photosensor to the left motor, and the right photosensor to the right motor. How do the two vehicles interact?
- Although this is a great place to start exploring some robotic designs, for the sake of exposition we will move to the next layer. Select CompositeController from the Beanbag, and create one. Note that it has no pins yet, so it cannot be connected to any sensors or actuators. You will have to go inside the CompositeController by double-clicking on it—this will take you to The Behavior Layer (B.7.4)—in order to add the inputs and outputs which will pop out the pins that allow it to interface with the rest of the Morphology layer.

B.7.4 The Behavior Layer





These are both Behavior Windows. We arrived at the left window by double-clicking on a CompositeController from the Morphology Layer. We arrived at the right window by double-clicking on a CompositeController in the left window.

You will have noticed that there only a few kinds of things you can create in the Morphology Layer: sensors, actuators, remote connections, and composite controllers. It provides connectability, but not flexibility. That's where the Behavior Layer comes in: it stills allows you to connect sprites, but all the connections are local. And you can keep nesting composite controllers, or you can decide that a controller can't be decomposed any further without specifying its rules for behavior. At this point you can control its behavior using Java code. Let's introduce the cast of characters before going any further:

- `robotworld.ctl.CompositeController`: It appears here as well, so that you can keep nest them to any degree. When you double-click on one, its contents will appear in another behavior window. This is because you're not visiting a new layer of observation; the kinds of things you can create, and the ways of interacting with the space haven't changed.
- `robotworld.ctl.SignalSource`: finally! this is the internal structure responsible for the output pins in its enclosing sprite. Whenever you add or remove a signal source internally, an output pin will be added or removed externally. You'll have to put two windows side by side to see this effect, where one window views the sprite externally, as a whole, and the other window views the sprite internally, as an assortment of parts. This can be the Morphology Window, and the Behavior Window that popped up when you double-clicked on the CompositeController. If you start nesting controllers, then you can do the same thing to give them output pins as well. You can add as many SignalSources as you wish.

  A note about the name: this object was named with its *external* responsibility in mind. Viewing a CompositeController as a whole, as a computer chip, the SignalSource is responsible for pushing data out to its outgoing connections (plural) via the output pin, upon demand.
- `robotworld.ctl.SignalDestination`: this bean is just a mirror image of a SignalSource. This is the internal structure responsible for the *input* pins in its enclosing sprite. Whenever you add or remove a signal destination internally, an *input* pin will be added or removed externally. You can add as many SignalDestinations as you wish.

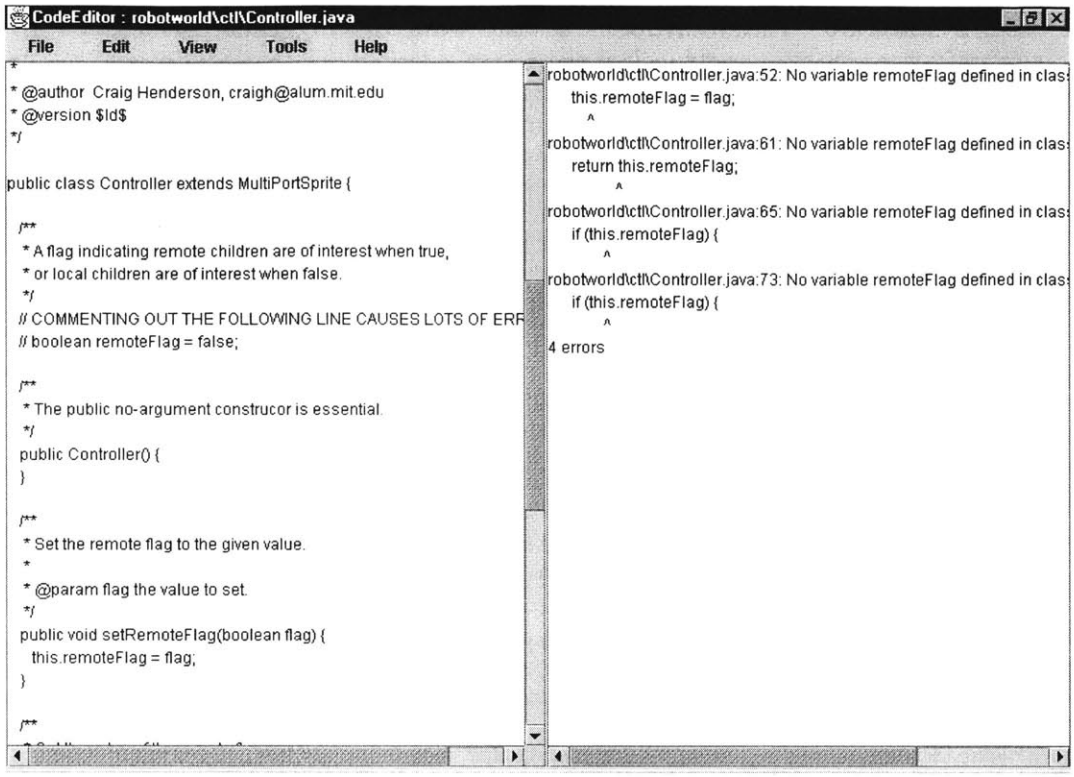
  Again, this object was named with its *external* responsibility in mind. Viewing a CompositeController as a whole, as a computer chip, the SignalDestination is responsible for pulling data from the incoming connection (singular) via the input pin, upon demand.
- `robotworld.ctl.Connection`: in contrast to the remote connection, this one is local. It too is a special sprite. It allows you to interconnect sources, destinations, and other components. *Important: when you select this bean in the bean bag, the standard user interface is disabled.* So long as this bean remains selected, the window is in *Special Mode* and this following user interface applies:

  - Clicking on a source (like a signal source), and then a destination (like an signal destination), creates a Connection between the two. It appears as a directed arrow from the source to the destination. *Rule:* A source can have many destinations, but a destination can have only one source.
  - Clicking on a Connection selects (or deselects it).

  Note that an output pin counts as a source, and an input pin counts as a destination. After all, they're just external representations of the internal structures.

- `robotworld.ctl.SensorPort`: this is just a specialized SignalSource whose job it is to continuously pull data from the network, and push it into the behavior system. As you might expect, its pins are designed to be connected to a Sensor (a remote connection, that is). Note that the Sensor is a remote source, and the SensorPort is a remote destination *in addition* to being a local source. This works out in the end. Here's why. In the Morphology layer, the arrow starts from the Sensor and *ends* at the SensorPort's pin. The SensorPort is a "consumer" of this remote data. But relative to the Behavior layer, the SensorPort is a "producer" of local data. Here, the arrow *starts* at the SensorPort and goes to a local destination. Contrast the Morphology and Behavior diagrams to see how this works.
- `robotworld.ctl.ActuatorPort`: similarly, this is a specialized SignalDestination whose job it is to continuously pull data from the behavior system, and push it to the network. This is just the opposite of what a SensorPort does. Its pins are designed to be remotely connected to an Actuator Note that the Actuator is a remote destination, and the ActuatorPort is a remote source *in addition* to being a local destination. This works out in the end. Here's why. In the Morphology layer, the arrow *starts* from the ActuatorPort's pin, and ends at the Actuator. The ActuatorPort is a "producer" of this remote data. But relative to the Behavior layer, the ActuatorPort is a "consumer" of local data. Here, the arrow starts at a local source and *ends* at the ActuatorPort. Contrast the Morphology and Behavior diagrams to see how this works.
- `robotworld.ctl.Controller`: this is a terminal controller–you can't nest any other controllers inside of it. This is also where the work gets done. A controller is responsible for continuously polling its inputs and pushing data to its outputs—and you'll get to write the code to do this. Create one and double-click on it to take you to the final layer of observation...

B.7.5 The Code Editing Layer



We arrived at this Code Editing Window by double-clicking on a Controller in the Behavior Layer.

The following assumes you are using the `jdk` configuration. Other configurations will be described at the end of this section.

The Code Editing layer is a very different beast from the other layers we've been examining. Fortunately, it should be very familiar as well. The Code Editing Window is a basic text editor which allows you to load, save, and edit Java files. You don't need to understand the Java code that has been loaded into the window, but we can tinker with it a little.

You should to comment out a portion of the code, in order to see what kinds of compiling errors you can get. Commenting means to mark off an area in your code file that will be deliberately ignored by the compiler. In java, you use the following scheme to comment your code: (here, the words "This is a comment." will be ignored by the compiler; note that in the second case, only the remaining text on the same line is ignored)

```
/* This is a comment. */
// This is also a comment.
/* This
 * is
```

```
* also
* a comment.
*/
```

Go ahead and comment out a line of code. Any line with a left curly brace '{' or right curly brace '}' will do. Now select the menu option **File** > **Save** to save the file, and then select **Tools** > **Compile** to compile it. Assuming your change was catastrophic, this will have generated a compiler error. The Code Editing Window will have inflated in size, so that now there are two panels. On the left is a panel containing your Java file. On the right is a panel listing all the errors that were generated as a result of your change. Try to see if they make any sense. Now, go back and restore the line that you commented out. Save it, and compile once more. The compiler errors should go away, and the error panel should shrivel up. This indicates that your code compiles without error.

All of the above describes the code editor that comes with RobotWorld. Your instructor may have supplied an alternate editor, such as Notepad (Windows), or Emacs (UNIX or Windows), or a commercial development environment such as IBM's VisualAge for Java. Here are the possible combinations:

- Using built-in editor, and an external compiler such as the JDK. The compiler output is captured in the Code Editor's error panel. (This was just described above).
- Using external text editor such as Notepad. The difference is that the text gets loaded into Notepad instead, and the code editor panel remains empty. You will have to use the save options in Notepad. However, you can still use an external compiler such as the JDK, and capture the compiler output in the error panel, as before. Just select **Tools** > **Compile** in the otherwise empty panel, and the compiling function will work as before.
- Using external environment for both editing and compiling. Emacs offers this, as do many commercially available systems. Your instructor will have to configure your system to hook into these external products. The text gets loaded into the external application, which most likely has its own options for compiling and saving, plus many other features. However, if you wish, you can select **Tools** > **Compile** in the otherwise empty panel to force the external application to compile. This might be desirable if compiling is not a straightforward operation using the application's interface.

That's it! You have just traversed all five layers of observation within RobotWorld. Now you can back up, and explore the interrelationships between them, and maybe try to program a few robot behaviors. And if you really want to roll your sleeves up, feel free to extend RobotWorld as you see fit. There are also a lot of areas we'd like to improve...

B.8 Future Directions

B.8.1 Example Problem Sets

In the future, we would like to provide several problem sets with the robotworld distribution. For now, see the problem set scenarios described in Craig Henderson's master's thesis, Chapter 10.

B.8.2 Extending RobotWorld

If you have decided to add a layer to RobotWorld or write your own implementation of a blackboard system, you should rely on The RobotWorld API (`/rw/javadoc/index.html`) for technical documentation. You may also be interested in Chapters 3–9 of Craig Henderson's master's thesis for an overview of the application framework and the details of each of its subsystems. Once you have written to the API, you can use the abstract factory mechanism by which RobotWorld pulls itself up by its bootstraps.

RobotWorld relies on .properties files for configuration. When the RobotWorld application is launched, it interprets all command line arguments as the names of .properties files. These files contain key-value pairs, one per line, that are loaded into the System properties of the running application. For instance,

```
java robotworld.Server singleuser layers jdk
```

would load the key-value pairs listed in files `singleuser.properties`, `layers.properties`, and `jdk.properties`. When you have written your extensions, use this mechanism to map the interfaces from the RobotWorld API to your implementations. Then when RobotWorld starts up, it will be able to use your classes instead.

B.8.3 Bugs and Features

There are (certainly) many features which we would like to add, and (probably) many bugs we would like to remove. Most importantly, RobotWorld's ability to save work is poor at best. Here is a partial list of features that one might expect:

- The provided Code Editor allows one to load and save text files, but it does not remind the user to save their work before a compile, or after the user closes the window.
- The bean bag allows one to add and remove beans, but the changes are lost when the application shuts down.
- The state of any particular workspace cannot currently be saved to a file or loaded. This means that all sprites instantiated or interconnected will be lost when the application shuts down.
- The multiuser blackboard is currently transient. This means that it doesn't have any of the persistence characteristics that a database might have. All blackboard entries are lost when the `server.bat` processes are terminated.
- Client changes aren't broadcast back to the server, so they are lost.

These will need to be addressed in future versions of RobotWorld. Keep in mind that the most current list of desired features is in Craig Henderson's master's thesis, Chapter 11. Please visit the Rethinking CS101 web pages for the latest status before sending email.

B.8.4 Finding Out More

You can find out more about the activities of the Rethinking CS101 project by following the links below.

This software is a part of Lynn Andrea Stein's (`http://www.ai.mit.edu/people/las/`) Rethinking CS101 project (`http://www.ai.mit.edu/projects/cs101/`) at the MIT Artificial Intelligence Lab (`http://www.ai.mit.edu/`) and the Department of Electrical Engineering and Computer Science (`http://www-eecs.mit.edu/`) at the Massachusetts Institute of Technology (`http://web.mit.edu/`).

Questions or comments:
<cs101-webmaster@ai.mit.edu>

Appendix C

Source Code

C.1 The robotworld.net package

Blackboard.java

```
/*
 * Blackboard Interface.
 * $Id: Blackboard.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.net;

/**
 * Provides an interface for asynchronous distributed communication.
 * Object entries can be written, read, or erased from
 * a common object store---the blackboard.  They can be retrieved by the
 * use of template matching, where specified fields are required to match
 * exactly and unspecified fields are treated as wildcards.
 * <p>
 * An entry matches a given template when:
 * <ul>
 * <li>The entry's class is assignable to the template's class, and
 * <li>Each of the template's public fields is either:
 *     <ul>
 *     <li>Null (indicating a wildcard), or
```

```
*     <li>Bytestream-equivalent to the corresponding field of the entry.
*         This means that the objects stored in those fields,
*         when serialized by a <code>java.rmi.MarshalledObject</code>,
*         are identical.  Here's a quick equivalence test:<br><br><code>
*         MarshalledObject marshA = new MarshalledObject(objA);<br>
*         MarshalledObject marshB = new MarshalledObject(objB);<br>
*         System.out.println("A==B?"+marshA.equals(marshB));<br>
*         System.out.println("B==A?"+marshB.equals(marshA));</code>
*     </ul>
* </ul>
* <p>
* This allows for loosely coupled inter-object communication, where
* the sender and receiver don't need to know whether they reside on the
* same machine, and don't even need to be synchronously connected.
* Note that a blackboard system is not an object database.
* All entries which match a given template are considered equal, and
* any one of them might be returned.  In other words, the contract is
* nondeterministic, even if a particular implementation is not.  Also,
* object identity is not preserved.  The blackboard only operates on copies.
* <p>
* This interface was heavily inspired by Sun's JavaSpaces.<br>
* Copyright (C) 1999 Massachusetts Institute of Technology.
*
* @author  Craig Henderson, craigh@alum.mit.edu
* @version $Id: Blackboard.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
*/

public interface Blackboard
{
  /**
   * Copy an entry onto the blackboard.
   *
   * @param     entry The blackboard entry to be copied.
   * @exception BlackboardException
   *            If the blackboard cannot comply.
   */
  public void write(BlackboardEntry entry)
    throws BlackboardException;

  /**
   * Immediately copy and return any entry matching the template.
   * If a match can be found, one entry is copied
   * from the blackboard and returned to the caller.
   * Otherwise, null is returned instead.
   *
   * @param     template a blackboard entry to match against.
   * @return    any blackboard entry matching the template,
   *            or null if none could be found.
   * @exception BlackboardException
   *            If the blackboard cannot comply.
   */
  public BlackboardEntry read(BlackboardEntry template)
    throws BlackboardException;

  /**
   * Immediately remove and return any entry matching the template.
   * If a match can be found, one entry is permanently
   * removed from the blackboard and returned to the caller.
   * Otherwise, the blackboard is unaffected and null
   * is returned instead.
   *
   * @param     template a blackboard entry to match against.
   * @return    any blackboard entry matching the template,
   *            or null if none could be found.
   * @exception BlackboardException
   *            If the blackboard cannot comply.
   */
  public BlackboardEntry take(BlackboardEntry template)
```

```
    throws BlackboardException;

  /**
   * Ask to be notified of any writes to the blackboard
   * that match the given template.  On every matching write
   * thereafter, the listener's notify method will be called.
   *
   * @param     template a blackboard entry to match against.
   * @param     listener the blackboard listener to be notified
   *            on every matching write.
   * @exception BlackboardException
   *            If the blackboard cannot comply.
   */
  public void listen(BlackboardEntry template,
    BlackboardListener listener)
    throws BlackboardException;
}

/**
 * $Log: Blackboard.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

BlackboardEntry.java

```
/*
 * Blackboard Entry Interface.
 * $Id: BlackboardEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.net;
import java.io.*;

/**
 * An interface for objects which can be stored on a blackboard.
 * Objects implementing <code>BlackboardEntry</code>
 * must take the same precautions as any other <code>Serializable</code>
 * object, plus additional design constraints.  To facilitate its use
 * by the blackboard, entries must have a public no-arg constructor.
 * Only public, non-static, non-transient, non-final fields will be
 * reconstituted after storage on the blackboard.  Additionally,
 * these fields cannot be of primitive type, and should only contain
 * references to <code>Serializable</code> objects, or null,
 * indicating a wildcard.
 * <p>
 * This interface was heavily inspired by Sun's JavaSpaces.<br>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: BlackboardEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */
```

```
public interface BlackboardEntry extends Serializable
{
}

/**
 * $Log: BlackboardEntry.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

BlackboardEvent.java

```
/*
 * Blackboard Event Class.
 * $Id: BlackboardEvent.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.net;
import java.util.*;

/**
 * Fired whenever a newly written entry matches the listener's template.
 * Records the source of the event,
 * an event descriptor, and a monotonically increasing
 * sequence number.
 * <p>
 * This class was heavily inspired by Sun's JavaSpaces.<br>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: BlackboardEvent.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class BlackboardEvent extends EventObject
{
  /**
   * A long which should uniquely identify the kind of event.
   */
  protected long eventID;

  /**
   * A long which acts as a kind of time stamp.
   */
  protected long seqNum;

  /**
   * Gets the event descriptor.
   *
   * @param src an object which is the source of the event.
   * @param eventID a long which should uniquely identify the kind of event.
   * @param seqNum a long which acts as a kind of time stamp.
   */
  public BlackboardEvent(Object src, long eventID, long seqNum) {
```

```
    super(src);
    this.eventID = eventID;
    this.seqNum = seqNum;
  }

  /**
   * Gets the event descriptor.
   *
   * @return a long which should uniquely identify the kind of event.
   */
  public long getID() {
    return eventID;
  }

  /**
   * Gets the sequence number.
   *
   * @return a long which acts as a kind of time stamp.
   */
  public long getSequenceNumber() {
    return seqNum;
  }

  public String toString() {
    return "EVENT( "+this.eventID+" , "+this.seqNum+" )";
  }
}

/**
 * $Log: BlackboardEvent.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

BlackboardException.java

```
/*
 * Blackboard Exception Class.
 * $Id: BlackboardException.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.net;
import java.io.*;

/**
 * Thrown when a blackboard cannot comply with a request.
 * Typically this is because of an I/O problem.
 * <p>
 * This class was heavily inspired by Sun's JavaSpaces.<br>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: BlackboardException.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
```

```
 */

public class BlackboardException extends IOException
{
  public BlackboardException() {
    super();
  }

  public BlackboardException(String s) {
    super(s);
  }
}

/**
 * $Log: BlackboardException.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

BlackboardListener.java

```
/*
 * Blackboard Listener Interface.
 * $Id: BlackboardListener.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.net;
import java.util.*;

/**
 * Listens for writes to the blackboard matching a given template.
 * <p>
 * This interface was heavily inspired by Sun's JavaSpaces.<br>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: BlackboardListener.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public interface BlackboardListener
   extends EventListener
{

  /**
   * Notifies this listener that a write happened.
   *
   * @param event a blackboard event capturing the relevant details.
   * @exception BlackboardException
   *              If the listener cannot comply.
   */
  public void notify(BlackboardEvent event)
    throws BlackboardException;
}
```

```
/**
 * $Log: BlackboardListener.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

WorkspaceEntry.java

```
/*
 * Workspace Entry Interface.
 * $Id: WorkspaceEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.net;

/**
 * A specialized entry for loosely coupled inter-agent communication.
 * Implementing classes must provide getter and setter methods for each
 * of the following: server, domain, verb, subject, client, all of type
 * <code>Object</code>.
 * <p>
 * This template is to be interpreted as follows:<br>
 * "[Server] in [Domain], please [Verb] [Subject] for [Client]."
 * <p>
 * If the entry is meant to be read, the server should be EVERYBODY.<br>
 * If the entry is meant to be taken, the server should be ANYBODY.<br>
 * If the entry is meant for a particular server, use its unique identifier.<br>
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: WorkspaceEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public interface WorkspaceEntry extends BlackboardEntry
{

  // These constants need to be part of the interface,
  // rather than the implementing class.  Otherwise,
  // since they're public, they'd get serialized as well and
  // would unnecessarily bloat the entries and waste bandwidth.

  /**
   * A server value indicating the entry can be taken.
   */
  public static final String ANYBODY = "ANYBODY";

  /**
   * A server value indicating the entry can be read.
   */
  public static final String EVERYBODY = "EVERYBODY";
```

```
/**
 * A domain value indicating the entry has to do with dynamically
 * changing signals.
 */
public static final String STATE_SPACE = "State Space";

/**
 * A domain value indicating the entry has to do with code editing.
 */
public static final String CODE_EDITOR = "Code Editor";

/**
 * A domain value indicating the entry has to do with behavior editing.
 */
public static final String BEHAVIOR_EDITOR = "Behavior Editor";

/**
 * A domain value indicating the entry has to do with morphology editing.
 */
public static final String MORPHOLOGY_EDITOR = "Morphology Editor";

/**
 * A domain value indicating the entry has to do with habitat editing.
 */
public static final String HABITAT_EDITOR = "Habitat Editor";

/**
 * A domain value indicating the entry has to do with architecture editing.
 */
public static final String ARCHITECTURE_EDITOR = "Architecture Editor";



/**
 * Identify the intended recipient[s] of this request.
 * Typical values include <code>EVERYBODY</code>,
 * <code>ANYBODY</code>.
 *
 * @param server an object which acts as an identifier.
 */
public void setServer(Object server);

/**
 * Determine the intended recipient[s] of this request.
 * Typical values include <code>EVERYBODY</code>,
 * <code>ANYBODY</code>.
 *
 * @return an object which acts as an identifier.
 */
public Object getServer();



/**
 * Identify the intended domain of this request.
 * This serves to further restrict the recipients of this message.
 * Typical values include <code>CODE_EDITOR</code>,
 * <code>BEHAVIOR_EDITOR</code>, <code>MORPHOLOGY_EDITOR</code>,
 * <code>HABITAT_EDITOR</code>, <code>ARCHITECTURE_EDITOR</code>.
 *
 * @param domain an object which acts as a domain.
 */
public void setDomain(Object domain);

/**
 * Determine the intended domain of this entry.
 * This serves to further restrict the recipients of this message.
```

```
    * Typical values include <code>CODE_EDITOR</code>,
    * <code>BEHAVIOR_EDITOR</code>, <code>MORPHOLOGY_EDITOR</code>,
    * <code>HABITAT_EDITOR</code>, <code>ARCHITECTURE_EDITOR</code>.
    *
    * @return an object which acts as a domain.
    */
   public Object getDomain();



   /**
    * Identify the desired action or effect.
    *
    * @param server an object which describes an action or effect.
    */
   public void setVerb(Object verb);

   /**
    * Determine the desired action or effect.
    *
    * @return an object which describes an action or effect.
    */
   public Object getVerb();



   /**
    * Identify the subject of the desired action or effect.
    *
    * @param server an object which acts as an identifier.
    */
   public void setSubject(Object subject);

   /**
    * Determine the subject of the desired action or effect.
    *
    * @return an object which acts as an identifier.
    */
   public Object getSubject();



   /**
    * Identify the entity making the request.
    *
    * @param server an object which acts as an identifier.
    */
   public void setClient(Object client);

   /**
    * Determine the entity making the request.
    *
    * @return an object which acts as an identifier.
    */
   public Object getClient();
}

/**
 * $Log: WorkspaceEntry.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

C.2 The robotworld.net.simple package

SimpleBlackboard.java

```
/*
 * Simple Blackboard Implementation.
 * $Id: SimpleBlackboard.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.net.simple;
import robotworld.net.*;
import java.util.*;
import java.lang.reflect.*;
import java.rmi.*;
import java.io.IOException;

/**                                                                            20
 * A simple, naive, shared-address space blackboard.
 * It can only be used by objects in the same Java virtual machine.
 * It doesn't use the network at all.  Currently, this implementation
 * preserves object identity, by storing object references directly.
 * This is in violation of the <code>Blackboard</code> specification.
 * Do not expect other blackboard implementations to adhere to this policy.
 * <p>
 * This clas was heavily inspired by Sun's JavaSpaces.<br>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *                                                                            30
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: SimpleBlackboard.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class SimpleBlackboard implements Blackboard
{
  /**
   * Entries must be unique in this implementation.
   */
  protected HashSet entrySet = new HashSet();                                  40

  /**
   * Each listener can have exactly one template.
   * Listeners are keys, templates are values.
   */
  protected HashMap listenerMap = new HashMap();

  /**
   * Only one flavor of Blackboard event is currently supported.
   */                                                                          50
  public static final long WRITE_EVENT = 5L;

  /**
   * A way of time-stamping Blackboard events.
   */
  protected long writeSequenceNumber = 0L;

  /**
   * The no-argument constructor is required.
```

```
 * Otherwise this implementation would not be compatibile
 * with the RobotWorld abstract factory mechanism.
 * @see robotworld.ide.Workspace#produce
 */
public SimpleBlackboard() {
}

/**
 * Determines if the entry matches the given template.
 * Returns true if:
 * <ul>
 * <li>The entry's class is assignable to the template's class, and
 * <li>Each of the template's public fields is either:
 *    <ul>
 *    <li>Null (indicating a wildcard), or
 *    <li>Object--equivalent to the corresponding field of the entry.
 *        This means that the objects stored in those fields,
 *        when compared using the <code>equals</code> method of one
 *        with the other as an argument, are identical.
 *        Here's a quick equivalence test:<br><br><code>
 *        System.out.println("A==B?"+objA.equals(objB));<br>
 *        System.out.println("B==A?"+objB.equals(objA));</code>
 *    </ul>
 * </ul>
 * Note that this method is not commutative.  In other words,
 * <code>SimpleServer.matches(a,b)</code> is not necessarily the same as
 * <code>SimpleServer.matches(b,a)</code>.  This is because of the use of
 * wildcards breaks the symmetry.
 * <p>
 * Also note that this method is not consistent with the description
 * of template matching in the <code>Blackboard</code> documentation.
 *
 * @param template a blackboard entry to test against.
 * @param entry a blackboard entry candidate.
 * @return true if the above conditions are met, false otherwise.
 */
public static boolean matches(BlackboardEntry template,
  BlackboardEntry entry) {
   try {

     // the template class must be assignable from the entry class
     Class templateClass = template.getClass();
     Class entryClass = entry.getClass();
     if (!templateClass.isAssignableFrom(entryClass)) return false;

     // for each field of templateClass
     Field[] fields = templateClass.getFields();
     for (int i=0; i<fields.length; i++) {
       Field f = fields[i];
       Object objTemplate = f.get(template);
       Object objEntry = f.get(entry);

       // if the template specifies a wildcard, continue
       if (objTemplate == null) continue;

       // otherwise, if the values are equal, continue
       else if (objTemplate.equals(objEntry)) continue;

       // otherwise, there was not a match
       else return false;
     }

     // perfect match!
     return true;

   } catch (Exception e) {
     // Reflection problems are a big pain.
     throw new IllegalArgumentException(e.toString());
```

```
  }
}
                                                                          130
// writes take O(n) time, where n is the number of listeners.
public synchronized void write(BlackboardEntry entry)
  throws BlackboardException {

  this.entrySet.add(entry);

  // iterate over all the listeners
  BlackboardListener listener = null;
  BlackboardEntry template = null;
  for (Iterator iter=this.listenerMap.keySet().iterator();                140
       iter.hasNext(); ) {
    listener = (BlackboardListener)iter.next();
    template = (BlackboardEntry)this.listenerMap.get(listener);

    // notify the listener if there's a match
    if (matches(template, entry))
    {
      listener.notify(new BlackboardEvent(entry,
                                          this.WRITE_EVENT,
                                          this.writeSequenceNumber));     150
    }
  }
  this.writeSequenceNumber++;
}

// reads take O(n) time, where n is the number of entries
public synchronized BlackboardEntry read(BlackboardEntry template)
  throws BlackboardException {

  // iterate over the entries, looking for a match                        160
  BlackboardEntry entry = null;
  for (Iterator iter=this.entrySet.iterator(); iter.hasNext(); ) {
    entry = (BlackboardEntry)iter.next();

    if (matches(template, entry))
    {
      // return immediately
      return entry;
    }
  }                                                                       170

  // no match found, return null
  return null;
}

// takes take O(n) time, where n is the number of entries
public synchronized BlackboardEntry take(BlackboardEntry template)
  throws BlackboardException {

  // iterate over the entries, looking for a match                        180
  BlackboardEntry entry = null;
  for (Iterator iter=this.entrySet.iterator(); iter.hasNext(); ) {
    entry = (BlackboardEntry)iter.next();

    if (matches(template, entry))
    {
        // remove the entry from the blackboard, and return immediately
      iter.remove();
      return entry;
    }                                                                     190
  }

  // no match found, return null
  return null;
}
```

```
  // listens take O(1) time
  public synchronized void listen(BlackboardEntry template,
    BlackboardListener listener) throws BlackboardException {
    this.listenerMap.put(listener, template);
  }
}
```

SimpleEntry.java

```
/*
 * Simple Entry Abstract Class.
 * $Id: SimpleEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.net.simple;
import robotworld.net.*;
import java.lang.reflect.*;

/**
 * Provides default entry behavior for the simple blackboard.
 * The instance methods attempt to call the static methods
 * wherever possible.
 * <p>
 * This class was heavily inspired by Sun's JavaSpaces.<br>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: SimpleEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public abstract class SimpleEntry implements BlackboardEntry
{
  /**
   * Entries must have a constructor with no arguments.
   * This is true of each and every subclass as well,
   * otherwise they cannot be used by the blackboard.
   */
  public SimpleEntry() {
  }

  public boolean equals(Object object) {
    if (object instanceof BlackboardEntry) {
      return SimpleEntry.equals(this, (BlackboardEntry)object);
    } else {
      return super.equals(object);
    }
  }

  public int hashCode() {
    return SimpleEntry.hashCode(this);
  }
```

```
public String toString() {
  return SimpleEntry.toString(this);
}

/**
 * Determines if the two entries are equal.
 * Returns true if:
 * <ul>
 * <li>The entries are of the same class.
 * <li>For each field of the two entries:                                    60
 *   <ul>
 *   <li>Both fields are null, or
 *   <li>Neither field is null, and calling equals() on the value of one
 *       with the value of the other returns true.
 *   </ul>
 * <ul>
 * Otherwise, returns false.
 *
 * @param entry1 a blackboard entry.
 * @param entry2 another blackboard entry.                                   70
 * @return true if the two are equivalent, false otherwise.
 */
public static boolean equals(BlackboardEntry entry1,
                             BlackboardEntry entry2) {

  // the entries must be of the same class
  Class c1 = entry1.getClass();
  Class c2 = entry2.getClass();
  if (!c1.getName().equals(c2.getName())) return false;
                                                                           80
  // for each field of the two entries
  try {
    Field[] fields = c1.getFields();
    for (int i=0; i<fields.length; i++) {
      Field f = fields[i];
      Object o1 = f.get(entry1);
      Object o2 = f.get(entry2);

      // both fields must be null in the spec to be
      // considered equal.                                                   90
      if (o1==null && o2==null) continue;

      else if (o1==null || o2==null) return false;

      // or the values must equal each other
      else if (o1.equals(o2)) continue;

      // or there was not a match
      else return false;
    }                                                                       100

  } catch (Exception e) {
    // Reflection problems are a big pain.
    throw new IllegalArgumentException(e.toString());
  }

  // perfect match!
  return true;
}
                                                                           110
/**
 * Generates a hash code for the given entry.
 * Returns the result of exclusive-OR'ing a seed value of zero
 * with the hash codes of all non-null fields.
 *
 * @param entry the blackboard entry to hash.
 * @return the resulting hash code.
 */
```

```
  public static int hashCode(BlackboardEntry entry) {
    Class c = entry.getClass();
    int hash = 0; // initial value of 0;

    // for each field of the entry
    try {
      Field[] fields = c.getFields();
      for (int i=0; i<fields.length; i++) {
        Object o = fields[i].get(entry);

        // non-null fields only
        if (o==null) continue;

        // XOR with the hashcode of the field's value
        hash ^= o.hashCode();
      }

    } catch (Exception x) {
      // Reflection problems are a big pain.
      throw new IllegalArgumentException(x.toString());
    }

    // done!
    return hash;
  }

  /**
   * Generates a human readable string for the given entry.
   * Intended for debugging.  When used in conjunction with
   * <code>System.out.println</code>, will have the result
   * of printing out the class name, a seperator line,
   * and the name and value of each field, one per line.
   *
   * @param entry the blackboard entry to convert.
   * @return a human readable string representation of the entry.
   */
  public static String toString(BlackboardEntry entry) {
    Class c = entry.getClass();
    String str = c.getName()+"\n----------------\n";

    // for each field of the entry
    try {
      Field[] fields = c.getFields();
      for (int i=0; i<fields.length; i++) {
        Field f = fields[i];
        Object o = f.get(entry);

        str += f.getName() +": "+ o + "\n";
      }

    } catch (Exception x) {
      // Reflection problems are a big pain.
      throw new IllegalArgumentException(x.toString());
    }

    // done!
    return str;
  }
}

/**
 * $Log: SimpleEntry.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

SimpleRedirectEntry.java

```
/*
 * Simple Redirect Entry Class.
 * $Id: SimpleRedirectEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                                10
 */

package robotworld.net.simple;
import robotworld.net.*;
import robotworld.ctl.*;

/**
 * Provides a simple implementation of a redirect entry.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.                     20
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: SimpleRedirectEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class SimpleRedirectEntry extends SimpleWorkspaceEntry
implements RedirectEntry {

  /**
   * A public no-argument constructor is essential.                              30
   * By default, the server is <code>WorkspaceEntry.ANYBODY</code>,
   * the domain is <code>WorkspaceEntry.MORPHOLOGY_EDITOR</code>, and the
   * the verb is <code>RedirectEntry.REDIRECT</code>.
   */
  public SimpleRedirectEntry() {
    this.server = WorkspaceEntry.ANYBODY;
    this.domain = WorkspaceEntry.MORPHOLOGY_EDITOR;
    this.verb = RedirectEntry.REDIRECT;
    this.subject = null;
    this.client = null;                                                         40
  }
}

/**
 * $Log: SimpleRedirectEntry.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

SimpleSignalEntry.java

```
/*
 * Simple Signal Entry Class.
 * $Id: SimpleSignalEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
```

```
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.net.simple;
import robotworld.net.*;
import robotworld.ctl.*;

/**
 * Provides a simple implementation of a signal entry.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.                   20
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: SimpleSignalEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class SimpleSignalEntry extends SimpleWorkspaceEntry
implements SignalEntry
{
  public Object sample;
  public Object timestamp;                                                     30

  /**
   * A public no-argument constructor is essential.
   * By default, the server is <code>WorkspaceEntry.EVERYBODY</code>,
   * the domain is <code>WorkspaceEntry.STATE_SPACE</code>, and the
   * the verb is <code>SignalEntry.SAMPLE</code>.
   */
  public SimpleSignalEntry() {
    this.server = WorkspaceEntry.EVERYBODY;
    this.domain = WorkspaceEntry.STATE_SPACE;                                  40
    this.verb = SignalEntry.SAMPLE;
    this.subject = null;
    this.client = null;
    this.sample = null;
    this.timestamp = null;
  }

  public void setSample(Object sample) {
    this.sample = sample;
  }                                                                            50

  public Object getSample() {
    return this.sample;
  }

  public void setTimestamp(Object timestamp) {
    this.timestamp = timestamp;
  }

  public Object getTimestamp() {                                               60
    return this.timestamp;
  }
}

/**
 * $Log: SimpleSignalEntry.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */                                                                            70
```

SimpleWorkspaceEntry.java

```
/*
 * Simple Workspace Entry Class.
 * $Id: SimpleWorkspaceEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.net.simple;
import robotworld.net.*;

/**
 * A workspace entry for the simple blackboard.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *                                                                             20
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: SimpleWorkspaceEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class SimpleWorkspaceEntry extends SimpleEntry implements WorkspaceEntry
{
  /**
   * If it's not public, it won't get serialized.
   */
  public Object server, domain, verb, subject, client;                         30

  /**
   * This class must have a no-arg constructor.
   */
  public SimpleWorkspaceEntry() {
  }

  public void setServer(Object server) {
    this.server = server;
  }                                                                            40

  public Object getServer() {
    return this.server;
  }

  public void setDomain(Object domain) {
    this.domain = domain;
  }

  public Object getDomain() {                                                  50
    return this.domain;
  }

  public void setVerb(Object verb) {
    this.verb = verb;
  }

  public Object getVerb() {
    return this.verb;
  }                                                                            60

  public void setSubject(Object subject) {
    this.subject = subject;
  }
```

```
  public Object getSubject() {
    return this.subject;
  }

  public void setClient(Object client) {
    this.client = client;
  }

  public Object getClient() {
    return this.client;
  }
}

/**
 * $Log: SimpleWorkspaceEntry.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

C.3 The robotworld.net.jini package

JiniBlackboard.java

```
/*
 * Jini Blackboard Class.
 * $Id: JiniBlackboard.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.net.jini;
import robotworld.net.*;

// Don't you love their naming conventions?  Not that mine are any better.
import com.sun.jini.mahout.binder.RefHolder;
import net.jini.core.lease.Lease;
import net.jini.core.entry.Entry;
import net.jini.core.transaction.Transaction;
import net.jini.space.JavaSpace;
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
import java.rmi.MarshalledObject;

/**
 * A network-enabled blackboard based on Jini/JavaSpaces.
 * It can be used by objects on many Java virtual machines, whether
 * they reside on the same physical machine or on multiple nodes
 * on the Internet.
 * <p>
```

```
 * This class was heavily inspired by Sun's JavaSpaces.<br>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: JiniBlackboard.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class JiniBlackboard implements Blackboard
{
  /**
   * A reference to a JavaSpace stub.
   */
  protected JavaSpace javaSpace;

  /**
   * Property which allows a JavaSpace instance to be requested by name.
   */
  public static String NAME = "NAME";

  /**
   * Get a remote reference to a particular space.
   */
  public JiniBlackboard()
    throws BlackboardException
  {
    // Much of the constructor is copied from examples.spaces.HelloWorld,
    // which is [no longer] bundled with the JavaSpaces distribution.
    // Until Sun gets its act together, this can only be found in the
    // javaspaces-users@java.sun.com mailing list archive, which lives at
    // http://archives.java.sun.com/archives/javaspaces-users.html.
    // Keywords for search: HelloWorld, RefHolder, Naming.lookup().

    try {
      // Set a security manager so that the entry classes can be
      // uploaded to the server from this codebase
      if (System.getSecurityManager() == null) {
        System.setSecurityManager(new RMISecurityManager());
      }

        // Find the System property for the space's name.
      String spaceName = System.getProperty(
        this.getClass().getName() + "." + NAME);

      // RefHolderImpl is the remote object registered with
      // the registry
      RefHolder rh = (RefHolder)Naming.lookup(spaceName);

      // Use the RefHolder's proxy method to get the space
      // reference
      this.javaSpace = (JavaSpace)rh.proxy();

    } catch (Exception e) {
      throw new BlackboardException(e.toString());
    }
  }

  public void write(BlackboardEntry entry)
    throws BlackboardException
  {
    try {
      // Cast to the right type
      Entry entry2 = (Entry)entry;

      // The transaction under which to perform the write
      Transaction txn = null;

      // The lease duration that should be requested for
      // this entry
```

```
      long timeToLive = Lease.FOREVER;

      this.javaSpace.write(entry2, txn, timeToLive);

    } catch (Exception e) {
      throw new BlackboardException(e.toString());
    }
  }

  // perform an immediate read.
  public BlackboardEntry read(BlackboardEntry template)
    throws BlackboardException
  {
    try {
      // Cast to the right type
      Entry template2 = (Entry)template;

      // The amount of time to wait for a match
      // Beware, a value of 0L means "wait forever".
      long timeToWait = JavaSpace.NO_WAIT;

      // The transaction under which to perform the read
      Transaction sotxn = null;

      Entry entry = this.javaSpace.readIfExists(
        template2, sotxn, timeToWait);
      BlackboardEntry result = (BlackboardEntry)entry;
      return result;

    } catch (Exception e) {
      throw new BlackboardException(e.toString());
    }
  }

  // perform an immediate take.
  public BlackboardEntry take(BlackboardEntry template)
    throws BlackboardException
  {
    try {
      // Cast to the right type
      Entry template2 = (Entry)template;

      // The amount of time to wait for a match
      // Beware, a value of 0L means "wait forever".
      long timeToWait = JavaSpace.NO_WAIT;

      // The transaction under which to perform the read
      Transaction sotxn = null;

      Entry entry = this.javaSpace.takeIfExists(
        template2, sotxn, timeToWait);
      BlackboardEntry result = (BlackboardEntry)entry;
      return result;

    } catch (Exception e) {
      throw new BlackboardException(e.toString());
    }
  }

  public void listen(BlackboardEntry template,
                     final BlackboardListener listener)
    throws BlackboardException
  {
    try {
      // Cast to the right type
      Entry template2 = (Entry)template;

      // The transaction under which to perform the write
```

```
        Transaction txn = null;

        // Provide an anonymous wrapper around our interface
        // so that it looks like the alien interface
        JiniListener listener2 = new JiniListener();
        listener2.setBlackboardListener(listener);
        listener2.setTemplate(template);

        // The lease duration that should be requested for
        // this entry
        long timeToLive = Lease.FOREVER;

        // The handback object to be included in the
        // triggered RemoteEvent
        MarshalledObject handback = null;

        // I think that you should be able to "listen"
           // to a blackboard, and your listener should be "notified".
           // JavaSpaces uses the same method name, "notify",
           // for both activities.  To me, this just invites confusion.
        this.javaSpace.notify(template2, txn, listener2,
                              timeToLive, handback);
      } catch (Exception e) {
        throw new BlackboardException(e.toString());
      }
    }
}

/**
 * $Log: JiniBlackboard.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

JiniEntry.java

```
/*
 * Jini Entry Class.
 * $Id: JiniEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.net.jini;
import robotworld.net.*;
import java.lang.reflect.*;
import net.jini.entry.*;

/**
 * Provides default entry behavior for the Jini blackboard.
 * <p>
 * This class was heavily inspired by Sun's JavaSpaces.<br>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
```

```
 * @version $Id: JiniEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class JiniEntry extends AbstractEntry implements BlackboardEntry
{
  /**                                                                    30
   * Must have a no-arg constructor.
   */
  public JiniEntry() {
  }
}

/**
 * $Log: JiniEntry.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.                                                    40
 *
 */
```

JiniListener.java

```
/*
 * Jini Listener Class.
 * $Id: JiniListener.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                       10
 */

package robotworld.net.jini;
import robotworld.net.*;
import net.jini.core.event.*;
import java.io.Serializable;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

/**                                                                     20
 * An adaptor which allows a blackboard listener to interface with Jini.
 * Listens for Jini-style remote events, and translates these into
 * blackboard events which a blackboard listener knows how to handle.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: JiniListener.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */
                                                                        30
public class JiniListener extends UnicastRemoteObject
implements RemoteEventListener {

  public BlackboardListener listener;
  public BlackboardEntry template;
  public Blackboard blackboard;

  public JiniListener() throws RemoteException {
    super();
  }                                                                     40
```

```
  public void setTemplate(BlackboardEntry template) {
    this.template = template;
  }

  public BlackboardEntry getTemplate() {
    return this.template;
  }

  public void setBlackboardListener(BlackboardListener listener) {
    this.listener = listener;
  }

  public BlackboardListener getBlackboardListener() {
    return this.listener;
  }

  public void setBlackboard(Blackboard blackboard) {
    this.blackboard = blackboard;
  }

  public Blackboard getBlackboard() {
    return this.blackboard;
  }

  public void notify(RemoteEvent event)
    throws UnknownEventException, RemoteException
  {
    // the given Source (a JavaSpaces stub) is useless to me.
    // I would much rather have the matching entry that
    // caused the notify.  the original template
    // is put to work to read the result from the space.
    //
    // CAH<28 Apr 1999> Oops! This is completely wrong.
    // There is no guarantee that a read() at this point
    // will return the entry that was just written, since
    // JavaSpaces implementations are free to return _any_
    // entry that matches a given template.  In fact, if take()
    // is being called anywhere, there's no guarantee
    // that the entry is still around to be read!
    // FIX: If the entry that caused the read is needed,
    // a blocking read() should be used instead.
    // Notify() is only a messenger, and a dumb one at that.
    try {
      BlackboardEntry result = this.getBlackboard().read(template);

      // now create an event with the result as the source
      BlackboardEvent event2 = new BlackboardEvent(
        result, event.getID(), event.getSequenceNumber());

      // and notify the blackboard listener
      listener.notify(event2);

    } catch (BlackboardException e) {
      throw new RemoteException(e.toString());
    }
  }
}

/**
 * $Log: JiniListener.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

JiniRedirectEntry.java

```
/*
 * Jini Redirect Entry Class.
 * $Id: JiniRedirectEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.net.jini;
import robotworld.net.*;
import robotworld.ctl.*;

/**
 * Provides a Jini implementation of a redirect entry.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.                   20
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: JiniRedirectEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class JiniRedirectEntry extends JiniWorkspaceEntry
implements RedirectEntry {

  /**
   * A public no-argument constructor is essential.                            30
   * By default, the server is <code>WorkspaceEntry.ANYBODY</code>,
   * the domain is <code>WorkspaceEntry.MORPHOLOGY_EDITOR</code>, and the
   * the verb is <code>RedirectEntry.REDIRECT</code>.
   */
  public JiniRedirectEntry() {
    this.server = WorkspaceEntry.ANYBODY;
    this.domain = WorkspaceEntry.MORPHOLOGY_EDITOR;
    this.verb = RedirectEntry.REDIRECT;
    this.subject = null;
    this.client = null;                                                       40
  }
}

/**
 * $Log: JiniRedirectEntry.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

JiniSignalEntry.java

```
/*
 * Jini Signal Entry Class.
 * $Id: JiniSignalEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
```

```
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.net.jini;
import robotworld.net.*;
import robotworld.ctl.*;

/**
 * Provides a Jini implementation of a signal entry.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.                   20
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: JiniSignalEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class JiniSignalEntry extends JiniWorkspaceEntry
implements SignalEntry
{
  public Object sample;
  public Object timestamp;                                                     30

  /**
   * A public no-argument constructor is essential.
   * By default, the server is <code>WorkspaceEntry.EVERYBODY</code>,
   * the domain is <code>WorkspaceEntry.STATE_SPACE</code>, and the
   * the verb is <code>SignalEntry.SAMPLE</code>.
   */
  public JiniSignalEntry() {
    this.server = WorkspaceEntry.EVERYBODY;
    this.domain = WorkspaceEntry.STATE_SPACE;                                  40
    this.verb = SignalEntry.SAMPLE;
    this.subject = null;
    this.client = null;
    this.sample = null;
    this.timestamp = null;
  }

  public void setSample(Object sample) {
    this.sample = sample;
  }                                                                            50

  public Object getSample() {
    return this.sample;
  }

  public void setTimestamp(Object timestamp) {
    this.timestamp = timestamp;
  }

  public Object getTimestamp() {                                               60
    return this.timestamp;
  }
}

/**
 * $Log: JiniSignalEntry.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */                                                                            70
```

JiniWorkspaceEntry.java

```
/*
 * Jini Workspace Entry Class.
 * $Id: JiniWorkspaceEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.net.jini;
import robotworld.net.*;

/**
 * A workspace entry for the Jini blackboard.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *                                                                             20
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: JiniWorkspaceEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class JiniWorkspaceEntry extends JiniEntry implements WorkspaceEntry
{
  /**
   * If it's not public, it won't get serialized.
   */
  public Object server, domain, verb, subject, client;                         30

  /**
   * This class must have a no-arg constructor.
   */
  public JiniWorkspaceEntry() {
  }

  public void setServer(Object server) {
    this.server = server;
  }                                                                            40

  public Object getServer() {
    return this.server;
  }

  public void setDomain(Object domain) {
    this.domain = domain;
  }

  public Object getDomain() {                                                  50
    return this.domain;
  }

  public void setVerb(Object verb) {
    this.verb = verb;
  }

  public Object getVerb() {
    return this.verb;
  }                                                                            60

  public void setSubject(Object subject) {
    this.subject = subject;
  }
```

```
  public Object getSubject() {
    return this.subject;
  }

  public void setClient(Object client) {
    this.client = client;
  }

  public Object getClient() {
    return this.client;
  }
}

/**
 * $Log: JiniWorkspaceEntry.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

C.4 The robotworld.ide package

BeanAnalyzer.java

```
/*
 * Bean Analyzer Class.
 * $Id: BeanAnalyzer.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.ide;
import java.lang.reflect.*;

/**
 * Provides a means of reconstructing the signatures of a class's
 * fields, methods, and constructors as string arrays.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: BeanAnalyzer.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 */

public class BeanAnalyzer {
  /**
   * The class under analysis.
   */
  public Class bean;
```

```
/**
 * A string array of signatures of the bean's constructors.
 */
public String[] constructors;

/**
 * A string array of signatures of the bean's fields.
 */
public String[] fields;

/**
 * A string array of signatures of the bean's methods.
 */
public String[] methods;

/**
 * The bean's raw reflected constructors.
 */
public Constructor[] rawConstructors;

/**
 * The bean's raw reflected fields.
 */
public Field[] rawFields;

/**
 * The bean's raw reflected methods.
 */
public Method[] rawMethods;

/**
 * Public no-arg constructor.
 */
public BeanAnalyzer() {
}

/**
 * Sets the class to be analyzed, and analyzes it.
 * Stores the string arrays of signatures in the fields:
 * <code>fields</code>, <code>constructors</code>,
 * and <code>methods</code> of this bean analyzer instance.
 *
 * @param bean the class to analyze.
 */
public void setClass(Class bean) {
  this.bean = bean;

  // get the Fields, Constructors, and Methods of the bean
  this.rawFields       = bean.getDeclaredFields();
  this.rawConstructors = bean.getDeclaredConstructors();
  this.rawMethods      = bean.getDeclaredMethods();

  // get the signatures as strings and store them away
  this.fields       = BeanAnalyzer.getSignatures(this.rawFields);
  this.constructors = BeanAnalyzer.getSignatures(this.rawConstructors);
  this.methods      = BeanAnalyzer.getSignatures(this.rawMethods);
}

/**
 * Returns a string array of signatures for the given members.
 *
 * @param items an array of Members to analyze
 * @return a string array of signatures
 */
public static String[] getSignatures(Member[] items) {
  String[] sigs = new String[items.length];
  for (int i=0; i<items.length; i++) {
    sigs[i]=BeanAnalyzer.signature(items[i]);
```

```
    }
    return sigs;
  }

  /**
   * Returns the signature of the given type as a String.
   * If the Class represents an array, uses reflection to
   * find the component type and generate a string
   * of the form: <br><pre>java.awt.Point[]</pre><br>
   * Otherwise just uses the class's <code>getName</code> method.
   *
   * @param type the type to reflect
   * @return the signature of the given type
   */
  public static String getType(Class type) {
    // is it an array?
    if (type.isArray()) {
      return type.getComponentType().getName() + "[]";
    } else return type.getName();
  }

  /**
   * Returns the signature of the given member as a String.
   * If the member is a <code>Method</code>, uses reflection
   * to reconstruct a method signature of the form:<br><pre>
   * public static java.awt.Point myMethod(java.lang.String, int[])</pre><br>
   * Otherwise just uses the member's <code>getName</code> method.
   *
   * @param member the member to reflect
   * @return the signature of the given member
   */
  public static String signature(Member member) {
    if (member instanceof Method) {

      // generate a space-seperated list of modifiers,
      // followed by the return type and a space
      Method meth = (Method)member;
      String sig = Modifier.toString(meth.getModifiers())+" ";
      sig += BeanAnalyzer.getType(meth.getReturnType())+" ";

        // generate a comma-seperated list of types,
        // enclosed in parenthesis.
      sig += meth.getName()+"(";
      Class[] params = meth.getParameterTypes();
      for (int i=0; i<params.length; i++) {
        sig += BeanAnalyzer.getType(params[i]);
        if (i<params.length-1) sig += ", ";
      }
      sig += ")";
      return sig;

    // otherwise just use Member.getName
    } else return ((Member)member).getName();
  }
}

/**
 * $Log: BeanAnalyzer.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:49  craigh
 * Original Version.
 *
 */
```

CodeEditor.java

```
/*
 * Code Editor Class.
 * $Id: CodeEditor.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                                10
 */

package robotworld.ide;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.JTextArea;
import javax.swing.event.DocumentEvent;
import javax.swing.event.DocumentListener;                                        20

/**
 * A text area which supports brace matching and auto-indent.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: CodeEditor.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 */
                                                                                  30
public class CodeEditor extends JTextArea  {

  // this is a helper class that records a character,
  // and its position in the textfield.
  public class MarkedChar {
    public char ch;
    public int mark;
    public MarkedChar(char ch, int mark) {
      this.ch = ch;
      this.mark = mark;                                                           40
    }
    public char charValue() {
      return this.ch;
    }
    public int markValue() {
     return this.mark;
    }
  }

  // all opens are pushed onto a stack.                                           50
  protected Stack openStack;
  protected MarkedChar openMark;
  protected int unindent;
  protected int unindentRemaining;
  protected int padRemaining;
  protected int indent;
  protected int indentIncr=3;

  // parenthesis, brackets, and braces are recognized.
  public static final int OPEN=0, CLOSE=1;                                        60
  public static final int BRACES=2;
  public static final char NEWLINE = '\n';
  public static final char[][] PARENS = {{ '(', ')' },
                                         { '[', ']' },
```

```
                                { '{', '}' }};

/**
 * Defaults to 24 rows, 40 columns.
 */
public CodeEditor() {                                                    70
  super("", 24, 40);
  this.openStack = new Stack();
  this.addKeyListener(new KeyAdapter() {
    public void keyTyped(KeyEvent e) {
      CodeEditor.this.monitorKeystrokes(e);
      CodeEditor.this.unindent=0;
    }
  });
  this.getDocument().addDocumentListener(new DocumentListener() {
    public void insertUpdate(DocumentEvent e) {                          80
      CodeEditor.this.handleIndent();
      CodeEditor.this.handleUnindent();
      CodeEditor.this.handleCursorFlash();
    }
    public void changedUpdate(DocumentEvent e) {
    }
    public void removeUpdate(DocumentEvent e) {
    }
  });
}                                                                        90

/**
 * Resets the state of the parser.
 */
public void resetParser() {
  this.openStack.removeAllElements();
  this.openMark = null;
  this.indent = 0;
  this.padRemaining = 0;
  this.unindent = 0;                                                    100
  this.unindentRemaining = 0;
}

/**
 * Handles the cursor flash.
 * If there is an open mark, move to the mark and back
 * with a human-detectable pause in between.  Then clear the mark.
 * Otherwise, do nothing.
 */
protected void handleCursorFlash() {                                    110
  // return if nothing to do
  if (this.openMark == null) return;

  // move to the mark, and back
  int caret = this.getCaretPosition();
  this.setCaretPosition(this.openMark.markValue());
  try {
    Thread.sleep(500);
  } catch (InterruptedException x) {
  }                                                                     120
  this.setCaretPosition(caret);

  // clear the mark
  this.openMark = null;
}

/**
 * Handles indent one character at a time.
 * If <code>padRemaining</code> is positive, it gets decremented and
 * a space is inserted.  Note that each inserted space triggers another  130
 * DocumentEvent, which has the effect of calling this method again.
 * <code>padRemaining</code> is decremented
```

```
 */
protected void handleIndent() {
  if (this.padRemaining <= 0) return;
  int caret = this.getCaretPosition();
  if (this.padRemaining == this.indentIncr) {
    this.unindent = caret;
  }
  this.padRemaining--;                                                    140
  this.insert(" ", caret);
}

/**
 * Handles unindent all at once.  If <code>unindentRemaining</code>
 * is positive, removes all characters between it and the caret,
 * and then sets it to zero.
 */
protected void handleUnindent() {
  if (this.unindentRemaining <= 0) return;                                150
  int caret = this.getCaretPosition();
  this.replaceRange("", this.unindentRemaining, caret-1);
  this.unindentRemaining = 0;
  this.setCaretPosition(caret);
}

/**
 * The next document event will cause a flash.
 */
protected void prepareForFlash(MarkedChar mark) {                         160
  this.openMark = mark;
}

/**
 * The next document event will cause an indent.
 */
protected void prepareForIndent() {
  this.padRemaining = this.indent;
}
                                                                          170
/**
 * The next document event will cause an unindent.
 */
protected void prepareForUnindent() {
  this.unindentRemaining = this.unindent;
}

/**
 * Monitors keystrokes.  Responsible for marking opens and pushing them
 * onto the stack; matching closes and popping them off the stack;          180
 * adjusting indentation for open and close braces; indenting on newlines;
 * and beeping if there are mismatched parenthesis or too many closes.
 *
 * @param e a key event
 */
protected void monitorKeystrokes(KeyEvent e) {
  char theKey = e.getKeyChar();

  // for newline, prepare for indent
  if (theKey == NEWLINE) {                                                190
    this.prepareForIndent();
    return;
  }

  // for parenthesis, brackets, and braces
  for (int i=0; i<PARENS.length; i++) {

    // mark and push the opens
    if (theKey == PARENS[i][OPEN]) {
                                                                          200
```

```
        // if braces, adjust the indent amount
        if (i==BRACES) {
          this.indent += this.indentIncr;
        }

        this.openStack.push(
          new MarkedChar(theKey, this.getCaretPosition()));
        return;

      // match and pop the closes                                    210
      } else if (theKey == PARENS[i][CLOSE]) {
        try {

          // if braces, adjust the indent amount
          if (i==BRACES) {
            this.indent -= this.indentIncr;
            if (this.indent < 0) this.indent = 0;
            prepareForUnindent();
          }                                                          220

          // do the parenthesis match?
          MarkedChar lastOpenChar = (MarkedChar)this.openStack.peek();
          if (lastOpenChar.charValue()==PARENS[i][OPEN]) {

            // pop and prepare for flash
            prepareForFlash((MarkedChar)this.openStack.pop());
            return;

          // mismatched parenthesis
          } else {                                                   230
            Toolkit.getDefaultToolkit().beep();
            return;
          }

        // too many close parenthesis.
        } catch (EmptyStackException x) {
          Toolkit.getDefaultToolkit().beep();
          return;
        }
      }                                                              240
    }
  }
}

/**
 * $Log: CodeEditor.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:49  craigh
 * Original Version.
 *
 */                                                                  250
```

CodeEditorWindow.java

```
/*
 * Code Editor Window Class.
 * $Id: CodeEditorWindow.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
```

```
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                          10
 */

package robotworld.ide;
import robotworld.net.*;
import java.awt.*;
import java.util.StringTokenizer;
import java.io.File;
import java.text.MessageFormat;
import java.io.IOException;
import javax.swing.JTextArea;                                               20
import javax.swing.JScrollPane;
import javax.swing.BoxLayout;

/**
 * Provides a code editing window.
 * Extends the UniformWorkspace by adding an option to COMPILE under TOOLS.
 * Consists of two scrollable text areas, side by side.
 * On the left is a CodeEditor, which is editable and set to its default size.
 * On the right is a debugger text area, which is not editable and is
 * initially invisible.  Selecting TOOLS-&gt;COMPILE in the menu has the    30
 * effect of calling this window's <code>compile</code> method.  If any
 * errors are generated, the debugger text area inflates and displays
 * those errors.  The debugger text area is deflated on a successful compile.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: CodeEditorWindow.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 */
                                                                            40
public class CodeEditorWindow extends UniformWorkspace {

  /**
   * Where you will do your typing.
   */
  public CodeEditor editor;   // bare bones

  /**
   * Where you will do your puzzling.
   */                                                                       50
  public JTextArea  debugger; // kind of a lame one at that

  /**
   * Public no-arg constructor.
   */
  public CodeEditorWindow() {
    super();
    this.makeMenuItem(this.toolsMenu, this.COMPILE);
    this.setWorkspaceTitle("CodeEditor");
                                                                            60
    // create two text areas: one for code, one for errors
    this.editor = new CodeEditor();
    this.debugger = new JTextArea(24,0); // initially invisible
    this.debugger.setEditable(false);

    // make them scrollable and side by side
    Container pane = this.getContentPane();
    pane.setLayout(new BoxLayout(pane, BoxLayout.X_AXIS));
    JScrollPane forEditor = new JScrollPane(this.editor);
    JScrollPane forDebugger = new JScrollPane(this.debugger);               70
    pane.add(forEditor);
    pane.add(forDebugger);

    this.pack();
    this.show();
  }
```

```
public void dispatch(String command) {
  if (command.equals(UniformWorkspace.COMPILE))
    this.compile();
  else
    super.dispatch(command);
}

public void configureToEdit(Object o) {
  String classname = o.getClass().getName();

  IDEManager ide = Workspace.getIDEManager();
  EditManager ed = ide.getEditManager();
  if (ed != null) {
    try {
      ed.edit(classname);
      this.setCurrentFilename(
        Workspace.classnameToFilename(classname));
    } catch (IOException e) {
      this.launchExceptionDialog(e);
    }

  } else { // null means you're on your own, buddy
    this.editor.resetParser();
    String text = this.loadTextFrom(
      Workspace.classnameToFilename(classname));
    if (text != null) this.editor.setText(text);
  }
}

public void compile() {
  IDEManager ide = Workspace.getIDEManager();
  CompileManager cm = ide.getCompileManager();
  try {
      // just in case they load a new file.
    String filename = this.getCurrentFilename();
    String classname = Workspace.filenameToClassname(filename);
    cm.compile(classname);
  } catch (IOException e) {
    this.launchExceptionDialog(e);
  } catch (CompilerException e) {
    // expand the debugger window with error text
    this.debugger.setText(e.getMessage());
    this.debugger.setColumns(40);
    this.pack();
    this.show();
    return;
  }

  // remove any previous error text and shrink debugger window
  this.debugger.setColumns(0);
  this.debugger.setText("");
  this.pack();
  this.show();
}

public void clear() {
  this.editor.resetParser();
  this.editor.setText("");
}

public void load() {
  this.editor.resetParser();
  String text = super.loadText();
  if (text != null) this.editor.setText(text);
}

public void save() {
```

```
    super.saveText(this.editor.getText());
  }

  public void saveAs() {
    super.saveTextAs(this.editor.getText());
  }                                                                    150

  public void rename() {
  }

  public void print() {
  }

  public void copy() {
  }
                                                                       160
  public void cut() {
  }

  public void delete() {
  }

  public void paste() {
  }
}
                                                                       170
/**
 * $Log: CodeEditorWindow.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:49  craigh
 * Original Version.
 *
 */
```

CompileManager.java

```
/*
 * Compile Manager Interface.
 * $Id: CompileManager.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                      10
 */

package robotworld.ide;
import java.io.*;

/**
 * The interface for classes that can compile source files on demand
 * at runtime.
 * <p>
 * <em>Note: Requires that the program was started from the codebase    20
 * directory.</em>
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Mike Harder, mharder@mit.edu
 * @version $Id: CompileManager.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
```

```
 */

public interface CompileManager {
                                                                          30
  /**
   * Creates the specified class.  Compiles the appropriate source file,
   * creating the class file in the same directory as the source file.
   * <p>
   * Finds the .java file to compile using the standard Java package hierarchy.
   * For example, to create the class <code>robotworld.ide.MyClass</code>,
   * it tries to compile the file robotworld/ide/MyClass.java.  The directory
   * is relative to the directory in which the program was started.
   *
   * @param fullyQualifiedClassName The fully qualified name of the class.   40
   *          For example, robotworld.ide.MyClass
   * @return the desired Class.
   * @exception CompilerException
   *              If there was an error compiling the specified file.
   *              Calling <code>getMessage</code> on the exception returns
   *              the compilation errors.
   * @exception IOException
   *              If an IO Error occurs.
   */
   public Class compile(String fullyQualifiedClassName)                   50
     throws CompilerException, IOException;
}

/**
 * $Log: CompileManager.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:49  craigh
 * Original Version.
 *
 */
```

CompilerException.java

```
/*
 * Compiler Exception Class.
 * $Id: CompilerException.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                          10
 */

package robotworld.ide;

/**
 * Thrown when a CompileManager cannot compile a given class.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Mike Harder, mharder@mit.edu                                   20
 * @version $Id: CompilerException.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 */

public class CompilerException extends Exception
{
```

```
    /**
     * This exception is born with a message.
     *
     * @param s the compiler error messages
     */
  public CompilerException(String s)
  {
    super(s);
  }
}

/**
 * $Log: CompilerException.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:49  craigh
 * Original Version.
 *
 */
```

EditManager.java

```
/*
 * Edit Manager Interface.
 * $Id: EditManager.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.ide;
import java.io.*;

/**
 * Allows a class file to be loaded into an editor.
 * <p>
 * <em>Note: Requires that the program was started from the codebase
 * directory.</em>
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Mike Harder, mharder@mit.edu
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: EditManager.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 */

public interface EditManager
{
  /**
   * Loads the specified class into an editor.  This might be a window
   * controlled by RobotWorld, or an external application like Emacs.
   * <p>
   * Finds the .java file to load using the standard Java package hierarchy.
   * For example, to edit the class <code>robotworld.ide.MyClass</code>,
   * it tries to load the file robotworld/ide/MyClass.java.  The directory
   * is relative to the directory in which the program was started.
   *
   * @param fullyQualifiedClassName The fully qualified name of the class.
   *          For example, robotworld.ide.MyClass
```

```
     * @exception IOException
     *              If there was an error loading the specified file.
     */
    public void edit(String fullyQualifiedClassName)
        throws IOException;
}

/**
 * $Log: EditManager.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:49  craigh
 * Original Version.
 *
 */
```

ExternalIDEManager.java

```
/*
 * External IDE Manager Class.
 * $Id: ExternalIDEManager.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.ide;
import java.text.MessageFormat;
import java.io.*;

/**
 * Provides managers for interacting with external applications.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Mike Harder, mharder@mit.edu
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: ExternalIDEManager.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 */

public class ExternalIDEManager
    implements IDEManager, EditManager, CompileManager
{
    /**
     * A process exit value indicating success.
     */
    public static int SUCCESSFUL = 0;

    /**
     * Property which allows a command substitution pattern for editing.
     */
    public static String EDIT = "EDIT";

    /**
     * Property which allows a command substitution pattern for compiling.
     */
    public static String COMPILE = "COMPILE";

    /**
```

```
 * Property which allows a command substitution pattern for compiling.
 */
public static String FILE_SEPERATOR = "FILE_SEPERATOR";

/**
 * Cache the command substitution patterns in memory.
 */
protected String editPattern, compilePattern;

/**
 * Custom file seperators are sometimes necessary.
 */
protected String fileSeperator;

/**
 * Public no-arg constructor.  Loads the EDIT and COMPILE
 * command substitution patterns now, to save time.
 */
public ExternalIDEManager()
{
  // Looks up the command substitution strings
  // in the System properties -- ahead of time
  String name = this.getClass().getName();
  this.editPattern = System.getProperty(name + "." + EDIT);
  this.compilePattern = System.getProperty(name + "." + COMPILE);
  this.fileSeperator = System.getProperty(name + "." + FILE_SEPERATOR);
}

/**
 * Returns an EditManager implementation.
 * If an EDIT pattern was found, returns a reference to itself.
 * Otherwise, returns null.
 *
 * @return an edit manager implementation.
 */
public EditManager getEditManager() {
  return (this.editPattern != null) ? this : null;
}

/**
 * Returns an CompileManager implementation.
 * If a COMPILE pattern was found, returns a reference to itself.
 * Otherwise, returns null.
 *
 * @return a compile manager implementation.
 */
public CompileManager getCompileManager() {
  return (this.compilePattern != null) ? this : null;
}

/**
 * Ask for a standard external service (such as COMPILE or EDIT).
 * Given a command substitution pattern and a list of arguments,
 * it performs the substitution, and then spawns off the external process.
 * If the process completes successfully, null is returned.  Otherwise,
 * a string will be returned containing all characters that were printed
 * to the error stream.
 *
 * @param pattern the command substitution pattern
 * @param arguments an array of strings to be substituted into the pattern.
 * @return a string representation of the error stream, or null if okay.
 */
public String shellExec(String pattern, String[] arguments)
  throws IOException
{
  // this operates in much the same manner as the C function sprintf().
  String shellCommand = MessageFormat.format(pattern, arguments);
```

```
  // Throws IOException
  Process process = Runtime.getRuntime().exec(shellCommand);

  // We cannot simply use MyProc.waitFor(). If the process sends a lot of
  // text to its error stream, we would have a deadlock situation.
  // Here is the description from the Java Developers Connection:
  //                                                                    120
  // The main program waits for the subprocess to terminate
  // before it attempts to read its output. If a small amount of output
  // is written, it is stored in the pipe, and the subprocess can terminate,
  // allowing the data in the pipe to be read and printed. If the pipe
  // fills, however, the subprocess blocks. There is then a deadlock,
  // because the subprocess cannot terminate until someone empties the
  // pipe, but this cannot happen unless the process terminates so the
  // waitFor can return.
  //
  // So we must remove the characters from the pipe while the subprocess is  130
  // running. That is the reason for the while(!done) loop below.

  BufferedReader d = new BufferedReader(new InputStreamReader(
    process.getErrorStream()));
  StringBuffer errors = new StringBuffer();
  // need to give returnValue a dummy value to avoid a compile error.
  // This value will always be reassigned before it is compared to
  // SUCCESSFUL.
  int returnValue = -1;
  boolean done = false;                                                 140

  while (!done) {
    while (d.ready()) {
      errors.append((char) d.read());
       }
    try {
      returnValue = process.exitValue();
      done = true;
    } catch (IllegalThreadStateException e) {
      // The process is not done yet. Wait for 500 ms to save some cpu    150
      try {
        Thread.sleep(500);
         } catch (InterruptedException ie) {}
       }
  }
  d.close();

  // success or failure?
  if (returnValue == SUCCESSFUL) {
    return null;                                                        160
  } else {
    return errors.toString();
  }
}

public void edit(String fullyQualifiedClassName)
throws IOException
{
  String filename = Workspace.classnameToFilename(
    fullyQualifiedClassName, this.fileSeperator);                       170
  // Performs the substitution, and then spawns off the external process.
  this.shellExec(this.editPattern, new String[]{filename});
}

public Class compile(String fullyQualifiedClassName)
throws IOException, CompilerException
{
  String filename = Workspace.classnameToFilename(
    fullyQualifiedClassName, this.fileSeperator);
                                                                        180
  // Performs the substitution, and then spawns off the external process.
```

```
    String errors = this.shellExec(
      this.compilePattern, new String[]{filename});

    if (errors == null) { // SUCCESSFUL
      try {
        return Class.forName(fullyQualifiedClassName);
      } catch (ClassNotFoundException cnfe) {
        // This should never happen since we just created the .class file
        // If it did I would consider it a "compiler error", so the
            // appropriate action is to throw a CompilerException
        throw new CompilerException(
          "Couldn't find the class we just compiled");
      }
    } else {
      throw new CompilerException(errors);
    }
  }
}

/**
 * $Log: ExternalIDEManager.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:49  craigh
 * Original Version.
 *
 */
```

IDEManager.java

```
/*
 * IDE Manager Interface.
 * $Id: IDEManager.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.ide;

/**
 * Provides access to an EditManager and a CompileManager.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: IDEManager.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 */

public interface IDEManager
{
  /**
   * Returns an EditManager implementation.
   *
   * @return an edit manager implementation.
   */
  public abstract EditManager    getEditManager();

  /**
```

```
    * Returns an CompileManager implementation.
    *
    * @return a compile manager implementation.
    */
   public abstract CompileManager getCompileManager();
}

/**
 * $Log: IDEManager.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:49  craigh
 * Original Version.
 *
 */
```

UniformWorkspace.java

```
/*
 * Uniform Workspace Abstract Class.
 * $Id: UniformWorkspace.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.ide;

import java.awt.*;
import java.awt.event.*;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import javax.swing.JMenuBar;

/**
 * Provides a uniform menu system for RobotWorld.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: UniformWorkspace.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public abstract class UniformWorkspace extends Workspace
implements ActionListener {

  // The five menus.
  public static final String FILE = "File";
  public static final String EDIT = "Edit";
  public static final String VIEW = "View";
  public static final String TOOLS = "Tools";
  public static final String HELP = "Help";

  // The FILE menu items.
  public static final String CLEAR = "Clear";
  public static final String LOAD = "Load";
  public static final String SAVE = "Save";
  public static final String SAVE_AS = "Save As";
  public static final String RENAME = "Rename";
```

```
public static final String PRINT = "Print";
public static final String EXIT = "Exit";

// The EDIT menu items.
public static final String COPY = "Copy";
public static final String CUT = "Cut";
public static final String PASTE = "Paste";
public static final String DELETE = "Delete";

// The VIEW menu items.
public static final String CODE = "Code";
public static final String BEHAVIOR = "Behavior";
public static final String MORPHOLOGY = "Morphology";
public static final String ENVIRONMENT = "Environment";
public static final String ARCHITECTURE = "Architecture";

// The TOOLS menu items.
public static final String BEANBAG = "Beanbag";
public static final String MONITOR = "Monitor";
public static final String COMPILE = "Compile";
public static final String PROPERTIES = "Properties";

// The HELP menu items.
public static final String ABOUT = "About_RobotWorld";
public static final String JAVA_REF = "Java_Reference";
public static final String TEXTBOOK = "Textbook";
public static final String MANUAL = "Users_Manual";

protected JMenu fileMenu, editMenu, viewMenu, toolsMenu, helpMenu;
protected JMenuBar menuBar;

/**
 * Makes a menu item, enables it for dispatch, and adds it
 * to the given menu.
 *
 * @param menu the menu to add the new menu item to.
 * @param name the action command string for this item.
 */
protected void makeMenuItem(JMenu menu, String name) {
  JMenuItem item = new JMenuItem(name);
  item.setActionCommand(name);
  item.addActionListener(this);
  menu.add(item);
}

public void actionPerformed(ActionEvent e) {
  String command = e.getActionCommand();
  this.dispatch(command);
}

/**
 * Conditionally dispatches the given action command.
 * If the command is recognized, calls the appropriate workspace method.
 * Otherwise, no action is taken.
 *
 * @param command the action to be performed.
 */
public void dispatch(String command) {
  if (command.equals(CLEAR)) this.clear();
  else if (command.equals(LOAD)) this.load();
  else if (command.equals(SAVE)) this.save();
  else if (command.equals(SAVE_AS)) this.saveAs();
  else if (command.equals(RENAME)) this.rename();
  else if (command.equals(PRINT)) this.print();
  else if (command.equals(EXIT)) this.exit();
  else if (command.equals(DELETE)) this.delete();
  else if (command.equals(ABOUT))
    this.browserDispatch(command, 300, 200);
```

```
    else if (command.equals(JAVA_REF))
      this.browserDispatch(command, 600, 400);
    else if (command.equals(TEXTBOOK))
      this.browserDispatch(command, 600, 400);
    else if (command.equals(MANUAL))
      this.browserDispatch(command, 600, 400);
  }

  /**
   * Exits the RobotWorld application.
   */
  public void exit() {
    super.close();
    System.exit(0);
  }

  /**
   * Dispatches a menu command by browsing content.
   * Looks up the UniformWorkspace property with the same name
   * as the command, interprets the property as a URL, and
   * calls <code>browseContent</code> with the resulting URL,
   * width, and height.
   *
   * @param command the action command.
   * @param w the preferred width of the browser.
   * @param h the preferred height of the browser.
   */
  protected void browserDispatch(String command, int w, int h) {
    String url = System.getProperty(
      UniformWorkspace.class.getName() + "." + command);
    if (url != null) {
      this.browseContent(command, url, w, h);
    }
  }

  /**
   * A public no-argument constructor is essential.
   * Constructs the menu bar with five menus: FILE, EDIT, VIEW, TOOLS, HELP.
   */
  public UniformWorkspace() {

    // FILE: clear, load, save/saveAs, rename, print, exit
    this.fileMenu = new JMenu(FILE);
    makeMenuItem(fileMenu, CLEAR);
    makeMenuItem(fileMenu, LOAD);
    makeMenuItem(fileMenu, SAVE);
    makeMenuItem(fileMenu, SAVE_AS);
    makeMenuItem(fileMenu, RENAME);
    makeMenuItem(fileMenu, PRINT);
    makeMenuItem(fileMenu, EXIT);

    // EDIT: copy, cut, paste, delete
    this.editMenu = new JMenu(EDIT);
    makeMenuItem(editMenu, COPY);
    makeMenuItem(editMenu, CUT);
    makeMenuItem(editMenu, PASTE);
    makeMenuItem(editMenu, DELETE);

    // VIEW: code, behavior, morphology, environment, architecture
    this.viewMenu = new JMenu(VIEW);
    makeMenuItem(viewMenu, CODE);
    makeMenuItem(viewMenu, BEHAVIOR);
    makeMenuItem(viewMenu, MORPHOLOGY);
    makeMenuItem(viewMenu, ENVIRONMENT);
    makeMenuItem(viewMenu, ARCHITECTURE);

    // TOOLS: beanbag
    this.toolsMenu = new JMenu(TOOLS);
```

```
    makeMenuItem(toolsMenu, BEANBAG);

    // HELP: about, Java Reference, online class textbook, users manual
    this.helpMenu = new JMenu(HELP);
    makeMenuItem(helpMenu, ABOUT);
    makeMenuItem(helpMenu, JAVA_REF);
    makeMenuItem(helpMenu, TEXTBOOK);
    makeMenuItem(helpMenu, MANUAL);

    // Add them all to the menu bar.
    this.menuBar = new JMenuBar();
    this.menuBar.add(this.fileMenu);
    this.menuBar.add(this.editMenu);
    this.menuBar.add(this.viewMenu);
    this.menuBar.add(this.toolsMenu);
    this.menuBar.add(this.helpMenu);
    this.setJMenuBar(this.menuBar);
  }
}

/**
 * $Log: UniformWorkspace.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

Workspace.java

```
/*
 * Workspace Abstract Class.
 * $Id: Workspace.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.ide;
import robotworld.net.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.JFrame;
import javax.swing.JFileChooser;
import javax.swing.JLabel;
import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JEditorPane;
import javax.swing.JScrollPane;
import javax.swing.DefaultListModel;
import javax.swing.event.HyperlinkEvent;
import javax.swing.event.HyperlinkListener;
import javax.swing.text.html.HTMLFrameHyperlinkEvent;
import javax.swing.text.html.HTMLDocument;

/**
```

```
 * Provides utilites for RobotWorld in general and frames in particular.
 * Static methods provide functionality for RobotWorld at large.
 * Instance methods provide functionality for frame instances.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @author  Mike Harder, mharder@mit.edu                                    40
 * @version $Id: Workspace.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 */

public abstract class Workspace extends JFrame
{

  /**
   * The master registry for workspaces.  Allows the last workspace
   *  to call <code>System.exit(0)</code> when closing down.
   */                                                                       50
  protected static Vector workspaceList = new Vector();

  /**
   * The filename (if any) to display in the title bar.
   */
  protected String currentFilename;

  /**
   * The editor name to display in the title bar.  For example, "Code Editor".
   */                                                                       60
  protected String workspaceTitle;

  /**
   * The blackboard singleton.
   */
  protected static Blackboard blackboard;

  /**
   * The IDE Manager singleton.
   */                                                                       70
  protected static IDEManager ideManager;

  /**
   * Remembers the directory you last visited.
   */
  protected static JFileChooser fileChooser = new JFileChooser();

  /**
   * Maps interfaces to implementations.
   */                                                                       80
  protected static HashMap implMap = new HashMap();

  /**
   * Maps sprites to layers.
   */
  protected static HashMap layerMap = new HashMap();

  /**
   * Maps layers to beanbags.
   */                                                                       90
  protected static HashMap beanbagMap = new HashMap();

  /**
   * A public no-arg construcor is essential.
   * Each workspace is born with a window listener that knows how
   * to properly shut down using <code>close</code>.  Also,
   * the workspace adds itself to the master registry.
   */
  public Workspace() {
    this.addWindowListener(new WindowAdapter() {                            100
```

```
      public void windowClosing(WindowEvent w) {
        Workspace.this.close();
      }
    });
    this.workspaceList.addElement(this);
}

/**
 * Returns an instance of a class that implements the given interface.
 * Looks up the system property corresponding to the fully-qualified
 * package name of the given interface.  Loads and instantiates the class
 * to which it is bound.  This method assumes that a properties file
 * has already been loaded, with entries of the form:<br><pre>
 *    package.name.of.given.interface=package.name.of.implementor</pre>
 *
 * @param givenInterface the interface or abstract class to look up.
 * @return an instance of a class that implements the given interface.
 * @exception ClassNotFoundException
 *            if the class to be loaded couldn't be found.
 * @exception InstantiationException
 *            found an interface or abstract class instead.
 * @exception IllegalAccessException
 *            if a public no-argument constructor couldn't be found.
 */
public static Object produce(Class givenInterface)
    throws ClassNotFoundException,
           InstantiationException, IllegalAccessException {

    // optimization: cache the impls in a HashMap, keyed off givenInterface
    Class impl = (Class)Workspace.implMap.get(givenInterface);
    if (impl == null) {

      String propertyName = givenInterface.getName();
      String implName = System.getProperty(propertyName);
      impl = Class.forName(implName);
      Workspace.implMap.put(givenInterface, impl);
    }

    return impl.newInstance();
}

/**
 * Gets the blackboard singleton, instantiating if necessary.
 * "Singleton" means there is only one such object in memory.
 *
 * @return the blackboard singleton.
 */
public static Blackboard getBlackboard() {
    if (Workspace.blackboard == null) {
      try {
        Workspace.blackboard = (Blackboard)Workspace.produce(
          Blackboard.class);
      } catch (Exception e) {
        e.printStackTrace();
        return null;
      }
    }
    return Workspace.blackboard;
}

/**
 * Gets the IDEManager singleton, instantiating if necessary.
 * "Singleton" means there is only one such object in memory.
 *
 * @return the IDEManager singleton.
 */
public static IDEManager getIDEManager() {
    if (Workspace.ideManager == null) {
```

```
    try {
      Workspace.ideManager = (IDEManager)Workspace.produce(
        IDEManager.class);
    } catch (Exception e) {
      e.printStackTrace();
      return null;
    }
  }
  return Workspace.ideManager;
}

/**
 * Sets the editor name to be displayed in the title bar.
 *
 * @param name the string to be displayed.
 */
protected void setWorkspaceTitle(String name) {
  this.workspaceTitle = name;
  this.setTitle(this.workspaceTitle);
}

/**
 * Sets the filename to be displayed in the title bar.
 *
 * @param name the string to be displayed.
 */
protected void setCurrentFilename(String name) {
  if (name == null) return;
  this.currentFilename = name;
  this.setTitle(this.workspaceTitle + "  :  " + name);
}

/**
 * Gets the name being displayed in the title bar.
 *
 * @return the string being displayed.
 */
protected String getCurrentFilename() {
  return this.currentFilename;
}

/**
 * Configures this workspace to edit the given object.
 * After a workspace is instantiated, this method is used to indicate
 * what the empty space should be filled with: text, graphics, you name it.
 *
 * @param object the object to be edited.
 */
public void configureToEdit(Object object) {
}

/**
 * Returns an instance of the layer class for the given sprite class.
 * Used to decide which kind of workspace should be instantiated
 * in order to edit a given class of sprite.  After calling this method,
 * one could call the resulting workspace's <code>configureToEdit</code>
 * method, and pass it the instance of the sprite to be edited.
 * <p>
 * Looks up the system property corresponding to the fully-qualified
 * package name of the given sprite class.  Loads and instantiates the
 * workspace class to which it is bound.  This method assumes
 * that a properties file has already been loaded, with entries
 * of the form:<br><pre>
 *    package.name.of.given.sprite=package.name.of.layer</pre>
 * <p>
 * NOTES: a layer is just a Workspace that knows how to edit
 * a given sprite.  "Sprite" here is meant in the loose sense of
 * "anything that might want to be edited by double clicking on it."
```

```
 *
 * @param sprite the class to look up.
 * @return an instance of the layer class for the given sprite class.
 * @exception ClassNotFoundException                                    240
 *            if the class to be loaded couldn't be found.
 * @exception InstantiationException
 *            found an interface or abstract class instead.
 * @exception IllegalAccessException
 *            if a public no-argument constructor couldn't be found.
 */
public Workspace getLayerFor(Class sprite)
  throws ClassNotFoundException,
  InstantiationException, IllegalAccessException {
                                                                        250
  // optimization: cache the layers in a HashMap, keyed off sprite
  Class layer = (Class)Workspace.layerMap.get(sprite);
  if (layer == null) {
    Class current = null;
    String layerName = null;

    // my Java is beginning to get curiouser & curiouser.
    // This monstrous statement is a polymorphous property lookup.
    // It starts at the bottom of the inheritance tree for this class,
    // and continues making lookups until either finds a non-null property,   260
    // or it walks off the top of the tree.
    for (current = sprite;
         (current != null) &&
         ((layerName = System.getProperty(current.getName())) == null);
         current = current.getSuperclass()
        ) ;

    if (current != null) { // then we found a mapping.
      layer = Class.forName(layerName);
      Workspace.layerMap.put(sprite, layer);                            270

      } else { // we walked off the top of the tree.
      throw new ClassNotFoundException(
        "Layer property for "+sprite.getName()+
        "not found");
      }
  }

  return (Workspace)layer.newInstance();
}                                                                       280

/**
 * Returns the bean bag object for the given class of layer.
 * The bean bag is a default list model containing a list of strings.
 * These strings are the fully-qualified package names of sprites
 * which are allowed to be instantiated on the given type of layer.
 * <p>
 * Looks up the system property corresponding to the fully-qualified
 * package name of the given layer.  Parses the property as a list of
 * strings, adds them to a default list model, and returns the model.      290
 * This method assumes that a properties file has already been loaded,
 * with entries of the form:<br><pre>
 *    package.name.of.given.layer = \
 *       package.name.of.bean1,     \
 *       package.name.of.bean2,     \
 *       ...,                       \
 *       package.name.of.beanN</pre>
 * <p>
 * The list items must be seperated by whitespace and/or commas in the file.
 * The proper use of backslashes for line continuation is described        300
 * under <code>java.util.Properties.load</code>.
 *
 * @param layer the class of layer to look up.
 * @return the default list model of beans for that layer.
```

```
 */
public static DefaultListModel getBeanbagForLayer(Class layer) {

  // optimization: cache the models in a HashMap, keyed off layer
  DefaultListModel model = null;
  model = (DefaultListModel)Workspace.beanbagMap.get(layer);                    310
  if (model == null) {

    String propertyName = layer.getName();
    String modelItems = System.getProperty(propertyName);
    model = new DefaultListModel();
    StringTokenizer tzer = new StringTokenizer(modelItems, " \n\r\t,");
    while (tzer.hasMoreTokens()) {
      model.addElement(tzer.nextToken());
    }
    Workspace.beanbagMap.put(layer, model);                                     320
  }

  return model;
}

/**
 * Provides a standard window closing behavior for RobotWorld.
 * Removes this workspace from the master registry, and frees up any
 * system resources by calling <code>dispose</code>.  If the master
 * registry is now empty, it also calls <code>System.exit(0)</code> to          330
 * terminate any rogue threads still active elsewhere in the application.
 */
public void close() {
  this.workspaceList.removeElement(this);
  this.dispose();
  if (this.workspaceList.isEmpty()) {
    System.exit(0);
  }
}
                                                                              340
/**
 * Clears the workspace.
 */
public abstract void clear();

/**
 * Loads previously saved work into the workspace.
 */
public abstract void load();
                                                                              350
/**
 * Saves the contents of the workspace under the current filename.
 */
public abstract void save();

/**
 * Saves the contents of the workspace under a new filename.
 */
public abstract void saveAs();
                                                                              360
/**
 * Renames the contents of the workspace.
 */
public abstract void rename();

/**
 * Prints the contents of the workspace.
 */
public abstract void print();
                                                                              370
/**
 * Copies the contents of the workspace to the clipboard.
```

```
 */
public abstract void copy();

/**
 * Cuts the contents of the workspace to the clipboard.
 */
public abstract void cut();
                                                                                380
/**
 * Deletes the contents of the workspace.
 */
public abstract void delete();

/**
 * Pastes the clipboard contents into the workspace.
 */
public abstract void paste();
                                                                                390
/**
 * Opens the current file and serializes the given object to it.
 * Launches an exception dialog if anything goes wrong.
 *
 * @param obj an object to serialize.
 */
public void saveObject(Object obj) {
  if (this.currentFilename != null) {
    try {
      FileOutputStream fos = new FileOutputStream(this.currentFilename);        400
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(obj);
      oos.flush();
      oos.close();
    } catch (Exception e) {
      this.launchExceptionDialog(e);
    }
  }
}
                                                                                410
/**
 * Opens a dialog allowing the user to choose what filename to save the
 * given object as.  Stores the filename the user chooses, and then calls
 * <code>saveObject</code> with the given object.
 *
 * @param obj an object to serialize.
 */
public void saveObjectAs(Object obj) {
  this.fileChooser.showSaveDialog(this);
  this.setCurrentFilename(this.fileChooser.getSelectedFile().getName());        420
  this.saveObject(obj);
}

/**
 * Opens the given object file and returns the reconstituted object.
 * Launches an exception dialog if anything goes wrong.
 *
 * @param filename the file to load.
 * @return the object reconstituted from the file.
 */                                                                             430
public Object loadObjectFrom(String filename) {
  this.setCurrentFilename(filename);

  Object obj = null;
  if (this.currentFilename != null) {
    try {
      FileInputStream fos = new FileInputStream(this.currentFilename);
      ObjectInputStream ois = new ObjectInputStream(fos);
      obj = ois.readObject();
      ois.close();                                                              440
```

```
    } catch (Exception e) {
      this.launchExceptionDialog(e);
    }
  }
  return obj;
}

/**
 * Opens a dialog allowing the user to choose an object file to load.
 * Passes the filename the user chooses to <code>loadObjectFrom</code>,
 * and returns the result.
 *
 * @return the object reconstituted from the file.
 */
public Object loadObject() {
  this.fileChooser.showOpenDialog(this);
  String filename = this.fileChooser.getSelectedFile().getName();
  return this.loadObjectFrom(filename);
}

/**
 * Opens the current file and saves the given text to it.
 * Launches an exception dialog if anything goes wrong.
 *
 * @param text a string containing the entire contents of the text file.
 */
public void saveText(String text) {
  if (this.currentFilename != null) {
    try {
      FileOutputStream fos = new FileOutputStream(this.currentFilename);
      OutputStreamWriter osw = new OutputStreamWriter(fos);
      PrintWriter pw = new PrintWriter(osw);
      pw.println(text);
      pw.flush();
      pw.close();
    } catch (Exception e) {
      this.launchExceptionDialog(e);
    }
  }
}

/**
 * Opens a dialog allowing the user to choose what filename to save the
 * given text as.  Stores the filename the user chooses, and then calls
 * <code>saveText</code> with the given text.
 *
 * @param a string containing the entire contents of the text file.
 */
public void saveTextAs(String text) {
  this.fileChooser.showSaveDialog(this);
  this.setCurrentFilename(this.fileChooser.getSelectedFile().getName());
  this.saveText(text);
}

/**
 * Opens the given text file and returns the entire contents as a String.
 * Launches an exception dialog if anything goes wrong.
 *
 * @param filename the file to load.
 * @return a string containing the entire contents of the text file.
 */
public String loadTextFrom(String filename) {
  this.setCurrentFilename(filename);

  String text = "";
  if (this.currentFilename != null) {
    try {
      FileInputStream fos = new FileInputStream(this.currentFilename);
```

```
        InputStreamReader isr = new InputStreamReader(fos);
        BufferedReader br = new BufferedReader(isr);
        // READ ALL THE LINES PLEASE
        String line = br.readLine();
        if (line != null) {
          text += line;
          while ((line = br.readLine()) != null) {
            text += "\n"+line;
          }
        }
        br.close();
        } catch (Exception e) {
        this.launchExceptionDialog(e);
      }
    }
    return text;
}

/**
 * Opens a dialog allowing the user to choose a text file to load.
 * Passes the filename the user chooses to <code>loadTextFrom</code>,
 * and returns the result.
 *
 * @return a string containing the entire contents of the text file.
 */
public String loadText() {
    this.fileChooser.showOpenDialog(this);
    String filename = this.fileChooser.getSelectedFile().getName();
    return this.loadTextFrom(filename);
}

/**
 * Launch an dialog alerting the user to the given exception.
 * The message string of the exception appears in the dialog, along with
 * an "Ok" button which allows the user to click it away.
 *
 * @param x the exception to display in the dialog.
 */
public void launchExceptionDialog(Exception x) {
    final JDialog dia = new JDialog(this, "Error", true);
    JLabel label = new JLabel(x.toString());
    JButton button = new JButton("Ok");
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        dia.dispose();
      }
    });
    dia.getContentPane().add("North", label);
    dia.getContentPane().add("Center", button);
    dia.pack();
    dia.show();
    x.printStackTrace();
}

/**
 * Loads the given .properties file into the System properties.
 * Inspired from Java in a Nutshell, 2nd ed., Chapter 14, p237.
 *
 * @param propsfilename the .properties file to load.
 * @return the resulting Properties object.
 * @exception IOException
 *              If anything goes wrong.
 */
public static Properties loadProperties(String propsfilename)
throws IOException {

    // Create a new Properties object with system props as its parent.
    Properties props = new Properties(System.getProperties());
```

```
    // Load a file of properties into it.  We may get an exception here...
    props.load(new BufferedInputStream(new FileInputStream(propsfilename)));

    // Set these new combined properties as the system properties.
    System.setProperties(props);

    // in case we wish to edit and save at a later time
    return props;
  }

  /**
   * Attempts to load all given property files in command line order.
   * Each of the String arguments must be the base name for a properties
   * file, without the .properties suffix.
   *
   * @param args an array of strings indicating what .property files to load.
   */
  public static void loadProperties(String[] args) {
    String propsfilename = null;
    for (int i=0; i<args.length; i++) {
      try {
        propsfilename = args[i]+".properties";
        Workspace.loadProperties(propsfilename);
      } catch (IOException e) {
        System.err.println(e+"\n...Unable to read:  "+propsfilename);
      }
    }
  }

  /**
   * Returns a String that is the location of the .java file
   * corresponding to the given class name, relative to the codebase.
   * The class name has to be a fully qualified.
   * Uses the platform independent <code>File.separatorChar</code>.
   *
   * @param fullyQualifiedClassName for example, robotworld.ide.Workspace
   * @return a string which is the location of the .java file
   */
  public static String classnameToFilename (String fullyQualifiedClassName) {
    return fullyQualifiedClassName.replace(
      '.', File.separatorChar) + ".java";
  }

  /**
   * Returns a String that is the location of the .java file
   * corresponding to the given class name, relative to the codebase.
   * The class name has to be fully qualified.  Uses the custom
   * fileSeparator string for use with external applications.
   * If the given <code>fileSeperator</code> argument is null,
   * uses the platform independent <code>File.separatorChar</code> instead.
   *
   * @param fullyQualifiedClassName for example, robotworld.ide.Workspace
   * @param fileSeperator a custom file seperator
   * @return a string which is the location of the .java file
   */
  public static String classnameToFilename (
    String fullyQualifiedClassName, String fileSeperator) {

    if (fileSeperator==null) {
      return Workspace.classnameToFilename(fullyQualifiedClassName);
    }

    StringTokenizer t = new StringTokenizer(fullyQualifiedClassName,".");
    String result = "";
    while (t.hasMoreTokens()) {
      result += t.nextToken() + (t.hasMoreTokens() ? fileSeperator : "");
    }
```

```
  return result + ".java";
}

/**
 * Returns a String that is the fully qualified class name for the
 * given .java file.  The filename must be relative to the codebase.
 * Uses the platform independent <code>File.separatorChar</code>.
 *
 * @param filenameDotJava for example, robotworld/ide/Workspace.java
 * @return a string which is the fully qualified class name
 */
public static String filenameToClassname (String filenameDotJava) {
  String prefix = filenameDotJava.substring(0,filenameDotJava.length()-5);
  return prefix.replace(File.separatorChar, '.');
}

/**
 * Pops up a window to browse the given URL.
 * Launches an exception dialog if anything goes wrong.
 *
 * @param frameTitle the title to appear in the frame.
 * @param url the url to visit.
 * @param w the preferred width of the window
 * @param h the preferred height of the window
 */
public void browseContent(String frameTitle, String url, int w, int h) {
  JFrame frame = new JFrame(frameTitle);
  frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
  try {
    final JEditorPane browser = new JEditorPane();
    browser.setEditable(false); // otherwise you can't click on the links.
    browser.setPage(url);
    JScrollPane scroll = new JScrollPane(browser);
    scroll.setPreferredSize(new Dimension(w,h));
    frame.getContentPane().add("Center", scroll);
    browser.addHyperlinkListener(new HyperlinkListener() {

      // this incantation came from the JDK 1.2 API for JEditorPane.
        public void hyperlinkUpdate(HyperlinkEvent e) {

        // if the link was activated
        if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED) {
          JEditorPane pane = (JEditorPane)e.getSource();

          // if it was a frame event, do frame type stuff
          if (e instanceof HTMLFrameHyperlinkEvent) {
            HTMLFrameHyperlinkEvent evt = (HTMLFrameHyperlinkEvent)e;
            HTMLDocument doc = (HTMLDocument)pane.getDocument();
            doc.processHTMLFrameHyperlinkEvent(evt);

          // otherwise just blindly load the url
                } else {
            try {
              browser.setPage(e.getURL());

            // launch an exception dialog if anything goes wrong
            } catch (IOException x) {
              Workspace.this.launchExceptionDialog(x);
                }
              }
          }
        }
      });

    // todaa!
    frame.pack();
    frame.show();
```

```
      // launch an exception dialog if anything goes wrong
      } catch (IOException x) {
        this.launchExceptionDialog(x);
      }
    }
}

/**                                                                    720
 * $Log: Workspace.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:49  craigh
 * Original Version.
 *
 */
```

C.5 The robotworld.gui package

Connectable.java

```
/*
 * Connectable Interface.
 * $Id: Connectable.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                      10
 */

package robotworld.gui;

/**
 * Provides a marker interface for objects wanting to play the role
 * of an intermediary object in a connection operation.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *                                                                     20
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: Connectable.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public interface Connectable
{
}

/**
 * $Log: Connectable.java,v $                                          30
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

ConnectorSprite.java

```
/*
 * Connector Sprite Class.
 * $Id: ConnectorSprite.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.gui;
import java.awt.*;

/**
 * A connectable that draws an arrow between the closest edges of
 * its source and destination sprites.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.                   20
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: ConnectorSprite.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class ConnectorSprite extends Sprite implements Connectable {

  /**
   * Where to start the arrow from.
   */                                                                          30
  protected Sprite source;

  /**
   * Where to end the arrow.
   */
  protected Sprite destination;

  /**
   * Public no-argument constructor.
   */                                                                          40
  public ConnectorSprite () {
  }

  public String toString() {
    String s = super.toString();
    return s+"    Source:  "+this.source.getClass().getName()+
          "\n   Destin:  "+this.destination.getClass().getName()+
          "\n   Parent:  "+this.parent.getClass().getName()+"\n";
  }
                                                                               50
  /**
   * Sets the source sprite.
   *
   * @param src the source sprite.
   */
  public void setSource(Sprite src) {
    this.source = src;
  }

  /**                                                                          60
   * Sets the destination sprite.
   *
   * @param dst the destination sprite
   */
```

```
public void setDestination(Sprite dst) {
  this.destination = dst;
}

/**
 * Gets the source sprite.                                          70
 *
 * @return the source sprite.
 */
public Sprite getSource() {
  return this.source;
}

/**
 * Gets the destination sprite.
 *                                                                  80
 * @return the destination sprite.
 */
public Sprite getDestination() {
  return this.destination;
}

public void paint(Graphics g) {
  // draw an arrow between the closest edges of
  // the source and destination sprites.
  Rectangle srcBounds = null;                                       90
  Rectangle dstBounds = null;

  if (this.source == null ||
      this.destination == null) {
    return;
  }

  // get the bounds of each sprite
  srcBounds = this.source.getBounds();
  dstBounds = this.destination.getBounds();                        100

  if (srcBounds == null
      || dstBounds == null) {
    return;
  }

  int srcX=0, srcY=0, dstX=0, dstY=0;
  int minX=0, minY=0, maxX=0, maxY=0;

  // if dest is completely to the left, right, or center           110
  // of source, set the X variables accordingly.
  if (dstBounds.x > (srcBounds.x + srcBounds.width)) {
    srcX = srcBounds.x + srcBounds.width;
    dstX = dstBounds.x;

  } else if (srcBounds.x > (dstBounds.x + dstBounds.width)) {
    srcX = srcBounds.x;
    dstX = dstBounds.x + dstBounds.width;

  } else {                                                         120
    srcX = srcBounds.x+srcBounds.width/2;
    dstX = dstBounds.x+dstBounds.width/2;
  }

  // if dest is completely below, above, or center
  // of source, set the Y variables accordingly.
  if (dstBounds.y > (srcBounds.y + srcBounds.height)) {
    srcY = srcBounds.y + srcBounds.height;
    dstY = dstBounds.y;
                                                                   130
  } else if (srcBounds.y > (dstBounds.y + dstBounds.height)) {
    srcY = srcBounds.y;
```

```
        dstY = dstBounds.y + dstBounds.height;

      } else {
        srcY = srcBounds.y+srcBounds.height/2;
        dstY = dstBounds.y+dstBounds.height/2;
      }

      // setup for the barbs of the arrow.                                    140
      int vecX = dstX-srcX;
      int vecY = dstY-srcY;
      double vecLength = Math.sqrt(vecX*vecX+vecY*vecY);
      double arrowX = -10 * vecX/vecLength;
      double arrowY = -10 * vecY/vecLength;
      int barbAX = (int)(dstX+arrowX-arrowY/2.0d);
      int barbAY = (int)(dstY+arrowY+arrowX/2.0d);
      int barbBX = (int)(dstX+arrowX+arrowY/2.0d);
      int barbBY = (int)(dstY+arrowY-arrowX/2.0d);
                                                                           150
      // now draw the shaft and barbs of the arrow.
      g.setColor(selected ? Color.green : Color.blue);
      g.drawLine(srcX, srcY, dstX, dstY);
      g.drawLine(barbAX, barbAY, dstX, dstY);
      g.drawLine(barbBX, barbBY, dstX, dstY);

      // reset the arrow's bounds.
      minX = Math.min(srcX, dstX);
      minX = Math.min(minX, barbAX);
      minX = Math.min(minX, barbBX);                                       160

      minY = Math.min(srcY, dstY);
      minY = Math.min(minY, barbAY);
      minY = Math.min(minY, barbBY);

      maxX = Math.max(srcX, dstX);
      maxX = Math.max(maxX, barbAX);
      maxX = Math.max(maxX, barbBX);

      maxY = Math.max(srcY, dstY);                                         170
      maxY = Math.max(maxY, barbAY);
      maxY = Math.max(maxY, barbBY);

      setBounds(new Rectangle(minX, minY, maxX-minX, maxY-minY));
    }

    public void dispose() {
      this.source = null;
      this.destination = null;
      super.dispose();                                                     180
    }
}

/**
 * $Log: ConnectorSprite.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

Destination.java

```
/*
```

```
 * Destination Interface.
 * $Id: Destination.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.gui;

/**
 * Provides a marker interface for objects wanting to play the role
 * of a destination in a connection operation.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *                                                                             20
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: Destination.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public interface Destination
{
}

/**
 * $Log: Destination.java,v $                                                  30
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

LabelSprite.java

```
/*
 * Label Sprite Class.
 * $Id: LabelSprite.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.gui;
import java.awt.*;

/**
 * Takes the form of a free-floating piece of text, possibly with
 * an embossed button effect.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.                   20
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: LabelSprite.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */
```

```
public class LabelSprite extends Sprite {

  /**
   * Records the inset, in pixels, from each of the four borders.
   */
  protected Insets insets;

  /**
   * The default inset size, in pixels, for a label sprite.
   */
  public static final int MARGIN = 5;

  /**
   * Records the amount, in pixels, by which the label must be
   * raised in order to see the whole thing.  Depends on the current
   * canvas font.
   */
  protected int descent;

  /**
   * True if the label is to be drawn as simply a floating piece of text.
   * False if the label is to be drawn as a three dimensional button
   * with text written onto it.  The button will get depressed when
   * selected and feel a lot better when deselected.
   */
  protected boolean noEmboss;

  /**
   * A public no-argument constructor.
   */
  public LabelSprite() {
    super();
    this.insets = new Insets(MARGIN, MARGIN, MARGIN, MARGIN);
    this.setLabel(this.getClass().getName());
  }

  public void setLabel(String label) {
    super.setLabel(label);
    int w=0, h=0;
    if (label != null && this.canvas != null) {
      FontMetrics fm = this.canvas.getFontMetrics(this.canvas.getFont());
      h = fm.getHeight();
      w = fm.stringWidth(this.label);
      this.descent = fm.getDescent();
    }
    Rectangle old = this.getBounds();
    int x=0, y=0;
    if (old != null) {
      x = old.x;
      y = old.y;
    }
    this.setBounds(new Rectangle(x, y,
      w+this.insets.left+this.insets.right,
      h+this.insets.top+this.insets.bottom));
  }

  public void setComponent(Component canvas) {
    super.setComponent(canvas);
    this.setLabel(this.getLabel());
  }

  public void paint(Graphics g) {
    if (this.bounds == null) return;
    if (!this.noEmboss) {
      g.setColor(Color.lightGray);
      g.fill3DRect(bounds.x, bounds.y,
                   bounds.width, bounds.height, !selected);
    }
```

```
    g.setColor(Color.blue);
    g.drawString(this.label, this.bounds.x+this.insets.left,
                 bounds.y+bounds.height-
                 this.insets.bottom-this.descent);
  }
}
                                                                    100
/**
 * $Log: LabelSprite.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

Source.java

```
/*
 * Source Interface.
 * $Id: Source.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                     10
 */

package robotworld.gui;

/**
 * Provides a marker interface for objects wanting to play the role
 * of a source in a connection operation.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *                                                                    20
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: Source.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public interface Source
{
}

/**
 * $Log: Source.java,v $                                              30
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

Sprite.java

```
/*
 * Sprite Abstract Class.
 * $Id: Sprite.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                                10
 */

package robotworld.gui;
import robotworld.ide.*;
import java.awt.*;
import java.util.*;
import java.io.*;

/**
 * Sprites are graphical objects which can contain other sprites,                20
 * manage their appearance, and interact with screen space.
 * For better or worse, these three functionalities---model, view, and
 * controller---are all rolled into one class.  Here's a quick breakdown
 * of the methods of this class by functionality:<br>
 * <ul>
 * <li>Model: sprites can contain other sprites
 *     <ul>
 *     <li><code>getParent, setParent</code>
 *     <li><code>addChild, removeChild, getChildAt, getNumChildren</code>
 *     <li><code>dispose</code>                                                  30
 *     </ul>
 * <li>View: sprites can manage their apperance
 *     <ul>
 *     <li><code>getLabel, setLabel</code>
 *     <li><code>getComponent, setComponent</code>
 *     <li><code>getSelected, setSelected</code>
 *     <li><code>paint, paintChildren</code>
 *     </ul>
 * <li>Controller: sprites can interact with screen space
 *     <ul>                                                                      40
 *     <li><code>getLocation, setLocation</code>
 *     <li><code>getBounds, setBounds</code>
 *     <li><code>getDepth, setDepth</code>
 *     <li><code>getRelativeOffset, setRelativeOffset</code>
 *     <li><code>getLinkSprite, setLinkSprite</code>
 *     <li><code>find, findChildAt</code>
 *     </ul>
 * </ul>
 * The class also has a no-argument constructor
 * and a <code>toString</code> method.                                           50
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: Sprite.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public abstract class Sprite implements Serializable {

  /**                                                                             60
   * When true, this flag indicates the sprite has been selected.
   */
  protected boolean selected;

  /**
   * The children of this sprite.
   */
  protected Vector children = new Vector();
```

```
/**
 * The rectangular bounds of this sprite.
 */
protected Rectangle bounds;

/**
 * The string label of this sprite.
 */
protected String label;

/**
 * Determines which sprites are "closest" the viewer both in terms
 * of mouse targeting behavior (closest wins) and paint sort order.
 * The default value is 0.  This is just a reference point.
 * Sprites may have positive or negative depth.  Integer.MAX_VALUE
 * is considered most distant, while Integer.MIN_VALUE is considered
 * closest to the viewer.
 */
protected int depth = 0;

/**
 * The parent of this sprite.  Serializing this field does not pose
 * pose a threat, since the serialization algorithm is careful to
 * output each object only once---it cannot enter an infinite loop.
 * Java in a Nutshell, 2nd ed., Ch. 9, p172.
 */
protected Sprite parent;

/**
 * The component which is rendering this sprite.  There is no need
 * to serialize it, so this field is declared transient.
 */
protected transient Component canvas;

/**
 * Linked sprites are useful for data structures and relative offsets.
 */
protected Sprite linkSprite;

/**
 * The offset to maintain relative to the link sprite.
 */
protected Point relativeOffset;

/**
 * A public no-arg constructor is essential.
 */
public Sprite() {
}

/**
 * Returns a string representation of this sprite.
 * This includes the class, the label, and number of children.
 */
public String toString() {
  String str=this.getClass().getName() + " (" +
          this.getLabel() + ") :\n   Children:  " +
          this.getNumChildren() + "\n";
  return str;
}

/*================================*
 * Methods to manage sprite tree *
 *================================*/

/**
 * Sets the parent of this sprite.
```

```
     *
     * @param parent the parent of this sprite.
     */
    public void setParent(Sprite parent) {
        this.parent = parent;
        if (this.parent != null) {
            parent.addChild(this);
        }
    }

    /**
     * Gets the parent of this sprite.
     *
     * @return the parent of this sprite.
     */
    public Sprite getParent() {
        return this.parent;
    }

    /**
     * Adds the given child.
     *
     * @param child the sprite to add
     */
    public void addChild(Sprite child) {
        this.children.addElement(child);
    }

    /**
     * Gets the child with the specified index.
     *
     * @param index the index of the child
     * @return the child sprite with that index
     */
    public Sprite getChildAt(int index) {
        return (Sprite)this.children.elementAt(index);
    }

    /**
     * Removes the given child from this sprite.
     *
     * @param child the child sprite to be removed.
     */
    public void removeChild(Sprite child) {
        this.children.removeElement(child);
    }

    /**
     * Gets this sprite's number of children
     *
     * @return this sprite's number of children
     */
    public int getNumChildren() {
        return this.children.size();
    }

    /**
     * Disposes of any resources before this sprite is garbage collected.
     * This includes removing references to itself from the parent sprite,
     * link sprite, and any other sources of garbage retention.
     */
    public void dispose() {
        if (this.parent != null) {
            this.parent.removeChild(this);
        }
    }
```

```
/*==================================*
 * Methods to manage screen space   *
 *==================================*/

/**
 * Gets the center point of this sprite.                                    210
 *
 * @param location the point which is the center of this sprite.
 */
public void setLocation(Point location) {
  this.setLocation(location.x, location.y);
}

/**
 * Sets the center coordinates of this sprite.
 *                                                                          220
 * @param xLocation the x location of the center
 * @param yLocation the y location of the center
 */
public void setLocation(int xLocation, int yLocation) {
  this.bounds.x = xLocation - this.bounds.width/2;
  this.bounds.y = yLocation - this.bounds.height/2;
  if (this.relativeOffset != null) {
    this.bounds.x += this.relativeOffset.x;
    this.bounds.y += this.relativeOffset.y;
  }                                                                         230
}

/**
 * Gets the center point of this sprite.
 *
 * @return the point which is the center of this sprite.
 */
public Point getLocation() {
  int x = this.bounds.x + this.bounds.width/2;
  int y = this.bounds.y + this.bounds.height/2;                             240
  if (this.relativeOffset != null) {
    x -= this.relativeOffset.x;
    y -= this.relativeOffset.y;
  }
  return new Point(x, y);
}

/**
 * Sets the offset to maintain relative to  the link sprite.
 *                                                                          250
 * @param offset the point offset to maintain relative to the link sprite.
 */
public void setRelativeOffset(Point offset) {
  this.relativeOffset = offset;
}

/**
 * Gets the relative offset from the link sprite.
 *
 * @return the point offset from the link sprite.                           260
 */
public Point getRelativeOffset() {
  return this.relativeOffset;
}

/**
 * Sets the bounds of this sprite.
 *
 * @param bounds the rectangular bounds of this sprite.
 */                                                                         270
public void setBounds(Rectangle bounds) {
  this.bounds = bounds;
```

```
}

/**
 * Gets the bounds of this sprite.
 *
 * @return the rectangular bounds of this sprite.
 */
public Rectangle getBounds() {
  return this.bounds;
}


/**
 * Sets the sprite be to linked to this one.
 *
 * @param link the sprite to be linked to this one.
 */
public void setLinkSprite(Sprite link) {
  this.linkSprite = link;
}

/**
 * Gets the sprite linked to this one.
 *
 * @return the sprite linked to this one.
 */
public Sprite getLinkSprite() {
  return this.linkSprite;
}

/**
 * Finds out what sprite contains the given point.  Usually
 * this sprite, or null if the point is outside the bounds
 * of this sprite.  Other sprites may also be returned,
 * for example, a linked sprite.  Candidates are allowed to
 * modulate their picking behavior based on the state of
 * the canvas indicated by the <code>specialMode</code> flag.
 *
 * @param p the point of interest.
 * @param specialMode a flag indicating the state of the canvas
 * @return the sprite containing the given point, if any
 */
public Sprite find(Point p, boolean specialMode) {
  Rectangle r = this.getBounds();
  if (r != null && r.contains(p)) {
    return this;
  } else {
    return null;
  }
}

/**
 * Finds the closest child sprite containing the given point.
 * Among candidates, picking preference is given to sprites
 * "closest" to the viewer (determined by calling <code>getDepth</code>).
 * Candidates are allowed to modulate their picking behavior based on
 * the state of the canvas indicated by the <code>specialMode</code> flag.
 *
 * @param p the point of interest.
 * @param specialMode a flag indicating the state of the canvas
 * @return the closest child sprite containing the given point
 */
public Sprite findChildAt(Point p, boolean specialMode) {
  int minDepth = Integer.MAX_VALUE;
  Sprite result = null;

  // for all children
  for (int i=0; i<this.getNumChildren(); i++) {
```

```
      Sprite child = this.getChildAt(i);
      int curDepth = child.getDepth();

         // if a child doesn't want to be picked, it returns null
      Sprite curSprite = child.find(p, specialMode);

      // closest child that wants to be picked, wins
      if (curSprite != null &&  curDepth < minDepth) {
         minDepth = curDepth;
         result = curSprite;                                                    350
      }
    }
    return result;
}

/**
 * Sets the depth of the sprite.
 *
 * @param depth the integer depth of the sprite.
 */                                                                             360
public void setDepth(int depth) {
    this.depth = depth;
}

/**
 * Gets the depth of the sprite.
 *
 * @return the integer depth of the sprite.
 */
public int getDepth() {                                                         370
    return this.depth;
}

/*================================*
 * Methods to manage appearance  *
 *================================*/

/**
 * Sets the string label of this sprite.
 *                                                                              380
 * @param label the string label of this sprite
 */
public void setLabel(String label) {
    this.label = label;
}

/**
 * Gets the string label of this sprite.
 *
 * @return the string label of this sprite                                     390
 */
public String getLabel() {
    return this.label;
}

/**
 * Sets the component which is rendering this sprite.
 *
 * @param canvas the component which is rendering this sprite.
 */                                                                             400
public void setComponent(Component canvas) {
    this.canvas = canvas;
}

/**
 * Gets the component which is rendering this sprite.
 *
 * @return the component which is rendering this sprite.
```

```
   */
  public Component getComponent() {
    return this.canvas;
  }

  /**
   * Sets the selected status of this sprite.
   *
   * @param selected true to select this sprite, false to deselect
   */
  public void setSelected(boolean selected) {
    this.selected = selected;
  }

  /**
   * Gets the selected status of this sprite.
   *
   * @return true if this sprite is selected, false otherwise.
   */
  public boolean getSelected() {
    return this.selected;
  }

  /**
   * Paints this sprite's children, as opposed to itself,
   * onto the given graphics object.
   * This involves iterating over every child, and giving it a chance
   * to <code>paint</code> itself onto the graphics object.
   *
   * @param g the graphics object to be painted.
   */
  public void paintChildren(Graphics g) {

    // if depth ever becomes an important factor in display--
    // as opposed just arbitrating the result of a mouse click--
    // then the children should be sorted by depth, and painted
    // from most distant to least distant.
    for (int i=0; i<this.getNumChildren(); i++) {
      this.getChildAt(i).paint(g);
    }
  }

  /**
   * Paints this sprite, but not its children, onto the given
   * graphics object.
   *
   * @param g the graphics object to be painted.
   */
  public abstract void paint(Graphics g);
}

/**
 * $Log: Sprite.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

SpriteCanvas.java

```
/*
```

```
 * Sprite Canvas Class.
 * $Id: SpriteCanvas.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.gui;
import robotworld.ide.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.JComponent;

/**                                                                            20
 * Provides animation and interactivity support for sprites.
 * Implements the following user interface:
 * <ul>
 * <li>Clicking once on a sprite selects (or deselects) it.
 * <li>Clicking twice on a sprite opens it.
 * <li>Dragging on a sprite moves it about.
 * <li>Clicking on open space creates a sprite there.
 * </ul>
 * The canvas is continually painted in an animation loop.
 * The apparent frame rate and hence the smoothness of motion are              30
 * determined by two things:
 * <ul>
 * <li>The canvas refresh rate <em>R</em>, approximately (1000L / sleepPeriod)
 *     frames per second.
 * <li>The rate of externally driven change to sprite state <em>D</em>
 * </ul>
 * The apparent frame rate <em>F</em> is then <em>min(R,D)</em>.
 * If <em>D</em> is much greater than <em>R</em>, then the sprite
 * animation will be quite jerky.  It also depends on the nature and magnitude
 * of the changes to sprite state.                                             40
 * <p>
 * For reference, the frame rate at which the human visual system
 * maintains the illusion of smoothness is about 18 frames per second.
 * (This is known as the critical flicker frequency).
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: SpriteCanvas.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */                                                                            50

public class SpriteCanvas extends JComponent implements Runnable {

  /**
   * This canvas animates the root sprite's children.
   */
  protected Sprite rootSprite;

  /**
   * The last selected sprite.                                                 60
   */
  protected Sprite lastSprite;

  /**
   * A means of ensuring unique sprite labels---within the
   * Java Virtual Machine, at least.
   */
  protected static int numSprites;
```

```
/**
 * The default canvas width.
 */
protected int canvWidth = 200;

/**
 * The default canvas height.
 */
protected int canvHeight = 150;

/**
 * The default sleep period, in milliseconds.  This is related to
 * the canvas frame rate, in frames per second (fps):<br><pre>
 *    R = 1000L / sleepPeriod</pre>
 */
protected long sleepPeriod = 50L;

/**
 * The class last selected by the user.
 */
protected Class currentBean;

/**
 * The workspace this canvas is yoked to.
 */
protected Workspace workspace;

/**
 * Subclasses can set this flag to put the canvas into a Special Mode.
 * The only effect on this class is to change the background color
 * to <code>specialModeBackground</code> rather than
 * <code>defaultBackground</code>.  However, the flag may have special
 * meaning to SpriteCanvas subclasses, which are free to offer an alternative
 * user interface to the one described above when the flag is set.
 * The flag also has special meaning for sprites, who are free to alter
 * their ability to be targeted when the flag is set.
 */
protected boolean specialMode;

/**
 * The background color when this canvas is in its special mode.
 */
protected Color specialModeBackground = Color.darkGray;

/**
 * The default background color when this canvas is not in its special mode.
 */
protected Color defaultBackground = Color.lightGray;

/**
 * A public no-arg constructor is key.  Kicks off an animation thread
 * and adds the appropriate mouse listeners.
 */
public SpriteCanvas() {
  this.setBackground(this.defaultBackground);

  // BUG #4177815: "Canvas does not pass on KeyEvents"
  // For now, the work will be shuffled off to the SpriteWindow.

  // remove all selected sprites on DEL key
  /*this.addKeyListener(new KeyAdapter() {
    public void keyPressed(KeyEvent e) {
      if (e.getKeyCode() == KeyEvent.VK_DELETE) {
        SpriteCanvas.this.removeLastSprite();
        SpriteCanvas.this.repaint();
      } else if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
        SpriteCanvas.this.editLastSpriteProperties();
      }
```

```
    }
  });*/

  // select or create sprites on mouse press events
  this.addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
      Sprite spr = SpriteCanvas.this.findSpriteAt(e.getPoint());

      if (spr != null && (e.getClickCount() > 1)) {
        SpriteCanvas.this.openSprite(spr);
      } else if (spr != null) {
        SpriteCanvas.this.selectSprite(spr);
      } else {
        SpriteCanvas.this.createSprite(e.getPoint());
      }

      SpriteCanvas.this.repaint();
    }
  });

  // move sprites on mouse drag events
  this.addMouseMotionListener(new MouseMotionAdapter() {
    public void mouseDragged(MouseEvent e) {
      if (SpriteCanvas.this.lastSprite != null) {
        SpriteCanvas.this.lastSprite.setLocation(e.getPoint());
        SpriteCanvas.this.repaint();
      }
    }
    public void mouseMoved(MouseEvent e) {
    }
  });

  // start the animation thread
  new Thread(this).start();
}

/**
 * Sets the root sprite for this canvas.  The root sprite's
 * children will be painted onto the canvas.
 */
public void setRootSprite(Sprite spr) {
  this.rootSprite = spr;
  if (spr != null) spr.setComponent(this);
}

/**
 * Determine this canvas's root sprite, if any.
 *
 * @return the root sprite of this canvas.
 */
public Sprite getRootSprite() {
  return this.rootSprite;
}

/**
 * Opens a new workspace with the given sprite as the root sprite.
 * Uses this workspace's <code>getLayerFor</code> utility to
 * instantiate an appropriate workspace for the given sprite.
 * Then it calls the new workspace's <code>configureToEdit</code>
 * method, passing it the sprite to be edited.  Launches an
 * exception dialog if anything goes wrong.
 *
 * @param spr the sprite to open.
 */
protected void openSprite(Sprite spr) {
  this.selectSprite(spr);
  Workspace ws = null;
  try {
```

```
      ws = this.workspace.getLayerFor(spr.getClass());
      ws.configureToEdit(spr);
    } catch (Exception x) {
      this.workspace.launchExceptionDialog(x);
    }
  }

  /**
   * Opens a property editor for the last selected sprite.
   */
  protected void editLastSpriteProperties() {
    if (this.lastSprite != null) {
      SpritePropertyEditor spe = new SpritePropertyEditor();
      spe.setSprite(this.lastSprite);
      spe.pack();
      spe.show();
    }
  }

  /**
   * Selects the given sprite.
   *
   * @param spr the sprite to select.
   */
  protected void selectSprite(Sprite spr) {
    if (this.lastSprite != null) {
      this.lastSprite.setSelected(false);
    }
    spr.setSelected(!spr.getSelected());
    this.lastSprite = spr;
  }

  /**
   * Sets the class which will be used to instantiate objects on
   * a click to create.
   *
   * @param c the class to remember.
   */
  public void setCurrentBean(Class c) {
    this.currentBean = c;
  }

  /**
   * Remembers the workspace which this canvas is yoked to.
   *
   * @param w the workspace this canvas is yoked to.
   */
  public void setWorkspace(Workspace w) {
    this.workspace = w;
  }

  /**
   * Creates a sprite on the canvas using the current bean class.
   * Launches an exception dialog if anything goes wrong.
   *
   * @return the sprite just created.
   */
  protected Sprite createSprite() {
    Sprite spr = null;
    try {
      if (this.currentBean != null) {
        spr = (Sprite)this.currentBean.newInstance();
        this.addSprite(spr);
      }
    } catch (Exception e) {
      this.workspace.launchExceptionDialog(e);
    }
    return spr;
```

```
}

/**
 * Creates a sprite on the canvas at the given point.
 * Calls <code>createSprite()</code> and does some other housekeeping.
 *
 * @param p the point to create the sprite at.                          280
 */
protected void createSprite(Point p) {
  Sprite spr = this.createSprite();
  if (spr != null) {
    spr.setLabel(spr.getLabel() + (++this.numSprites));
    spr.setLocation(p);
    this.lastSprite = spr;
  }
}
                                                                        290
/**
 * Adds the given sprite to the canvas.  Selects the sprite,
 * sets its component to be this very canvas, and sets its parent
 * to be the root sprite. This has the reciprocal effect of telling
 * the root sprite to add the sprite as a new child, so that
 * all further paint and find operations on the root sprite will
 * implicitly affect the added sprite.
 *
 * @param spr the sprite to add.
 */                                                                     300
public void addSprite(Sprite spr) {
  if (this.lastSprite != null) {
    this.lastSprite.setSelected(false);
  }
  spr.setParent(this.rootSprite);
  spr.setSelected(true);
  spr.setComponent(this);
}

                                                                        310
/**
 * Removes the last selected sprite, if any.
 * Has the side effect of calling the sprite's <code>dispose</code>
 * method.
 */
public void removeLastSprite() {
  if (this.lastSprite != null) {
    this.lastSprite.dispose();
    this.lastSprite = null;
  }                                                                     320
}

 /**
  * Forever calls the act method in a while-true loop.
  * Sleeps for <code>sleepPeriod</code> between each call.
  */
public void run() {
  while (true) {
    act();
    try {                                                               330
      Thread.sleep(this.sleepPeriod);
    } catch (InterruptedException e) {
    }
  }
}

/**
 * The unit of action.  Causes a repaint.
 */
public void act() {                                                     340
  this.repaint();
```

```
  }

  /**
   * Returns the sprite that contains the given point,
   * or null if there is no such sprite.
   * <p>
   * Here are the chain of events.  On every mouse press in the canvas,
   * the <code>findSpriteAt(Point p)</code> method is called to determine
   * the sprite that was clicked on.  This in turn calls the root sprite's
   * <code>findChildAt(Point p, boolean specialMode)</code> method,
   * passing it the current value of the <code>specialMode</code> flag.
   * Most likely, the root sprite will call each child sprite's
   * <code>find(Point p, boolean specialMode)</code> method, arbitrating
   * if more than one child decide they've been targeted.
   *
   * @param p the point of interest
   * @return the sprite that contains the point, if any.
   */
  public Sprite findSpriteAt(Point p) {
    if (this.rootSprite==null) {
      return null;

    } else {
      return this.rootSprite.findChildAt(p, this.specialMode);
    }
  }

   /**
    * Paints the root sprite's children.
    * If the canvas is in its <code>specialMode</code>, changes
    * the background color to <code>specialModeBackground</code>.
    * Otherwise sets the background color to <code>defaultBackground</code>.
    * Does nothing if <code>rootSprite</code> is null.
    *
    * @param g the graphics object to paint on
    */
  public void paint(Graphics g) {
    if (this.rootSprite == null) return;
    if (this.specialMode) {
      this.setBackground(this.specialModeBackground);
    } else {
      this.setBackground(this.defaultBackground);
    }
    this.rootSprite.paintChildren(g);
  }

  public Dimension getPreferredSize() {
    return new Dimension(this.canvWidth, this.canvHeight);
  }
}

/**
 * $Log: SpriteCanvas.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

SpritePropertyEditor.java

```
/*
```

```
 * Sprite Property Editor Class.
 * $Id: SpritePropertyEditor.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.gui;
import java.awt.*;
import java.awt.event.*;

/**
 * Provides a means of tinkering with sprite properties.
 * Currently only lets you alter the sprite's label.
 * <p>                                                                         20
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: SpritePropertyEditor.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class SpritePropertyEditor extends Frame
{
  /**
   * A field to contain sprite data.                                           30
   */
  protected TextField labelField, classField;

  /**
   * A human-readable label to accompany the text field.
   */
  protected Label labelLabel, classLabel;

  /**
   * The sprite being tinkered with.                                           40
   */
  protected Sprite sprite;

  /**
   * A no-argument constructor is always nice.
   * Creates a field for the sprite's class, and labels it.
   * Creates a field for the sprite's label, and labels it.
   */
  public SpritePropertyEditor() {
    // submit to the absolute ontology of the AWT                             50
    this.setTitle("Sprite Property Editor");
    Panel pLeft = new Panel();
    LayoutManager mLeft = new GridLayout(2,1);
    pLeft.setLayout(mLeft);
    this.add("West",pLeft);
    this.classLabel = new Label("Class:  ");
    this.labelLabel = new Label("Label:  ");
    pLeft.add(this.classLabel);
    pLeft.add(this.labelLabel);
                                                                              60
    Panel pRight = new Panel();
    LayoutManager mRight = new GridLayout(2,1);
    pRight.setLayout(mRight);
    this.add("East",pRight);
    this.classField = new TextField();
    this.classField.setEditable(false);
    this.labelField = new TextField();
    this.labelField.setEditable(true);
    pRight.add(this.classField);
```

```
    pRight.add(this.labelField);
  }

  /**
   * Sets the sprite's label to the contents of the label field
   * and disposes of this window.
   */
  public void updateLabel() {
    this.sprite.setLabel(this.labelField.getText());
    this.dispose();
  }

  /**
   * Sets the sprite to be tinkered with.
   * Fills in the class and label fields, and adds an action listener
   * that causes <code>updateLabel</code> to be called when the user
   * hits enter.
   *
   * @param spr the sprite to be tinkered with.
   */
  public void setSprite(Sprite spr) {
    this.sprite = spr;
    this.classField.setText(spr.getClass().getName());
    this.labelField.setText(spr.getLabel());
    this.labelField.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        SpritePropertyEditor.this.updateLabel();
      }
    });
  }
}

/**
 * $Log: SpritePropertyEditor.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

SpriteWindow.java

```
/*
 * Sprite Window Class.
 * $Id: SpriteWindow.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.gui;
import robotworld.ide.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.JList;
import javax.swing.JButton;
import javax.swing.Box;
import javax.swing.JScrollPane;
```

```
import javax.swing.JOptionPane;
import javax.swing.DefaultListModel;
import javax.swing.event.ListSelectionListener;
import javax.swing.event.ListSelectionEvent;

/**
 * Provides a window for a sprite canvas to go into.
 * Not very bright, but it does manage the "beanbag" which
 * is the list of strings to the left, which controls what can be
 * instantiated into the sprite canvas.  The strings must be
 * fully qualified package names of classes accessible from the codebase.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: SpriteWindow.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class SpriteWindow extends UniformWorkspace
  implements ListSelectionListener {

   /**
    * Sprite canvas goes on the right.
    */
  public SpriteCanvas spriteCanvas;

   /**
    * Data list (inside a scroll pane) goes on the left.
    */
  public JList dataList;

  /**
   * This button allows the currently selected bean to be removed from the
   * bean bag.
   */
  public JButton removeBeanButton;

  /**
   * This button allows a bean to be added to the bean bag.
   */
  public JButton addBeanButton;

  /**
   * This action command indicates bean addition.
   */
  public String ADD_BEAN = "Add Bean";

  /**
   * This action command indicates bean removal.
   */
  public String REMOVE_BEAN = "Remove Bean";

  /**
   * A public no-argument constructor is essential.
   */
  public SpriteWindow() {
  }

  /**
   * Initializes the window and makes it appear.
   * Sets up a scrolling list on the left, using the items from
   * <code>dataList</code>, and identifies this window as a
   * list selection listener.  Sets up a sprite canvas on the right.
   * Finally, its packs and shows the window.
   */
  protected void init() {
    // set up a scrolling list on the left
    Box leftHalf = Box.createVerticalBox();
```

```
    this.dataList = new JList(Workspace.getBeanbagForLayer(this.getClass()));
    JScrollPane scrollPane = new JScrollPane(this.dataList);
    this.dataList.addListSelectionListener(this);
    leftHalf.add(scrollPane);

    Box addAndRemove = Box.createHorizontalBox();
    this.addBeanButton = new JButton(ADD_BEAN);
    this.addBeanButton.addActionListener(this);
    addAndRemove.add(this.addBeanButton);
    this.removeBeanButton = new JButton(REMOVE_BEAN);
    this.removeBeanButton.addActionListener(this);
    addAndRemove.add(Box.createHorizontalGlue());
    addAndRemove.add(this.removeBeanButton);

    leftHalf.add(addAndRemove);
    this.getContentPane().add("West", leftHalf);

    // and the sprite canvas on the right
    this.spriteCanvas.setWorkspace(this);
    this.getContentPane().add("Center", this.spriteCanvas);

    // make the window appear
    this.pack();
    this.show();
  }

  public void actionPerformed(ActionEvent e) {
    // get the beanbag for this layer.
    DefaultListModel model = Workspace.getBeanbagForLayer(this.getClass());
    String command = e.getActionCommand();

    // if they clicked the ADD_BEAN button, ask which one
    // then add their answer to the beanbag
    if (command.equals(ADD_BEAN)) {
      String answer = JOptionPane.showInputDialog(
        this, "Add which bean to the bean bag?\n"+
        "Example:  robotworld.gui.Sprite", "Add Bean",
        JOptionPane.QUESTION_MESSAGE);
      model.addElement(answer);

    // if they clicked on the REMOVE_BEAN button, remove
    // the currently selected bean from the beanbag.
    } else if (command.equals(REMOVE_BEAN)) {
      model.removeElement(this.dataList.getSelectedValue());

    // guess this was meant for the superclass instead.
    } else {
      super.actionPerformed(e);
    }
  }

  public void configureToEdit(Object o) {
    if (o instanceof Sprite) {
      Sprite spr = (Sprite)o;
      this.spriteCanvas.setRootSprite(spr);
    }
  }

  /**
   * Handles value changes on a list selection event.
   * Turns selections into beans using <code>Class.forName</code>,
   * and clears beans on a deselect.  Assumes selections are strings
   * which are fully qualified package names of classes accessible
   * from the codebase.  Launches an exception dialog if there are
   * any problems.
   *
   * @param e the list selection event.
   */
```

```
  public void valueChanged(ListSelectionEvent e) {
    try {

      // turn a selection into a bean                                    160
      if (!this.dataList.isSelectionEmpty()) {
        String str = (String)this.dataList.getSelectedValue();
        Class bean = Class.forName(str);
        this.spriteCanvas.setCurrentBean(bean);

      // clear beans on a deselect
      } else {
        this.spriteCanvas.setCurrentBean(null);
      }
                                                                        170
    // and yell if there are problems
    } catch (Exception x) {
      this.launchExceptionDialog(x);
    }
  }

  public void clear() {}
  public void cut() {}
  public void copy() {}
  public void delete() {                                                180
    this.spriteCanvas.removeLastSprite();
    this.spriteCanvas.repaint();
  }
  public void paste() {}
  public void load() {}
  public void save() {}
  public void saveAs() {}
  public void rename() {
    this.spriteCanvas.editLastSpriteProperties();
  }                                                                     190
  public void print() {}

}

/**
 * $Log: SpriteWindow.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */                                                                     200
```

C.6 The robotworld.ctl package

ActuatorPort.java

```
/*
 * Actuator Port Class.
 * $Id: ActuatorPort.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
```

```
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.ctl;
import robotworld.gui.*;
import robotworld.ide.*;
import robotworld.net.*;
import java.io.*;

/**
 * Provides a means of polling a local connection, and broadcasting the
 * command to an unknown number of remote actuators.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: ActuatorPort.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class ActuatorPort extends SignalDestination
implements RemoteSource, Runnable {

  /**
   * The period, in milliseconds, between calls to <code>act</code>.
   */
  protected long period = 200;

  /**
   * The sensor port's unique identifier.
   */
  protected Object uid;

  /**
   * Public no-argument constructor.  Kicks off an animacy.
   */
  public ActuatorPort() {
    new Thread(this).start();
  }

  /**
   * Continuously calls act every <code>period</code> milliseconds.
   */
  public void run() {
    while (true) {
      act();
      try {
        Thread.sleep(this.period);
      } catch (InterruptedException e) {
      }
    }
  }

  /**
   * Gets the unique identifier for this actuator port.
   * If <code>uid</code> is non-null, simply returns it.
   * Otherwise, it starts with this sprite and climbs up
   * the sprite tree, concatenating sprite labels seperated
   * by a colon, until it reaches root.  The result is
   * a UID string of the form:<br><pre>
   *    my name : parent's name : ... : root sprite's name</pre>
   *
   * @return the unique identifier object for this sprite.
   */
  public Object getUID() {
    if (this.uid == null) {
      String tempUID = this.getLabel();
```

```
        Sprite tempSprite = null;
        for (tempSprite = this.getParent();
             tempSprite != null;
             tempSprite = tempSprite.getParent()) {
          tempUID += ":"+tempSprite.getLabel();
        }
        this.uid = tempUID;
      }
      return this.uid;
    }

    /**
     * Polls for the latest command and writes it to the blackboard
     * using this sprite's UID as a subject, making sure to remove
     * the previous entry first.
     */
    public void act() {
      try {
        Blackboard bb = Workspace.getBlackboard();
        SignalEntry update = (SignalEntry)Workspace.produce(SignalEntry.class);
        update.setSubject(this.getUID());
        update.setClient(null);
        bb.take(update);
        update.setSample(this.getSample());
        bb.write(update);
      } catch (Exception x) {
      }
    }
}

/**
 * $Log: ActuatorPort.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

CompositeController.java

```
/*
 * Composite Controller Class.
 * $Id: CompositeController.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.ctl;
import robotworld.gui.*;
import java.awt.*;
import java.util.*;

/**
 * Allows controllers to be recursively nested.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
```

```
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: CompositeController.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class CompositeController extends Controller {

  /**
   * A bunch of nested controllers.                                          30
   */
  protected Vector controllers = new Vector();

  /**
   * A public no-argument constructor is essential.
   */
  public CompositeController() {
  }

  public String toString() {                                               40
    String str = super.toString();
    str += "    Controllers:  "+this.getNumControllers()+"\n\n";
    for (int i=0; i<this.getNumControllers(); i++) {
      str += this.getControllerAt(i).toString() + "\n";
    }
    return str;
  }

  public void addChild(Sprite child) {
    super.addChild(child);                                                 50
    if (child instanceof Controller) {
      this.addController((Controller)child);
    }
  }

  public void removeChild(Sprite child) {
    super.removeChild(child);
    if (child instanceof Controller) {
      this.removeController((Controller)child);
    }                                                                      60
  }

  public void addController(Controller ctlr) {
    this.controllers.addElement(ctlr);
  }

  public int getNumControllers() {
    return this.controllers.size();
  }
                                                                           70
  public Controller getControllerAt(int index) {
          return (Controller)this.controllers.elementAt(index);
  }

  public void removeController(Controller ctlr) {
    this.controllers.removeElement(ctlr);
  }

  public void dispose() {
    int count=this.getNumControllers();                                    80
    for (int i=count-1; i>=0; i--) {
      this.getControllerAt(i).dispose();
    }
    super.dispose();
  }
}

/**
 * $Log: CompositeController.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh                          90
```

```
 * Original Version.
 *
 */
```

Connection.java

```
/*
 * Connection Class.
 * $Id: Connection.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.ctl;
import robotworld.gui.*;
import robotworld.net.*;
import java.awt.*;

/**
 * Provides a thread-safe local communication mechanism between two sprites,
 * with visual connectivity in the canvas as well. It resamples or drops       20
 * incoming values, depending on the speed at which values are generated
 * (by a <code>SignalSource</code>) versus the speed which they are sampled
 * (by a <code>SignalDestination</code>). If generation is faster, some
 * values will be dropped. If sampling is faster, some values will appear
 * multiple times.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: Connection.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $    30
 */

public class Connection extends ConnectorSprite {

  /**
   * This field contains the current sample.
   */
  protected Object sample;

  /**                                                                          40
   * A parameterless constructor is essential.
   */
  public Connection() {
  }

  public void addChild(Sprite child) {
    if (child instanceof SignalSource) {
      this.setSource(child);
    } else if (child instanceof SignalDestination) {
      this.setDestination(child);                                             50
    }
  }

  public void setSource(Sprite src) {
    if (src instanceof SignalSource) {
```

```
      SignalSource sigSrc = (SignalSource)src;
      sigSrc.addConnection(this);

      // check whether pins are more appropriate
      if (this.getParent() != src.getParent()) {
        super.setSource(src.getLinkSprite());
       } else {
        super.setSource(src);
      }
    }
}

public void setDestination(Sprite dst) {
    if (dst instanceof SignalDestination) {
      SignalDestination sigDst = (SignalDestination)dst;
      sigDst.setConnection(this);

      // check whether pins are more appropriate
      if (this.getParent() != dst.getParent()) {
        super.setDestination(dst.getLinkSprite());
      } else {
        super.setDestination(dst);
      }
    }
}

/**
 * Sets the current value of the sample.
 * It is synchronized in order to avoid internal read/write
 * conflicts with <code>getSample</code>, which is public.
 *
 * @param sample the sample object.
 */
protected synchronized void setSample(Object sample) {
    this.sample = sample;
}

// Overrides the ability to set location.
public void setLocation(Point p) {
}

/**
 * Gets the current value of the sample.
 *
 * @return the sample object.
 */
public synchronized Object getSample() {
    return this.sample;
}

// can you say "convoluted"?
public void dispose() {

    // the arrow might have begun at the real signal source,
    SignalSource sigSrc = null;
    Pin pin = null;
    if (this.source instanceof SignalSource) {
      sigSrc = (SignalSource)this.source;

    // or it might have begun at a pin,
    } else if (this.source instanceof Pin) {
      pin = (Pin)this.source;

        // which is merely a proxy to the real signal source.
      sigSrc = (SignalSource)pin.getLinkSprite();
    }

    // okay, now unplug me from the signal source.
```

```
    if (sigSrc != null) {
      sigSrc.removeConnection(this);
    }

    // the arrow might have ended at the real signal destination,
    SignalDestination sigDst = null;
    if (this.destination instanceof SignalDestination) {
      sigDst = (SignalDestination)this.destination;

    // or it might have ended at a pin,
    } else if (this.destination instanceof Pin) {
      pin = (Pin)this.destination;

        // which is merely a proxy to the real signal destination.
      sigDst = (SignalDestination)pin.getLinkSprite();
    }

    // okay, now unplug me from the signal destination.
    if (sigDst != null) {
      sigDst.setConnection(null);
    }
    super.dispose();
  }
}


/**
 * $Log: Connection.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

Controller.java

```
/*
 * Controller Class.
 * $Id: Controller.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.ctl;
import robotworld.gui.*;
import java.awt.*;
import java.util.*;

/**
 * Provides a multi port sprite implementation which can handle
 * local and remote sources and destinations.  Remote objects are
 * handled by setting the <code>remoteFlag</code> to true.
 * Local objects are handled by setting the <code>remoteFlag</code>
 * to false.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
```

```
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: Controller.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */
public class Controller extends MultiPortSprite {

  /**
   * A flag indicating remote children are of interest when true,
   * or local children are of interest when false.
   */
  boolean remoteFlag = false;

  /**
   * The public no-argument construcor is essential.
   */
  public Controller() {
  }

  /**
   * Set the remote flag to the given value.
   *
   * @param flag the value to set.
   */
  public void setRemoteFlag(boolean flag) {
    this.remoteFlag = flag;
  }

  /**
   * Get the value of the remote flag.
   *
   * @return the value of the remote flag.
   */
  public boolean getRemoteFlag() {
    return this.remoteFlag;
  }

  public boolean isSource(Sprite spr) {
    if (this.remoteFlag) {
      return (spr instanceof RemoteSource);
    } else {
      return (spr instanceof SignalSource);
    }
  }

  public boolean isDestination(Sprite spr) {
    if (this.remoteFlag) {
      return (spr instanceof RemoteDestination);
    } else {
      return (spr instanceof SignalDestination);
    }
  }
}

/**
 * $Log: Controller.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

Directable.java

```
/*
 * Directable Interface.
 * $Id: Directable.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.ctl;
import robotworld.gui.*;

/**
 * Directable components are remotely connectable.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *                                                                             20
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: Directable.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public interface Directable extends Connectable {
}

/**
 * $Log: Directable.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh                               30
 * Original Version.
 *
 */
```

MultiPortSprite.java

```
/*
 * Multi Port Sprite Abstract Class.
 * $Id: MultiPortSprite.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.ctl;
import robotworld.gui.*;
import java.awt.*;
import java.util.*;

/**
 * A sprite that looks like a integrated circuit (vaguely).
 * The pins along the upper edge are equally spaced, one for every             20
 * child sprite which is a destination according to <code>isDestination</code>.
```

```
 * The pins along the bottom edge are equally spaced, one for every
 * child sprite which is a source according to <code>isSource</code>.
 * The layout will reconfigure whenever one of these children is added
 * or removed.  Clicking on a pin will yield the corresponding child.
 * Clicking on the central "wafer" will yield the multi port sprite
 * itself, allowing the whole assembly to be moved around as a unit.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *                                                                    30
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: MultiPortSprite.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public abstract class MultiPortSprite extends LabelSprite {

  /**
   * The sprites represented as pins along the upper edge.
   */
  protected Vector destinations = new Vector();                        40

  /**
   * The sprites represented as pins along the lower edge.
   */
  protected Vector sources = new Vector();

  /**
   * Maps sources (or destinations) to pins.
   * The source (or destination) is the key, the pin is the value.
   */                                                                  50
  protected Hashtable pinOuts = new Hashtable();

  /**
   * A public no-argument constructor is essential.
   */
  public MultiPortSprite() {
  }

  public String toString() {
    String str = super.toString();                                     60
    str += "    Sources:  "+this.getNumSources()+"\n";
    str += "    Destinations:  "+this.getNumDestinations()+"\n";
    return str;
  }

  public int getNumDestinations() {
    return this.destinations.size();
  }

  public int getNumSources() {                                         70
    return this.sources.size();
  }

  /**
   * Evaluate whether the given sprite is a source---for the
   * purposes of layout.
   *
   * @param spr the sprite to evaluate.
   * @return true if the given sprite is a source, false otherwise.
   */                                                                  80
  public abstract boolean isSource(Sprite spr);

  /**
   * Evaluate whether the given sprite is a destination---for the
   * purposes of layout.
   *
   * @param spr the sprite to evaluate.
   * @return true if the given sprite is a destination, false otherwise.
   */
```

```
public abstract boolean isDestination(Sprite spr);

public void addChild(Sprite child) {
  super.addChild(child);
  if (this.isSource(child)) {
    this.addSource(child);
  }
  else if (this.isDestination(child)) {
    this.addDestination(child);
  }
}

public void removeChild(Sprite child) {
  super.removeChild(child);
  if (this.isSource(child)) {
    this.removeSource(child);
  }
  else if (this.isDestination(child)) {
    this.removeDestination(child);
  }
}

public void addSource(Sprite src) {
  Pin pin = new Pin();
  pin.setLinkSprite(src);
  pin.setBounds(new Rectangle(0,0,Pin.PINWIDTH,bounds.height));

  this.sources.addElement(src);
  this.pinOuts.put(src, pin);
  this.layoutSources();
}

public void addDestination(Sprite dst) {
  Pin pin = new Pin();
  pin.setLinkSprite(dst);
  pin.setBounds(new Rectangle(0,0,Pin.PINWIDTH,bounds.height));

  this.destinations.addElement(dst);
  this.pinOuts.put(dst,pin);
  this.layoutDestinations();
}

/**
 * Resets the pin graphics for all sources.
 */
public void layoutSources() {
  // for each source, reset its pin's relative offset
  int n = this.getNumSources();
  for (int i=0; i<n; i++) {
    Pin tmpPin = (Pin)this.pinOuts.get(this.getSourceAt(i));
    int x = (2*i+1-n)*bounds.width/(2*n);
    int y = bounds.height;
    tmpPin.setRelativeOffset(new Point(x,y));
    tmpPin.setLocation(this.getLocation());
  }
}

/**
 * Resets the pin graphics for all destinations.
 */
public void layoutDestinations() {
  // for each dest, reset its pin's relative offset
  int n = this.getNumDestinations();
  for (int i=0; i<n; i++) {
    Pin tmpPin = (Pin)this.pinOuts.get(this.getDestinationAt(i));
    int x = (2*i+1-n)*bounds.width/(2*n);
    int y = -bounds.height;
    tmpPin.setRelativeOffset(new Point(x,y));
```

```
      tmpPin.setLocation(this.getLocation());
    }
  }

  public void setLocation(int x, int y) {
    super.setLocation(x, y);

    int numD = this.getNumDestinations();
    for (int i=0; i<numD; i++) {
      Pin tmpPin = (Pin)this.pinOuts.get(this.getDestinationAt(i));
      tmpPin.setLocation(x, y);
    }

    int numS = this.getNumSources();
    for (int i=0; i<numS; i++) {
      Pin tmpPin = (Pin)this.pinOuts.get(this.getSourceAt(i));
      tmpPin.setLocation(x, y);
    }
  }

  public Sprite find(Point p, boolean specialMode) {

    Rectangle r = this.getBounds();
    if (r != null && r.contains(p)) {
      return this;

    } else {
      int x = p.x-r.x;
      int y = p.y-r.y;
      int h = r.height;
      int w = r.width;
      int numD = this.getNumDestinations();
      int numS = this.getNumSources();
      if (x < 0 || x > w) {
        return null;

      } else if (y > -h && y < 0 && numD > 0) {
        Pin pin = (Pin)this.pinOuts.get(this.getDestinationAt(numD*x/w));
        return pin.find(p, specialMode);

      } else if (y > h && y < 2*h && numS > 0) {
        Pin pin = (Pin)this.pinOuts.get(this.getSourceAt(numS*x/w));
        return pin.find(p, specialMode);

      } else {
        return null;
      }
    }
  }

  public void paint(Graphics g) {
    super.paint(g);

    int numD = this.getNumDestinations();
    for (int i=0; i<numD; i++) {
      Pin tmpPin = (Pin)this.pinOuts.get(this.getDestinationAt(i));
      tmpPin.paint(g);
    }

    int numS = this.getNumSources();
    for (int i=0; i<numS; i++) {
      Pin tmpPin = (Pin)this.pinOuts.get(this.getSourceAt(i));
      tmpPin.paint(g);
    }
  }

  public Sprite getDestinationAt(int index) {
    return (Sprite)this.destinations.elementAt(index);
```

```
  }

  public Sprite getSourceAt(int index) {
    return (Sprite)this.sources.elementAt(index);
  }

  public void removeSource(Sprite whom) {
    this.sources.removeElement(whom);
    this.layoutSources();
  }

  /**
   * Removes the given destination and adjusts the layout.
   *
   * @param whom the sprite to remove.
   */
  public void removeDestination(Sprite whom) {
    this.destinations.removeElement(whom);
    this.layoutDestinations();
  }

  public void dispose() {
    int i=0;
    int count=this.getNumSources();
    for (i=count-1; i>=0; i--) {
      this.getSourceAt(i).dispose();
    }
    count = this.getNumDestinations();
    for (i=count-1; i>=0; i--) {
      this.getDestinationAt(i).dispose();
    }
    super.dispose();
  }
}

/**
 * $Log: MultiPortSprite.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

Pin.java

```
/*
 * Pin Class.
 * $Id: Pin.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.ctl;
import robotworld.gui.*;
import java.awt.*;

/**
```

```
 * Provides a visual proxy to its link sprite.  Pins are usually
 * manipulated by other sprites in order to appear attached to them.
 * A Pin is a sprite that:                                              20
 * <ul>
 * <li>maintains a link sprite which it reveals upon a find()
 *     -- but only when the canvas is in its specialMode.
 * <li>cannot be deleted, edited, dragged, or selected,
 *     unless some other sprite tells it to.
 * </ul>
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu                        30
 * @version $Id: Pin.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class Pin extends Sprite
{
  /**
   * Default width of a pin, in pixels.
   */
  public static final int PINWIDTH = 5;
                                                                        40
  /**
   * Public no-argument constructor.
   */
  public Pin() {
  }

  public void setLinkSprite(Sprite spr) {
    // two-way references.
    super.setLinkSprite(spr);
    spr.setLinkSprite(this);                                            50
  }

  // return the sprite that contains the given point,
  // or null if there is no such sprite.
  public Sprite find(Point p, boolean specialMode) {
    if (specialMode && (super.find(p, specialMode) != null)) {
      return this.getLinkSprite();
    } else {
      return null;
    }                                                                   60
  }

  public void paint(Graphics g) {
    if (this.bounds == null) return;
    g.setColor(Color.lightGray);
    g.fill3DRect(bounds.x, bounds.y,
                 bounds.width, bounds.height, !selected);
  }
}
                                                                        70
/**
 * $Log: Pin.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

RedirectEntry.java

```
/*
 * Redirect Entry Interface.
 * $Id: RedirectEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.ctl;
import robotworld.net.*;

/**
 * Provides a way to inform a remote listener that it should listen
 * elsewhere.  It allows a user to connect and disconnect representations
 * of remote sources and destinations in the GUI, and have a message sent
 * off to the appropriate remote party to announce the change.                 20
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: RedirectEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public interface RedirectEntry extends WorkspaceEntry {

  /**                                                                          30
   * A verb indicating a redirection announcement.
   */
  public static final String REDIRECT = "Redirect";
}

/**
 * $Log: RedirectEntry.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */                                                                           40
```

RemoteConnection.java

```
/*
 * Remote Connection Class.
 * $Id: RemoteConnection.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.ctl;
```

```
import robotworld.ide.*;
import robotworld.gui.*;
import robotworld.net.*;
import java.awt.*;

/**
 * Provides a thread-safe remote communication mechanism between two sprites,
 * with visual connectivity in the canvas as well. It resamples or drops
 * incoming values, depending on the speed at which values are generated
 * (by a <code>RemoteSource</code>) versus the speed which they are sampled
 * (by a <code>RemoteDestination</code>). If generation is faster, some
 * values will be dropped. If sampling is faster, some values will appear
 * multiple times.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: RemoteConnection.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class RemoteConnection extends ConnectorSprite
implements Directable {

  /**
   * Remember who's talking and who's listening.
   */
  protected Object redirectSubject, redirectClient;

  /**
   * A parameterless constructor is essential.
   */
  public RemoteConnection() {
  }

  public void addChild(Sprite child) {
    if (child instanceof RemoteSource) {
      this.setSource(child);
    } else if (child instanceof RemoteDestination) {
      this.setDestination(child);
    }
  }

  /**
   * Performs a blackboard write that notifies the <code>redirectClient</code>
   * to poll for messages from the <code>redirectSubject</code>,
   * using redirection.
   */
  protected void writeRedirectEntry() {
    try {
      Blackboard bb = Workspace.getBlackboard();
      RedirectEntry redirect = (RedirectEntry)Workspace.produce(
        RedirectEntry.class);
      redirect.setSubject(this.redirectSubject);
      redirect.setClient(this.redirectClient);
      bb.write(redirect);
    } catch (Exception x) {
    }
  }

  // this is called after the first mouse click.
  public void setSource(Sprite src) {
    if (src instanceof RemoteSource) {
      RemoteSource remoteSrc = (RemoteSource)src;
      this.redirectSubject = remoteSrc.getUID();

      // check whether pins are more appropriate
      if (this.getParent() != src.getParent()) {
        super.setSource(src.getLinkSprite());
```

```
          } else {
            super.setSource(src);
          }
        }
      }

      // this is called after the second mouse click.
      // tells the client to start polling.
      public void setDestination(Sprite dst) {
        if (dst instanceof RemoteDestination) {
          RemoteDestination remoteDst = (RemoteDestination)dst;
          this.redirectClient = remoteDst.getUID();
          this.writeRedirectEntry();

          // check whether pins are more appropriate
          if (this.getParent() != dst.getParent()) {
            super.setDestination(dst.getLinkSprite());
          } else {
            super.setDestination(dst);
          }
        }
      }

      // tells the client to stop polling.
      public void dispose() {
        this.redirectSubject = null;
        this.writeRedirectEntry();
        super.dispose();
      }
    }

    /**
     * $Log: RemoteConnection.java,v $
     * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
     * Original Version.
     *
     */
```

RemoteDestination.java

```
    /*
     * Remote Destination Interface.
     * $Id: RemoteDestination.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
     *
     * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
     * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
     * CS101 homepage</a> or email <las@ai.mit.edu>.
     *
     * Copyright (C) 1999 Massachusetts Institute of Technology.
     * Please do not redistribute without permission.
     */

    package robotworld.ctl;

    /**
     * An object playing the role of a remote destination in a remote connection
     * operation.  RemoteDestinations are often Sources.  Sounds weird,
     * but just remember that remote relationships are just the duals of
     * internal relationships.  (Where did you think those packets were
     * coming from, anyways?)
     * <p>
```

```
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: RemoteDestination.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public interface RemoteDestination
{
  /**
   * Gets the Unique Identifier of the remote destination
   *
   * @return the unique identifier string of the remote destination.
   */
  public abstract Object getUID();
}

/**
 * $Log: RemoteDestination.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

RemoteSource.java

```
/*
 * Remote Source Interface.
 * $Id: RemoteSource.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.ctl;

/**
 * An object playing the role of a remote source in a remote connection
 * operation.  RemoteSources are often Destinations.  Sounds weird,
 * but just remember that remote relationships are just the duals of
 * internal relationships.  (Where did you think those packets were
 * going, anyways?)
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: RemoteSource.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public interface RemoteSource
{
  /**
   * Gets the Unique Identifier of the remote source.
   *
   * @return the unique identifier string of the remote source.
   */
```

```
  public abstract Object getUID();
}

/**
 * $Log: RemoteSource.java,v $                                          40
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

SensorPort.java

```
/*
 * Sensor Port Class.
 * $Id: SensorPort.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                       10
 */

package robotworld.ctl;
import robotworld.gui.*;
import robotworld.ide.*;
import robotworld.net.*;
import java.io.*;

/**
 * Provides a means of polling remote sensors, and broadcasting the data  20
 * to local connections.  At startup, and every time the SensorPort
 * (or any of its parents) changes name, the SensorPort registers its
 * interest in any <code>REDIRECT</code> messages addressed to it.
 * The Subject indicates which Sensor to poll from until the next
 * <code>REDIRECT</code>.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: SensorPort.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $  30
 */

public class SensorPort extends SignalSource
implements RemoteDestination, BlackboardListener, Runnable {

  /**
   * The period, in milliseconds, between calls to <code>act</code>.
   */
  protected long period = 100;
                                                                         40
  /**
   * The unique identifier of the source from which to poll.
   */
  protected Object sourceUID;

  /**
   * The sensor port's unique identifier.
   */
  protected Object uid;
```

```
                                                                        50
/**
 * Public no-argument constructor.  Kicks off an animacy.
 */
public SensorPort() {
  new Thread(this).start();
}

/**
 * Notifies the blackboard of this sprite's existence, so that it
 * may respond to redirect events.  This method should be called          60
 * anytime the label, or the parent's label, changes.
 */
public void init() {
  try {
    Blackboard bb = Workspace.getBlackboard();
    RedirectEntry notice = (RedirectEntry)Workspace.produce(
      RedirectEntry.class);
    notice.setSubject(null);
    notice.setClient(this.getUID());
    bb.listen(notice, this);                                            70
  } catch (Exception x) {
  }
}

/**
 * Gets the unique identifier for this sensor port.
 * If <code>uid</code> is non-null, simply returns it.
 * Otherwise, it starts with this sprite and climbs up
 * the sprite tree, concatenating sprite labels seperated
 * by a colon, until it reaches root.  The result is                      80
 * a UID string of the form:<br><pre>
 *    my name : parent's name : ... : root sprite's name</pre>
 *
 * @return the unique identifier object for this sprite.
 */
public Object getUID() {
  if (this.uid == null) {
    String tempUID = this.getLabel();
    Sprite tempSprite = null;
    for (tempSprite = this.getParent();                                 90
        tempSprite != null;
        tempSprite = tempSprite.getParent()) {
      tempUID += ":"+tempSprite.getLabel();
    }
    this.uid = tempUID;
  }
  return this.uid;
}

/**                                                                    100
 * Gets the source UID from which to poll.
 *
 * @return the source UID object.
 */
public Object getSourceUID() {
  return this.sourceUID;
}

public void notify(BlackboardEvent e)
throws BlackboardException {                                           110

  // the source of the event is the entry.
  RedirectEntry notice = (RedirectEntry)e.getSource();

  // poll for data from this guy until told otherwise.
  this.sourceUID = notice.getSubject();
}
```

```
  /**
   * Calls init, then continuously calls act every <code>period</code>
   * milliseconds.
   */
  public void run() {
    // notify the blackboard of your existence
    init();

    while (true) {
      act();
      try {
        Thread.sleep(this.period);
      } catch (InterruptedException e) {
      }
    }
  }

  /**
   * Polls for the latest data using the source UID.
   * If successful, updates the internal value.
   */
  public void act() {
    if (this.getSourceUID() == null) return;
    try {

        // poll for the latest data with the given Source UID
      Blackboard bb = Workspace.getBlackboard();
      SignalEntry request = (SignalEntry)Workspace.produce(
        SignalEntry.class);
      request.setSubject(this.getSourceUID());
      SignalEntry response = (SignalEntry)bb.read(request);

      // if successful, update internal value
      if (response != null) {
        this.setSample(response.getSample());
      }

    } catch (Exception x) {
    }
  }
}

/**
 * $Log: SensorPort.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

SignalDestination.java

```
/*
 * Signal Destination Class.
 * $Id: SignalDestination.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
```

```
 * Please do not redistribute without permission.
 */

package robotworld.ctl;
import robotworld.gui.*;

/**
 * A sprite which provides a means of reading from a single
 * local connection.  Looks like a piece of free-floating text.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: SignalDestination.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class SignalDestination extends LabelSprite
implements Destination {

  /**
   * The one and only connection.
   */
  private Connection connection;

  /**
   * A parameterless constructor is essential.
   */
  public SignalDestination() {
    this.noEmboss = true;
  }

  /**
   * Gets the one and only connection.
   *
   * @return the one and only connection.
   */
  public Connection getConnection() {
    return this.connection;
  }

  /**
   * Sets the one and only connection.
   *
   * @param cnxn the one and only connection.
   */
  public void setConnection(Connection cnxn) {
    this.connection = cnxn;
  }

  /**
   * Gets the current sample from the connection.
   * If the connection is null, returns null.
   *
   * @return the current sample object from the connection, if any.
   */
  public Object getSample() {
    if (this.connection == null) return null;
    else return this.connection.getSample();
  }

  public void dispose() {
    if (this.connection != null) {
      this.connection.dispose();
    }
    super.dispose();
  }
}
```

```
/**
 * $Log: SignalDestination.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

SignalEntry.java

```
/*
 * Signal Entry Interface.
 * $Id: SignalEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.ctl;
import robotworld.net.*;

/**
 * Provides a way of sending and receiving timestamped samples of
 * a dynamically changing signal.  Both the sampled quantity and
 * the timestamp are of type <code>Object</code> for maximum generality,
 * which means that implementing classes may need to provide some
 * conversion and utility methods.  For instance, a simple implementing
 * class might use Doubles for the sample, and Integers for the
 * timestamp.  A more complex implementing class might use an
 * <em>N</em>-dimensional vector for the sample, and some special
 * agreed-upon constant to indicate that only the freshest value is desired.
 * <p>
 * <em>Caveat:</em> due to the nature of the blackboard, queries based
 * on timestamp ordering are not possible.  For example, one cannot ask for
 * any timestamp greater than a given value using comparison operators.
 * Instead, one must accept whatever the wildcard yields,
 * or specify an exact timestamp object to look for.  This is possible
 * when one can easily predict the next value of the timestamp, for
 * instance by adding one to an integer timestamp.  There are probably
 * more clever ways, involving distributed data structures customized
 * to the traversal needs of the querying agents.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: SignalEntry.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public interface SignalEntry extends WorkspaceEntry
{
  /**
   * A verb that can be used to help distinguish signal entries from other
   * types of entries.
   */
  public static final String SAMPLE = "Sample";

  /**
```

```
    * Sets the sampled value of the signal.
    *
    * @param sample the sampled value of the signal.
    */
   public void setSample(Object sample);

   /**
    * Gets the sampled value of the signal.
    *
    * @return the sampled value of the signal.
    */
   public Object getSample();

   /**
    * Sets the timestamp of the sample.
    *
    * @param timestamp the timestamp of the sample.
    */
   public void setTimestamp(Object timestamp);

   /**
    * Gets the timestamp of the sample.
    *
    * @return the timestamp of the sample.
    */
   public Object getTimestamp();
}

/**
 * $Log: SignalEntry.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

SignalSource.java

```
/*
 * Signal Source Class.
 * $Id: SignalSource.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.ctl;
import robotworld.gui.*;
import java.util.*;

/**
 * A sprite which provides a means of broadcasting to a set of
 * local connections.  Looks like a piece of free-floating text.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: SignalSource.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
```

```
*/

public class SignalSource extends LabelSprite
implements Source {

  /**
   * A bunch of connections.
   */
  protected Vector connections = new Vector();

  /**
   * Public no-argument constructor.
   */
  public SignalSource() {
    this.noEmboss = true;
  }

  /**
   * Gets the number of connections.
   *
   * @return the number of connections.
   */
  public int getNumConnections() {
    return this.connections.size();
  }

  /**
   * Gets the connections with the specified index.
   *
   * @param index the index of the connection to be returned
   * @return the connection with the specified index
   */
  public Connection getConnectionAt(int index) {
    return (Connection)this.connections.elementAt(index);
  }

  /**
   * Adds the given connection to this source.
   *
   * @param cnxn the connection to be added.
   */
  public void addConnection(Connection cnxn) {
    this.connections.addElement(cnxn);
  }

  /**
   * Removes the given connection from this source.
   *
   * @param cnxn the connection to be removed.
   */
  public void removeConnection(Connection cnxn) {
    this.connections.removeElement(cnxn);
  }

  /**
   * Sets the sample object for all current connections.
   *
   * @param val the sample object.
   */
  public void setSample(Object val) {
    int count=this.getNumConnections();
    for (int i=count-1; i>=0; i--) {
      this.getConnectionAt(i).setSample(val);
    }
  }

  public void dispose() {
    int count=this.getNumConnections();
```

```
    for (int i=count-1; i>=0; i--) {
      this.getConnectionAt(i).dispose();
    }
    super.dispose();
  }
}

/**                                                                    100
 * $Log: SignalSource.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

C.7 The robotworld.sim package

Actuator.java

```
/*
 * Actuator Abstract Class.
 * $Id: Actuator.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                      10
 */

package robotworld.sim;
import robotworld.net.*;
import robotworld.ide.*;
import robotworld.gui.*;
import robotworld.ctl.*;
import java.awt.*;
import java.awt.geom.*;
import java.util.*;                                                    20

/**
 * Provides a means of polling remote actuator ports, and broadcasting the
 * commands to local connections.  At startup, and every time the Actuator
 * (or any of its parents) changes name, the Actuator registers its
 * interest in any <code>REDIRECT</code> messages addressed to it.
 * The SUBJECT indicates which Sensor to poll from until the next
 * <code>REDIRECT</code>.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.           30
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: Actuator.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public abstract class Actuator extends MultiPortSprite
implements ForceLaw, RemoteDestination, BlackboardListener
{
```

```
/**
 * The rigid body which this actuator is attached to.
 */
protected RigidBody body;

/**
 * The latest sample of the actuator command signal.
 */
protected double sample;

/**
 * The unique identifier of the source from which to poll.
 */
protected Object sourceUID;

/**
 * The actuator's unique identifier.
 */
protected Object uid;

/**
 * Public no-argument constructor.
 */
public Actuator() {
}

public void setLabel(String label) {
  super.setLabel(label);

  // we want UID to get recalculated in init(), so set it to null.
  this.uid = null;

  // init tells the blackboard to forward all messages to the new UID.
  init();
}

/**
 * Notifies the blackboard of this sprite's existence, so that it
 * may respond to redirect events.  This method is called
 * anytime the label, or any of its parent's labels, changes.
 */
public void init() {
  try {
    Blackboard bb = Workspace.getBlackboard();
    RedirectEntry notice = (RedirectEntry)Workspace.produce(
      RedirectEntry.class);
    notice.setSubject(null);
    notice.setClient(this.getUID());
    bb.listen(notice, this);
  } catch (Exception x) {
  }
}

/**
 * Gets the source UID from which to poll.
 *
 * @return the source UID object.
 */
public Object getSourceUID() {
  return this.sourceUID;
}

public void notify(BlackboardEvent e)
throws BlackboardException {

  // the source of the event is the entry.
  RedirectEntry notice = (RedirectEntry)e.getSource();
```

```
    // poll for data from this guy until told otherwise.
    this.sourceUID = notice.getSubject();
  }

  public boolean isSource(Sprite spr) {
    return false;
  }

  public boolean isDestination(Sprite spr) {
    return false;
  }

  /**
   * Sets the rigid body which this actuator is attached to.
   *
   * @param b the rigid body which this actuator is attached to.
   */
  public void setRigidBody(RigidBody b) {
    this.body = b;
  }

  /**
   * Gets the rigid body which this actuator is attached to.
   *
   * @return the rigid body which this actuator is attached to.
   */
  public RigidBody getRigidBody() {
    return this.body;
  }

  /**
   * Gets the unique identifier for this actuator.
   * If <code>uid</code> is non-null, simply returns it.
   * Otherwise, it starts with this sprite and climbs up
   * the sprite tree, concatenating sprite labels seperated
   * by a colon, until it reaches root.  The result is
   * a UID string of the form:<br><pre>
   *    my name : parent's name : ... : root sprite's name</pre>
   *
   * @return the unique identifier object for this sprite.
   */
  public Object getUID() {
    if (this.uid == null) {
      String tempUID = this.getLabel();
      Sprite tempSprite = null;
      for (tempSprite = this.getParent();
           tempSprite != null;
           tempSprite = tempSprite.getParent()) {
        tempUID += ":"+tempSprite.getLabel();
      }
      this.uid = tempUID;
    }
    return this.uid;
  }

  /**
   * Gets the current sample of the actuator command signal.
   *
   * @return the current sample of the actuator command signal.
   */
  public double getSample() {
    return this.sample;
  }

  /**
   * Sets the current sample of the actuator command signal.
   *
   * @param s the current sample of the actuator command signal.
```

```
   */
  public void setSample(double s) {
    this.sample = s;
  }

  /**
   * Polls for the latest data using the source UID.
   * If successful, updates the internal value.
   */
  public void act() {
    if (this.getSourceUID() == null) return;
    try {

        // poll for the latest data with the given Source UID
      Blackboard bb = Workspace.getBlackboard();
      SignalEntry request = (SignalEntry)Workspace.produce(
        SignalEntry.class);
      request.setSubject(this.getSourceUID());
      SignalEntry response = (SignalEntry)bb.read(request);

      // if successful, update internal value
      if (response != null) {
        Double d = (Double)response.getSample();
        this.setSample(d.doubleValue());
      }
    } catch (Exception x) {
    }
  }
}

/**
 * $Log: Actuator.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

Beacon.java

```
/*
 * Beacon Class.
 * $Id: Beacon.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.sim;

/**
 * Provides a model of an omnidirectional point light source.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: Beacon.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */
```

```
public class Beacon extends Actuator implements Emitter
{
  /**
   * The relative offset, in pixels, from the rigid body.
   */
  public double[] offset;

  /**
   * The relative angle, in radians, from the rigid body.
   */
  public double orientation;

  /**
   * Public no-argument constructor.
   */
  public Beacon() {
    this.offset = new double[2];
    this.setSample(400.0d);
  }

  public double[] getPosition() {
    return DoubleVector.add(this.getRigidBody().getPosition(),
                            this.offset);
  }

  public double getOrientation() {
    return this.getRigidBody().getOrientation() + this.orientation;
  }

  public void apply() {
  }

  public double getIntensity(Detector d) {
    // intensity is directly proportional to the commanded value
    // and inversely proportional to the square of the distance.

    double distance = DoubleVector.magnitude(
      DoubleVector.subtract(this.getPosition(), d.getPosition()));

    // note that the denominatator can never be zero.
    return this.getSample()/(1+distance*distance);
  }

  public double getAngle(Detector d) {
    // construct a vector from the detector to the emitter.
    // return the vector's angular component minus the detector's angle.

    double[] detectorToEmitter = DoubleVector.subtract(
      this.getPosition(), d.getPosition());
    double myOrientation = Math.atan2(
      detectorToEmitter[1], detectorToEmitter[0]);
    double phi = myOrientation - d.getOrientation();

    // restrict phi to [-PI,PI]
    while (phi > Math.PI) phi -= Math.PI*2;
    while (phi < -Math.PI) phi += Math.PI*2;

    return phi;
  }
}

/**
 * $Log: Beacon.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

CollisionDetector.java

```
/*
 * Collision Detector Class.
 * $Id: CollisionDetector.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.sim;
import java.util.*;
import java.awt.*;
import java.awt.event.*;

/**
 * Provides a simple means of collision detection.
 * <p>                                                                         20
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: CollisionDetector.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class CollisionDetector
{
  /**
   * The particle system this collision detector is yoked to.                  30
   */
  protected ParticleSystem pSystem;

  /**
   * The rigid bodies being monitored for collisions.
   */
  protected Vector bodies = new Vector();

  /**
   * Current list of contact forces.                                           40
   */
  protected Vector contactForces = new Vector();

  /**
   * Seperating planes being monitored for collisions.
   * Meant to track component bounds only--this list is a subset of
   * <code>bodies</code>.  Cleared on component resize events.
   */
  protected Vector planes = new Vector();
                                                                              50
  /**
   * The minimum distance between two rigid bodies, in pixels,
   * before a collision or contact has occurred.
   */
  protected double epsilon;

  /**
   * Public no-argument constructor.
```

```
   */
public CollisionDetector() {
   this.setEpsilon(1.0);
}

/**
 * Sets the particle system which this collision detector is yoked to.
 * Also grabs the component which is rendering the particle system,
 * passes it to <code>createPlanes</code>, and adds a component listener
 * which will call <code>createPlanes</code> every time the component
 * is resized.
 *
 * @param p the particle system which this collision detector is yoked to.
 */
public void setParticleSystem(ParticleSystem p) {
   this.pSystem = p;
   final Component comp = p.getComponent();
   this.createPlanes(comp);
   comp.addComponentListener(new ComponentAdapter() {
     public void componentResized(ComponentEvent e) {
       CollisionDetector.this.createPlanes(comp);
     }
  });
}

/**
 * Generates four planes from the component bounds.
 * Does nothing if the component is null.
 */
public void createPlanes(Component comp) {
   // if there is no component, don't bother
   if (comp == null) return;

   // clear the list
   for (int i=this.getNumPlanes()-1; i>=0; i--) {
     this.removePlane(this.getPlaneAt(i));
   }

   // generate four planes from component bounds
   Rectangle bounds = comp.getBounds();
   this.addPlane(new SeperatingPlane(
     0, 0, 1.0, 0.0));
   this.addPlane(new SeperatingPlane(
     0, 0, 0.0, 1.0));
   this.addPlane(new SeperatingPlane(
     bounds.width-1, bounds.height-1, -1.0, 0.0));
   this.addPlane(new SeperatingPlane(
     bounds.width-1, bounds.height-1, 0.0, -1.0));
}

/**
 * Sets the minimum distance between two rigid bodies, in pixels,
 * before a collision or contact has occurred.
 *
 * @param eps the double value of epsilon
 */
public void setEpsilon(double eps) {
   this.epsilon = eps;
}

/**
 * Gets the minimum distance between two rigid bodies, in pixels,
 * before a collision or contact has occurred.
 *
 * @return the double value of epsilon
 */
public double getEpsilon() {
   return this.epsilon;
```

```
}

/**
 * Scan the geometry, update data structures, and
 * return the earliest collision.
 *
 * @return the contact force of the earliest collision.
 */
public ContactForce findEarliestCollision() {

  double maxDistance = 0, tempDistance = 0;
  ContactForce earliestCollision = null;
  int n = this.getNumRigidBodies();

  // for simplicity, just clear all contacts & rescan.
  this.contactForces.removeAllElements();

  // nasty O(n^2) collision check.
  for (int i=0; i<n; i++) {
    RigidBody a = this.getRigidBodyAt(i);

    // only test each pair once.
    for (int j=i+1; j<n; j++) {
      RigidBody b = this.getRigidBodyAt(j);

      // walls and walls shouldn't collide
      if (a.getNormal()!=null && b.getNormal()!=null) {
        continue;
      }

      // the Calculation.
      tempDistance = RigidBody.getInterPenetrationDistance(a,b);
      if (Math.abs(tempDistance) <= this.epsilon) {

        // they are either contacting or colliding,
        // the contact force instance will decide which.
        this.addContactForce(
          new ContactForce(a, b, ContactForce.COLLIDING));

      } else if (tempDistance < 0) {

        // they are interpenetrating
        ContactForce f = new ContactForce(
          a, b, ContactForce.PENETRATING);
        this.addContactForce(f);

        if (tempDistance < maxDistance) {

          // only keep the earliest one
          earliestCollision = f;
          maxDistance = tempDistance;
        }
      }
    }
  }

  return earliestCollision;
}

// Contact Force methods
public int getNumContactForces() {
  return this.contactForces.size();
}

public ContactForce getContactForceAt(int index) {
  return (ContactForce)this.contactForces.elementAt(index);
}
```

```
  public void addContactForce(ContactForce c) {
    this.contactForces.addElement(c);
  }

  public void removeContactForce(ContactForce c) {
    this.contactForces.removeElement(c);                                    200
  }

  // Seperating Plane methods.
  // Note that planes are rigid bodies too.
  public int getNumPlanes() {
    return this.planes.size();
  }

  public SeperatingPlane getPlaneAt(int index) {
    return (SeperatingPlane)this.planes.elementAt(index);                   210
  }

  public void addPlane(SeperatingPlane p) {
    this.planes.addElement(p);
    this.bodies.addElement(p);
  }

  public void removePlane(SeperatingPlane p) {
    this.planes.removeElement(p);
    this.bodies.removeElement(p);                                           220
  }

  // Rigid Body methods
  public int getNumRigidBodies() {
    return this.bodies.size();
  }

  public RigidBody getRigidBodyAt(int index) {
    return (RigidBody)this.bodies.elementAt(index);
  }                                                                         230

  public void addRigidBody(RigidBody b) {
    this.bodies.addElement(b);
  }

  public void removeRigidBody(RigidBody b) {
    this.bodies.removeElement(b);
  }

}                                                                           240

/**
 * $Log: CollisionDetector.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

ContactForce.java

```
/*
 * Contact Force Class.
 * $Id: ContactForce.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
```

```
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                          10
 */

package robotworld.sim;

/**
 * Provides a simple means of collision reaction.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu                            20
 * @version $Id: ContactForce.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class ContactForce implements ForceLaw {

  /**
   * A kind of reaction in which the bodies are overlapping.
   */
  public static final int PENETRATING = 0;
                                                                          30
  /**
   * A kind of reaction in which the bodies are close enough
   * to be considered in contact, and are moving towards each
   * other.
   */
  public static final int COLLIDING = 1;

  /**
   * A kind of reaction in which the bodies are close enough to be
   * considered in contact, and are moving slowly enough to be              40
   * considered at rest with respect to each other.
   */
  public static final int RESTING = 2;

  /**
   * The reference body, by convention.
   */
  public RigidBody bodyA;

  /**                                                                     50
   * The other body.
   */
  public RigidBody bodyB;

  /**
   * The type of reaction; one of <code>PENETRATING, COLLIDING, or
   * RESTING</code>.
   */
  public int reaction;
                                                                          60
  /**
   * The velocity threshold, in pixels per time step, used to arbitrate
   * between <code>RESTING</code> and <code>COLLIDING</code> states.
   */
  public double threshold = 0.25;

  /**
   * The constant of restoration, allowed to vary from 0 to 1.
   * A value of 0 indicates a completely inelastic collision.
   * A value of 1 indicates a completely elastic collision.                 70
   */
  public double constantOfRestoration = 0.5;
```

```
/**
 * This constructor takes the two given rigid bodies, and the
 * suggested reaction type, and decides if another reaction type
 * is more appropriate in the final analysis.
 *
 * @param a the rigid body for reference
 * @param b the other rigid body                                          80
 * @param reaction the reaction type, guesses are okay.
 */
public ContactForce(RigidBody a, RigidBody b, int reaction) {
  this.bodyA = a;
  this.bodyB = b;

  if (reaction == PENETRATING) {
    this.reaction = PENETRATING;
  } else {
                                                                          90
    // reaction type depends on relative velocity
    double[] vRelative = DoubleVector.subtract(a.getVelocity(),
                                       b.getVelocity());
    double vel = 0;

    if (a.getNormal()==null && b.getNormal() == null) {
      double magn = DoubleVector.magnitude(vRelative);
      if (magn==0) {
        vel = 0;
      } else {                                                            100
        double[]normal = DoubleVector.multiply(1/magn, vRelative);
        vel = DoubleVector.dotProduct(vRelative, normal);
      }
    } else if (b.getNormal()==null) {
      vel = -DoubleVector.dotProduct(vRelative, a.getNormal());
    } else {
      vel = DoubleVector.dotProduct(vRelative, b.getNormal());
    }

    if (Math.abs(vel) <= this.threshold) {                                110
      this.reaction = RESTING;
    } else if (vel < -this.threshold) {
      this.reaction = COLLIDING;
    }
  }
}

/**
 * Apply forces and torques to the two bodies, as appropriate
 * to their reaction type.                                               120
 */
public void apply() {
  RigidBody a = this.bodyA;
  RigidBody b = this.bodyB;

  // reaction type depends on relative velocity
  double[] pRelative = DoubleVector.subtract(a.getPosition(),
                                     b.getPosition());
  double[] vRelative = DoubleVector.subtract(a.getVelocity(),
                                     b.getVelocity());                  130
  double normalDistance = RigidBody.getInterPenetrationDistance(a,b);
  double normalVelocity = 0, normalForce = 0;
  double[] force = null, pos = null, normal = null;


  // length of their normal components
  if (a.getNormal()==null && b.getNormal()==null) {
    // both particles
    double magn = DoubleVector.magnitude(pRelative);
    if (magn==0) return; // I hate degeneracy                            140
    normal = DoubleVector.multiply(1/magn, pRelative);
```

```
    pos = a.getPosition(); // by convention
    force = a.getForce();  // by convention
    normalVelocity = DoubleVector.dotProduct(vRelative,
                                             normal);
    normalForce = DoubleVector.dotProduct(force,
                                          normal);

  } else if (a.getNormal() == null) {
    // b is the wall                                                    150
    normal = b.getNormal();
    pos = a.getPosition();
    force = a.getForce();
    normalVelocity = DoubleVector.dotProduct(vRelative,
                                             normal);
    normalForce = DoubleVector.dotProduct(force,
                                          normal);

  } else if (b.getNormal() == null) {
    // a is the wall                                                    160
    normal = a.getNormal();
    pos = b.getPosition();
    force = b.getForce();
    normalVelocity = -DoubleVector.dotProduct(vRelative,
                                              normal);
    normalForce = DoubleVector.dotProduct(force,
                                          normal);

  } else { // two walls shouldn't collide
    return;                                                            170
  }

  switch (this.reaction) {
  case ContactForce.PENETRATING:

    // zero the normal component of the position
    DoubleVector.add(pos,
                     DoubleVector.multiply(-normalDistance, normal),
                     pos);
    break;                                                             180

  case ContactForce.RESTING:
    if (normalForce < 0) {
      // zero the normal component of the force
      DoubleVector.add(force,
        DoubleVector.multiply(-normalForce, normal),
        force);

      // inform any of A's touch sensors
      for (int i=a.getNumSensors()-1; i>=0; i--) {                     190
        Sensor sensor = (Sensor)a.getSensorAt(i);
        if (sensor.medium.equals(Sensor.CONTACT_FORCE)) {
          sensor.sample += sensor.filter(normalForce,0);
        }
      }

      // inform any of B's touch sensors
      for (int i=b.getNumSensors()-1; i>=0; i--) {
        Sensor sensor = (Sensor)b.getSensorAt(i);
        if (sensor.medium.equals(Sensor.CONTACT_FORCE)) {              200
          sensor.sample += sensor.filter(normalForce,0);
        }
      }
    }
    break;

  case ContactForce.COLLIDING:

    // apply velocity impulse along normal:
```

```
        //                                                                    210
        // when constantOfRestoration = 0 (completely inelastic),
        //      the normal component is zeroed.
        // when constantOfRestoration = 1 (completely elastic),
        //      the normal component is reversed.
        double denom = a.getInverseMass() + b.getInverseMass();
        double impulseMagnitude =
          -(1+constantOfRestoration) * normalVelocity / denom;

        // contribution to A
        double[] aVelocityChange = DoubleVector.multiply(                   220
          impulseMagnitude * a.getInverseMass(), normal);
        DoubleVector.add(a.getVelocity(),
                         aVelocityChange,
                         a.getVelocity());

        // inform any of A's touch sensors
        for (int i=a.getNumSensors()-1; i>=0; i--) {
          Sensor sensor = (Sensor)a.getSensorAt(i);
          if (sensor.medium.equals(Sensor.CONTACT_FORCE)) {
            sensor.sample += sensor.filter(impulseMagnitude,0);             230
          }
        }

        // contribution to B
        double[] bVelocityChange = DoubleVector.multiply(
          -impulseMagnitude * b.getInverseMass(), normal);
        DoubleVector.add(b.getVelocity(),
                         bVelocityChange,
                         b.getVelocity());
                                                                            240
        // inform any of B's touch sensors
        for (int i=b.getNumSensors()-1; i>=0; i--) {
          Sensor sensor = (Sensor)b.getSensorAt(i);
          if (sensor.medium.equals(Sensor.CONTACT_FORCE)) {
            sensor.sample += sensor.filter(impulseMagnitude,0);
          }
        }
        break;
      }
    }                                                                       250
}

/**
 * $Log: ContactForce.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

Detector.java

```
/*
 * Detector Interface.
 * $Id: Detector.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
```

```
 * Please do not redistribute without permission.                    10
 */

package robotworld.sim;

/**
 * Provides the minimal interface necessary for an emitter to
 * determine the intensity and angle of incident radiation.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *                                                                    20
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: Detector.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public interface Detector
{
  /**
   * Gets the position of the detector.
   *
   * @return the position vector, in absolute coordinates.            30
   */
  public abstract double[] getPosition();

  /**
   * Gets the orientation of the detector.
   *
   * @return the orientation, in absolute coordinates.
   */
  public abstract double getOrientation();
}                                                                     40

/**
 * $Log: Detector.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

DiffeqSolver.java

```
/*
 * Differential Equation Solver Interface.
 * $Id: DiffeqSolver.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                     10
 */

package robotworld.sim;

/**
 * Provides an interface to a differential equation solver.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu                       20
```

```
 * @version $Id: DiffeqSolver.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public interface DiffeqSolver
{
  /**
   * Solves for the state of the dynamical system at the next time step,
   * using numerical integration.
   *
   * @param d the dynamical system to update.                                  30
   * @param timeStep the size of the time step to use.
   */
  public void solve(DynamicalSystem d, double timeStep);
}

/**
 * $Log: DiffeqSolver.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *                                                                           40
 */
```

DifferentialDrive.java

```
/*
 * Differential Drive Class.
 * $Id: DifferentialDrive.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                            10
 */

package robotworld.sim;
import robotworld.net.*;
import robotworld.ide.*;
import robotworld.gui.*;
import robotworld.ctl.*;
import java.awt.*;
import java.awt.geom.*;
import java.util.*;                                                          20

/**
 * An actuator which yokes together two motors in a differential-drive
 * fashion to allow simple tank-like steering.  Graphically, it provides
 * exactly two pins, one for each motor.  As a force law, it interprets
 * the angular velocity command values for the motors and generates the
 * appropriate forces and torques to drive the vehicle.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *                                                                           30
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: DifferentialDrive.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class DifferentialDrive extends Actuator {

  /**
```

```
 * Constants used to uniquely identify each of the two wheels.
 */
public static final String LEFT = "LEFT", RIGHT = "RIGHT";

/**
 * The radius of the wheels, in pixels.  The wheels are considered
 * to be perpendicular to the computer screen.
 */
protected double wheelRadius;

/**
 * The length of the axle, in pixels.
 */
protected double axleLength;

/**
 * The proportional gain of the controller.  This number should be
 * high enough to get a fast response, but not so high that it causes
 * instability for the numerical integration scheme.
 */
protected double controllerGain = 100;

/**
 * The maximum linear acceleration possible.  Prevent being flung off
 * to infinity.
 */
protected double maxLinearAccel = 10;

/**
 * The maximum angular acceleration permitted.  Prevents being spun into
 * oblivion.
 */
protected double maxAngularAccel = Math.PI/16.0;

/**
 * Convenience field for type casting.  Redundant with <code>body</code>.
 */
protected Vehicle2D vehicle;

/**
 * Nested actuators.
 */
protected Actuator leftMotor, rightMotor;

// this was made a private method class so that the outside world
// wouldn't try to instantiate it.
private class Motor extends Actuator {

  private Motor(String s) {
    super();
    this.setLabel(s);
  }

  // all the work is done by DifferentialDrive.apply().
  public void apply() {
  }
}

/**
 * Public no-argument constructor.
 */
public DifferentialDrive() {
  super();
}

public boolean isSource(Sprite spr) {
  return false;
}
```

```
public boolean isDestination(Sprite spr) {
  return (spr instanceof Motor);
}

/**
 * Sets the rigid body for this differential drive.
 * Anything other than a <code>Vehicle2D</code> is ignored.
 * Also sets the <code>axleLength</code> and <code>wheelRadius</code>
 * to be proportional to the radius of the vehicle.
 *
 * @param body the 2D vehicle which this differential drive is attached to.
 */
public void setRigidBody(RigidBody body) {
  if (body instanceof Vehicle2D) {
    super.setRigidBody(body);
    this.vehicle = (Vehicle2D)body;

    // and set default parameters
    this.axleLength = this.vehicle.getRadius() * 2.0;
    this.wheelRadius = this.vehicle.getRadius() / 2.0;
  }
}

// we can't add pins to the gui until we know the layout.
public void setComponent(Component canvas) {
  // layout the label
  super.setComponent(canvas);

  // notify the blackboard of the motors' existence
  this.leftMotor = new Motor(LEFT);
  this.rightMotor = new Motor(RIGHT);

  // add pins to the gui
  this.leftMotor.setParent(this);
  this.rightMotor.setParent(this);
}

/**
 * Apply forces and torques based on the actuator command values.
 * Polls the left and right motor pins for the desired angular velocity
 * of the left and right wheels.  This translates into a desired
 * linear and angular velocity for the vehicle as a whole.
 * Uses a feedback controller to generate linear and angular accelerations
 * proportional to the difference between desired and actual velocity,
 * so that the vehicle approaches the desired parameters within a handful
 * of time steps.
 */
public void apply() {
  if (this.vehicle == null ||
      this.leftMotor == null ||
      this.rightMotor == null) {
    return;
  }
  this.leftMotor.act(); //HACK
  this.rightMotor.act(); //HACK

  // desired angular velocity of the left and right wheels
  double vLeft = this.leftMotor.getSample();
  double vRight = this.rightMotor.getSample();

  // desired linear velocity (pixels/sec), and angular velocity (rad/sec).
  double vLinear = (vRight+vLeft) * wheelRadius/2.0;
  double vAngular = (vRight-vLeft) * wheelRadius / (axleLength/2.0);

  // actual linear and angular velocity
  double v = DoubleVector.magnitude(this.vehicle.getVelocity());
```

```
    double w = this.vehicle.getAngularVelocity();

    // proportional feedback gives linear and angular acceleration
    double aLinear = controllerGain*(vLinear - v);
    double aAngular = controllerGain*(vAngular - w);

    // clamp the linear and angular acceleration commands,                    180
    // otherwise things get ridiculous
    if (aLinear > maxLinearAccel) aLinear = maxLinearAccel;
    else if (aLinear < -maxLinearAccel) aLinear = -maxLinearAccel;

    if (aAngular > maxAngularAccel) aAngular = maxAngularAccel;
    else if (aAngular < -maxAngularAccel) aAngular = -maxAngularAccel;

    // now translate this into forces and torques
    this.process(aLinear, aAngular);
  }                                                                          190

  /**
   * Given a linear and angular acceleration, alter the force
   * and torque on the vehicle's center of mass.  This includes
   * the centripetal acceleration that couples linear and angular
   * velocity to produce curved trajectories.  Without it, vehicles
   * would always move in a straight line, despite their orientation.
   *
   * @param linearAccel the linear acceleration
   * @param angularAccel the angular acceleration                            200
   */
  public void process(double linearAccel, double angularAccel) {
    if (this.vehicle == null) return;
    double[] totalForce = this.vehicle.getForce();
    double m = 1/this.vehicle.getInverseMass();

    // linear force = m * a
    double[] linearForce = DoubleVector.multiply(
      m * linearAccel,
      this.vehicle.getHeading());                                            210
    DoubleVector.add(totalForce, linearForce, totalForce);

    // centripetal accel = w^2*r = v^2/r = w*v  (since v = r*w),
    // where r is the radius of curvature
    double w = this.vehicle.getAngularVelocity();
    double v = DoubleVector.magnitude(this.vehicle.getVelocity());
    double[] centripetalForce = DoubleVector.multiply(
      m*w*v,
      this.vehicle.getHeadingPerp());
    DoubleVector.add(totalForce, centripetalForce, totalForce);            220

    // torque = I * alpha
    double I = 1/this.vehicle.getInverseInertia();
    double t = I * angularAccel;
    this.vehicle.torque += t;
  }
}

/**
 * $Log: DifferentialDrive.java,v $                                        230
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

DoubleVector.java

```
/*
 * Double Vector Class.
 * $Id: DoubleVector.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.sim;

/**
 * Provides math utilities for double precision N-dimensional vectors.
 * These vectors are manipulated as arrays of doubles.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *                                                                             20
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: DoubleVector.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public final class DoubleVector
{
  /**
   * Returns the vector dot product of v1 and v2.
   *
   * @param v1 an array of doubles                                              30
   * @param v2 another array of doubles
   * @return v1 . v2
   */
  public static double dotProduct(double[] v1, double[] v2) {
    double result=0;
    for (int i=v1.length-1; i>=0; i--) {
      result += v1[i] * v2[i];
    }
    return result;
  }                                                                            40

  /**
   * Returns the magnitude of the vector v.
   *
   * @param v an array of doubles
   * @return ||v||
   */
  public static double magnitude(double[] v) {
    return Math.sqrt(DoubleVector.dotProduct(v,v));
  }                                                                            50

  /**
   * Returns the unit normal of the vector v, or null if v is the
   * zero vector.
   *
   * @param v an array of doubles
   * @return v / ||v||, or null if v is the zero vector.
   */
  public static double[] normal(double[] v) {
    double magn = DoubleVector.magnitude(v);                                   60
    if (magn==0) return null;
    else return DoubleVector.multiply(1/magn, v);
  }
```

```
/*============================*
 * Methods to conserve memory *
 *============================*/

/**
 * Scales the vector v by the scalar s, and writes the result into
 * vector sv.
 *
 * @param v an array of doubles
 * @param s the scaling factor
 * @param sv sv = s * v
 */
public static void multiply(double[] sv, double s, double[] v) {
  for (int i = sv.length-1; i>=0; i--) {
    sv[i] = s * v[i];
  }
}

/**
 * Adds the vectors v1 and v2, and writes the result into sum.
 *
 * @param v1 an array of doubles
 * @param v2 another array of doubles
 * @param sum sum = v1 + v2;
 */
public static void add(double[] sum, double[] v1, double[] v2) {
  for (int i = sum.length-1; i>=0; i--) {
    sum[i] = v1[i] + v2[i];
  }
}

/**
 * Subtracts vector v2 from v1, and writes the result into diff.
 *
 * @param v1 an array of doubles
 * @param v2 another array of doubles
 * @param diff diff = v1 - v2;
 */
public static void subtract(double[] diff, double[] v1, double[] v2) {
  for (int i = diff.length-1; i>=0; i--) {
    diff[i] = v1[i] - v2[i];
  }
}

/**
 * Sets every element of vector v to the constant c.
 *
 * @param c a constant
 * @param v v[..] = c;
 */
public static void set(double[] v, double c) {
  for (int i = v.length-1; i>=0; i--) {
    v[i] = c;
  }
}

/**
 * Copies the elements of source into the vector dest, starting at
 * dest[offset].  Returns the position following the last element written,
 * which can be used as an offset for the next embedding operation.
 *
 * @param source an array of doubles
 * @param offset the position within dest to begin writing
 * @param dest dest[offset] = source;
 * @return the offset for the next embedding operation.
 */
public static int embed(double[] dest, int offset, double[] source) {
  System.arraycopy(source, 0, dest, offset, source.length);
```

```
    return offset+source.length;
}

/**
 * Copies the elements of source into the vector dest, starting
 * at source[offset].  Returns the position following the last element read,
 * which can be used as an offset for the next extracting operation.
 *
 * @param source an array of doubles
 * @param offset the position within source to begin reading
 * @param dest dest = source[offset]
 * @return the offset for the next extracting operation.
 */
public static int extract(double[] source, int offset, double[] dest) {
    System.arraycopy(source, offset, dest, 0, dest.length);
    return offset+dest.length;
}

/*=========================*
 * Methods to waste memory *
 *=========================*/

/**
 * Scales the vector v by the scalar s, and returns the result.
 *
 * @param v an array of doubles
 * @param s the scaling factor
 * @return s * v;
 */
public static double[] multiply(double s, double[] v) {
    double[] sv = new double[v.length];
    DoubleVector.multiply(sv, s, v);
    return sv;
}

/**
 * Adds the vectors v1 and v2, and returns the result.
 *
 * @param v1 an array of doubles
 * @param v2 another array of doubles
 * @return v1 + v2;
 */
public static double[] add(double[] v1, double[] v2) {
    double[] sum = new double[v1.length];
    DoubleVector.add(sum, v1, v2);
    return sum;
}

/**
 * Subtracts vector v2 from v1, and returns the result.
 *
 * @param v1 an array of doubles
 * @param v2 another array of doubles
 * @param return v1 - v2;
 */
public static double[] subtract(double[] v1, double[] v2) {
    double[] diff = new double[v1.length];
    DoubleVector.subtract(diff, v1, v2);
    return diff;
}

/*====================*
 * Debugging methods *
 *====================*/

/**
 * Returns a human-readable string version of the vector v.
 * Used for debugging.
```

```
    *
    * @return a human-readable string version of the vector v.
    */
  public static String toString(double[] v) {
    String str="[ ";
    int i=0;
    for (i=0; i<v.length; i++) {
      str += v[i];
      if (i < v.length-1) {
        str += ", ";
      }
    }
    return str + " ]";
  }
}

/**
 * $Log: DoubleVector.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

DynamicalSystem.java

```
/*
 * Dynamical System Interface.
 * $Id: DynamicalSystem.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.sim;

/**
 * Objects implementing this interface behave according to an
 * Ordinary Differential Equation (ODE).  This ODE is expressed
 * in the form:<br><pre>
 *  .
 *  x = f(x,t)  where x is the state vector,
 *                    t is the time,
 *              and f is an expression for the derivative.</pre>
 * <p>
 * This interface allows an ODE Solver to determine the
 * current state of the system, calculate the next state,
 * and then advance the system to its next state.
 * <p>
 * The ODE Solver is not bound to advance in uniform
 * time steps, or even forward in time.  The reversibility
 * of the system will depend on the stability and accuracy
 * of the ODE Solver as well as the presence of stochastic
 * variables in the system components.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
```

```
 * @version $Id: DynamicalSystem.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
*/

public interface DynamicalSystem
{
  /**
   * Gets the dimension of the system.
   *
   * @return the dimension of the system.
   */
  public int getDimension();

  /**
   * Gets the derivative vector.
   *
   * @return the derivative vector.
   */
  public double[] getDerivative();

  /**
   * Gets the current state vector.
   *
   * @return the current state vector.
   */
  public double[] getStateVector();

  /**
   * Sets the current state vector.
   *
   * @param state the current state vector.
   */
  public void setStateVector(double[] state);

  /**
   * Gets the current time.
   *
   * @return the current time.
   */
  public double getTime();

  /**
   * Sets the current time.
   *
   * @param time the current time.
   */
  public void setTime(double time);
}

/**
 * $Log: DynamicalSystem.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

EarthGravity.java

```
/*
 * Earth Gravity Class.
 * $Id: EarthGravity.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
```

```
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.sim;
import robotworld.ide.*;
import robotworld.gui.*;
import java.awt.*;

/**
 * Imposes a constant acceleration upon all inhabitants of RobotWorld.
 * The result is a simplistic approximation to the gravity field of the        20
 * surface of the Earth, which RobotWorld users should immediately recognise.
 * The acceleration is a vector, so it can be imposed in any desired
 * direction.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: EarthGravity.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */
                                                                            30
public class EarthGravity extends Sprite implements ForceLaw
{
  /**
   * The particle system this force law is yoked to.
   */
  protected ParticleSystem pSystem;

  /**
   * The acceleration to impose upon all inhabitants of RobotWorld.
   */                                                                       40
  protected double[] acceleration;

  /**
   * A trick to conserve memory during numerous force computations.
   */
  private double[] force;

  /**
   * Public no-argument constructor.
   */                                                                       50
  public EarthGravity() {
    this.acceleration = new double[2];
    this.force = new double[2];
  }

  /**
   * Sets the particle system this force law is yoked to.
   *
   * @param p the particle system to yoke this force law to.
   */                                                                       60
  public void setParticleSystem(ParticleSystem p) {
    this.pSystem = p;
  }

  public void apply() {

     // for all particles
    int n = this.pSystem.getNumParticles();
    for (int i=0; i<n; i++) {
      Particle p = this.pSystem.getParticleAt(i);                           70

        // f = ma
```

```
        DoubleVector.multiply(this.force,
                              1/p.getInverseMass(), this.acceleration);
        DoubleVector.add(p.getForce(),
                         this.force, p.getForce());
      }
    }

    public void paint(Graphics g) {
    }
}

/**
 * $Log: EarthGravity.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

Emitter.java

```
/*
 * Emitter Interface.
 * $Id: Emitter.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.sim;

/**
 * Objects that emit radiation implement this interface.  Note that the
 * emitter does not reveal its position and orientation, which allows
 * for effects such as light sources at infinity or ambient light levels.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: Emitter.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */
public interface Emitter
{
  /**
   * Returns the intensity of the radiation at the detector,
   * which is a function of the position and orientation of the detector.
   *
   * @param d the detector to evaluate
   * @return the double value of the intensity at the detector.
   */
  public abstract double getIntensity(Detector d);

  /**
   * Returns the orientation of the emitter with respect to the detector.
   *
   * @param d the detector to evaluate
   * @return the orientation of the emitter with respect to the detector.
   */
```

```
  public abstract double getAngle(Detector d);
}

/**
 * $Log: Emitter.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

EulersMethod.java

```
/*
 * Euler's Method Class.
 * $Id: EulersMethod.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.sim;

/**
 * Provides an Euler solver.  This method of numerical integration
 * has <em>O(h^2)</em> error, where <em>h</em> is the time step.
 * In other words, it's fast, but it's not that accurate.  This kind of
 * solver tends to be unstable in the presence of large derivatives.
 * Using a smaller time step usually helps.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: EulersMethod.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class EulersMethod implements DiffeqSolver
{
  /**
   * Public no-argument constructor.
   */
  public EulersMethod() {
  }

  public void solve(DynamicalSystem system, double timeStep) {
    double[] deriv = null, state = null;

    // x(t+dt) ~= x(t) + xDot(t) * dt
    deriv = system.getDerivative();
    DoubleVector.multiply(deriv, timeStep, deriv);
    state = system.getStateVector();
    DoubleVector.add(state, deriv, state);

    system.setStateVector(state);
    system.setTime(system.getTime()+timeStep);
  }
}
```

```
/**
 * $Log: EulersMethod.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

ForceLaw.java

```
/*
 * Force Law Interface.
 * $Id: ForceLaw.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.sim;

/**
 * Objects wanting to impose physical laws need to implement this interface.
 * This includes applying forces and torques to rigid bodies.  The implementing
 * object is responsible for keeping track of the affected bodies, and
 * manipulating their force and torque vectors whenever <code>apply</code>
 * is called by the particle system.  This allows for force laws which affect
 * all objects (such as earth gravity or viscous drag), two objects
 * (such as spring laws), or one object (such as an actuator).
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: ForceLaw.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public interface ForceLaw
{
  /**
   * Apply forces and torques to the rigid bodies affected by this
   * force law.
   */
  public void apply();
}

/**
 * $Log: ForceLaw.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

Particle.java

```
/*
 * Particle Class.
 * $Id: Particle.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.sim;
import robotworld.ide.*;
import java.awt.*;
import java.util.*;

/**
 * Provides a simple model of a particle with a defined radius and
 * orientation.  Drawn as a point, although it maintains a bounding
 * rectangle for user interactivity.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: Particle.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class Particle extends RigidBody
{
  protected double[] position, velocity, acceleration;
  protected double[] force, linearMomentum;
  protected double inverseMass, inverseInertia, time, radius;
  protected double orientation, angularVelocity, angularAcceleration;
  protected double torque, angularMomentum;
  protected double[] heading, headingPerp;
  protected int dimension, clickRadius=9;
  public static final int X=0, Y=1;
  protected static Random rng = new Random();

  public Particle() {
    this.setDimension(2);
    double mass = 40.0;
    this.orientation = this.rng.nextDouble() * 2.0 * Math.PI;
    this.setRotationVectors();
    this.inverseMass = 1/mass;
    this.setBounds(new Rectangle());
  }

  /**
   * Sets the rotation vectors.  Uses <code>orientation</code> to determine
   * the vectors <code>heading</code> and <code>headingPerp</code> such
   * that their cross product points out of the computer screen.
   * This means that a positive angular velocity corresponds to
   * counter-clockwise rotation.
   */
  public void setRotationVectors() {
    double theta = this.orientation;
    double cos = Math.cos(theta);
    double sin = Math.sin(theta);
    this.heading[X] = cos;
    this.heading[Y] = -sin;
    this.headingPerp[X] = -sin;
    this.headingPerp[Y] = -cos;
```

```
}

public void setLocation(Point p) {
  super.setLocation(p.x, p.y);
  this.position[X]=p.x;
  this.position[Y]=p.y;                                                    70
}

public String toString() {
  String str = " D"+this.dimension;
  str += "\t M"+this.inverseMass;
  str += "\t T"+this.time;
  str += "\n P"+DoubleVector.toString(this.position);
  str += "\n V"+DoubleVector.toString(this.velocity);
  str += "\n A"+DoubleVector.toString(this.acceleration);
  str += "\n F"+DoubleVector.toString(this.force);                         80
  return str;
}

public int getDimension() {
  return this.dimension;
}

public void setDimension(int dim) {
  this.heading = new double[dim];
  this.headingPerp = new double[dim];                                      90
  this.position = new double[dim];
  this.velocity = new double[dim];
  this.acceleration = new double[dim];
  this.force = new double[dim];
  this.linearMomentum = new double[dim];
  this.dimension = dim*2+2; // HACK for (x,y,theta) space
}

public double[] getPosition() {
  return this.position;                                                   100
}

public double[] getVelocity() {
  return this.velocity;
}

public double[] getAcceleration() {
  return this.acceleration;
}
                                                                          110
public double[] getNormal() {
  return null;
}

public double[] getForce() {
  return this.force;
}

public double getInverseMass() {
  return this.inverseMass;                                                120
}

public double getRadius() {
  return this.radius;
}

public double getInverseInertia() {
  return this.inverseInertia;
}
                                                                          130
public double getTorque() {
  return this.torque;
```

```
}

public double[] getLinearMomentum() {
  return this.linearMomentum;
}

public double getAngularMomentum() {
  return this.angularMomentum;                                          140
}

public double getAngularVelocity() {
  return this.angularVelocity;
}

public double getAngularAcceleration() {
  return this.angularAcceleration;
}
                                                                        150
public double getOrientation() {
  return this.orientation;
}

public double[] getHeading() {
  return this.heading;
}

public double[] getHeadingPerp() {
  return this.headingPerp;                                              160
}

// copy into state vector, starting at offset.
public void getStateVector(double[] state, int offset) {
  int index = offset;
  index = DoubleVector.embed(state, index, this.position);
  state[index++] = this.orientation;
  index = DoubleVector.embed(state, index, this.linearMomentum);
  state[index++] = this.angularMomentum;
}                                                                       170

// copy out of state vector, starting at offset.
public void setStateVector(double[] state, int offset) {
  int index = offset;
  index = DoubleVector.extract(state, index, this.position);
  this.orientation = state[index++];
  index = DoubleVector.extract(state, index, this.linearMomentum);
  this.angularMomentum = state[index++];
}
                                                                        180
public void getDerivative(double[] deriv, int offset) {
  int index = offset;

  // xDot = v = P / M
  DoubleVector.multiply(this.velocity, this.inverseMass,
                        this.linearMomentum);
  index = DoubleVector.embed(deriv, index, this.velocity);

  // thetaDot = omega = L / I
  this.angularVelocity = this.inverseInertia *                          190
                         this.angularMomentum;
  deriv[index++] = this.angularVelocity;

  // pDot = f, a = f/m
  index = DoubleVector.embed(deriv, index, this.force);
  DoubleVector.multiply(this.acceleration, this.inverseMass, this.force);

  // lDot = tau, alpha = tau / I
  deriv[index++] = this.torque;
  this.angularAcceleration = this.inverseInertia * this.torque;         200
```

```
  }

  public void paint(Graphics g) {
    int xPos = (int)this.position[X];
    int yPos = (int)this.position[Y];
    this.bounds.setBounds(xPos-clickRadius, yPos-clickRadius,
                          clickRadius*2, clickRadius*2);
    g.setColor(Color.blue);
    g.drawLine(xPos, yPos, xPos, yPos);
  }                                                                    210
}

/**
 * $Log: Particle.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

ParticleSystem.java

```
/*
 * Particle System Class.
 * $Id: ParticleSystem.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                       10
 */

package robotworld.sim;
import robotworld.gui.*;
import robotworld.net.*;
import robotworld.ide.*;
import java.awt.*;
import java.awt.geom.*;
import java.util.*;
                                                                       20
/**
 * Models a set of rigid bodies subject to physical force laws
 * as a dynamical system, complete with collisions and a radiation model.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: ParticleSystem.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */
                                                                       30
public class ParticleSystem extends LabelSprite
  implements Runnable, DynamicalSystem
{

  /**
   * The particles being animated by this particle system.
   */
  protected Vector particles = new Vector();

  /**                                                                  40
```

```
 * The force laws which affect this particle system.
 */
protected Vector forceLaws = new Vector();

/**
 * The current simulation time.  Can be converted to real time in seconds
 * (only approximately) by multiplying by the conversion factor
 * <code>1000*period/timeStep</code>.
 */
protected double time;

/**
 * The simulation time step.  This needs to be small enough
 * to obtain good accuracy and stability, but large enough that
 * the vehicles actually appear to move.
 */
protected double timeStep=.005;

/**
 * The default sleep period between calls to act, in milliseconds.
 * The real time equivalent of one time step in simulation time.
 */
protected long period = 50L;

/**
 * The dimension of the particle system, should be proportional to
 * the number of particles.
 */
protected int dimension;

/**
 * A flag indicating whether the component bounds are set.
 * Setting this flag indicates that the collision detector has had
 * time to set up bounding behavior, and the simulation can now proceed.
 */
protected boolean componentBoundsSet;

/**
 * Current state vector of the dynamical system.  Continually overwritten.
 */
protected transient double[] state;

/**
 * Current derivative of the state vector.  Continually overwritten.
 */
protected transient double[] deriv;

/**
 * Every particle system needs a differential equation solver.
 */
protected transient DiffeqSolver ode;

/**
 * Collisions handled here.
 */
protected transient CollisionDetector collider;

/**
 * Used to triplicate the ellipse and electron when painted.
 */
private transient AffineTransform transform;

/**
 * Used to make the electron streak part of an atomic symbol when painted.
 */
private transient Arc2D arc;

/**
```

```
 * Used to stroke the electron streak part of an atomic symbol when painted.
 */
private transient Stroke stroke;
/**
 * Used to make the elliptical part of an atomic symbol when painted.
 */
private transient Ellipse2D ellipse;

/**
 * Used to animate the atomic symbol.
 */
private int frames = 0;

/**
 * Public no-argument constructor.  Creates a solver and a collision
 * detector.  Sets the initial conditions and kicks off an animacy.
 */
public ParticleSystem() {
  this.noEmboss = true;
  this.insets = new Insets(MARGIN*2, MARGIN*2, MARGIN*2, MARGIN*2);
  this.ode = new RungeKutta();
  this.collider = new CollisionDetector();
  this.setTime(0.0);
  new Thread(this).start();
}

public String toString() {
  int n = this.getNumParticles();
  String str = n + "-Particle System\n----------------";

  for (int i=0; i<n; i++) {
    Particle p = this.getParticleAt(i);
    str += "\n\n" + p.toString();
  }
  return str;
}

public void paint(Graphics g) {
  super.paint(g);

  Rectangle r = this.getBounds();
  Point p = this.getLocation();
  if (r == null || p == null) return;
  if (ellipse == null) ellipse = new Ellipse2D.Double();
  if (arc == null) arc = new Arc2D.Double();
  if (transform == null) transform = new AffineTransform();
  if (stroke == null) stroke = new BasicStroke(
    3.0f, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND);

  // now get down to business, drawing the three atomic orbits.
  Graphics2D g2 = (Graphics2D)g;
  ellipse.setFrame(r.x, r.y, r.width, r.height);
  arc.setArc(r.x, r.y, r.width, r.height, frames*4.0d, 15.0d, Arc2D.OPEN);

  transform.setToRotation(0, p.x, p.y);
  g2.draw(transform.createTransformedShape(ellipse));
  g2.draw(stroke.createStrokedShape(transform.createTransformedShape(arc)));

  transform.setToRotation(Math.PI*2/3, p.x, p.y);
  g2.draw(transform.createTransformedShape(ellipse));
  g2.draw(stroke.createStrokedShape(transform.createTransformedShape(arc)));

  transform.setToRotation(Math.PI*4/3, p.x, p.y);
  g2.draw(transform.createTransformedShape(ellipse));
  g2.draw(stroke.createStrokedShape(transform.createTransformedShape(arc)));
}

public void setComponent(Component c) {
```

```
  super.setComponent(c);
  if (this.collider != null) this.collider.setParticleSystem(this);
  this.componentBoundsSet = true;
}

public synchronized void paintChildren(Graphics g) {
  super.paintChildren(g);
}

public synchronized void addChild(Sprite child) {
  super.addChild(child);
  if (child instanceof Particle) {
    this.addParticle((Particle)child);
  }
}

public synchronized void removeChild(Sprite child) {
  super.removeChild(child);
  if (child instanceof Particle) {
    this.removeParticle((Particle)child);
  }
}

public void addParticle(Particle p) {
  p.setParticleSystem(this);
  this.particles.addElement(p);
  this.setDimension(this.getDimension()+p.getDimension());
  this.collider.addRigidBody(p);
}

public void removeParticle(Particle p) {
  p.setParticleSystem(null);
  this.particles.removeElement(p);
  this.setDimension(this.getDimension()-p.getDimension());
  this.collider.removeRigidBody(p);
}

public int getNumParticles() {
  return this.particles.size();
}

public Particle getParticleAt(int k) {
  return (Particle)this.particles.elementAt(k);
}

public int getNumForceLaws() {
  return this.forceLaws.size();
}

public ForceLaw getForceLawAt(int k) {
  return (ForceLaw)this.forceLaws.elementAt(k);
}

public void addForceLaw(ForceLaw f) {
  this.forceLaws.addElement(f);
}

public void removeForceLaw(ForceLaw f) {
  this.forceLaws.removeElement(f);
}

/*============================*
 * Dynamical System methods *
 *============================*/

public int getDimension() {
  return this.dimension;
}
```

```
public void setDimension(int dim) {
   if (dim > 0) {
      // there is no need to preserve old state.
      // it gets copied over by the particles, anyways.
      this.state = new double[dim];
      this.deriv = new double[dim];
   } else {
      this.state = null;
      this.deriv = null;
   }
   this.dimension = dim;
}

public double[] getStateVector() {
   int n = this.getNumParticles();
   int index = 0;

   for (int i=0; i<n; i++) {
      Particle p = this.getParticleAt(i);
      p.getStateVector(this.state, index);
      index += p.getDimension();
   }
   return this.state;
}

public void setStateVector(double[] state) {
   int n = this.getNumParticles();
   int index = 0;

   for (int i=0; i<n; i++) {
      Particle p = this.getParticleAt(i);
      p.setStateVector(this.state, index);
      index += p.getDimension();
   }

   this.state = state;
}

public double getTime() {
   return this.time;
}

public void setTime(double time) {
   this.time = time;
}

public double[] getDerivative() {
   int n = this.getNumParticles();
   int index = 0;

   for (int i=0; i<n; i++) {
      Particle p = this.getParticleAt(i);
      p.getDerivative(this.deriv, index);
      index += p.getDimension();
   }
   return this.deriv;
}

/*==============================*
 * Core methods of the animacy *
 *==============================*/

 /**
  * Forever calls the act method in a while-true loop.
  * Sleeps for <code>period</code> between each call.
  */
public void run() {
```

```
  while (true) {
        if (this.componentBoundsSet) {
       this.act();
       frames = (frames+1)%90;
        }
    try {
          Thread.sleep(this.period);
    } catch (InterruptedException e) {
    }
  }
}

/**
 * The unit of action.  Performs the following steps:
 * <ul>
 * <li>Calls <code>computeForces</code>
 * <li>asks the solver to advance this particle system by one time step
 * <li>asks the collider to find all contact forces (collisions) that result
 * <li>resolves contact forces by calling their <code>apply</code> methods.
 * </ul>
 */
public synchronized void act() {
  if (this.dimension <= 0) return;

  this.computeForces();
  this.ode.solve(this, this.timeStep);
  ContactForce cf = this.collider.findEarliestCollision();

  int n = this.collider.getNumContactForces();
  for (int i=0; i<n; i++) {
    ContactForce f = this.collider.getContactForceAt(i);
    f.apply();
  }
}

/**
  * Computes forces and torques.  Performs the following steps:
  * <ul>
  * <li>Clears force/torque accumulators
  * <li>Clears sensor accumulators
  * <li>Computes incident radiation using the emitter/detector
  *     radiation model.
  * <li>Calls the <code>apply</code> method of all force laws
  * </ul>
  */
public void computeForces() {

  // clear force (and torque) accumulators
  for (int i=0; i<this.getNumParticles(); i++) {
    Particle particle = this.getParticleAt(i);
    DoubleVector.set(particle.getForce(), 0);
    particle.torque = 0;
    particle.setRotationVectors();

    // clear sensor accumulators
    for (int j=0; j<particle.getNumSensors(); j++) {
      Sensor sensor = particle.getSensorAt(j);
      sensor.act();
      sensor.sample = 0;

      // BEGIN RADIATION MODEL
      // It should really have its own class.
        // However, at the moment it's too intertwined
      // with the single-thread particle system code.
      // Look for the (*) symbol to see what I mean.

      // quick & dirty way to do radiation detection
      if (sensor instanceof Detector) {
```

```
            Detector detector = (Detector)sensor;

            // for all force laws that are emitters as well
            for (int k=0; k<this.getNumForceLaws(); k++) {
              ForceLaw force = this.getForceLawAt(k);
              if (force instanceof Emitter) {
                Emitter emitter = (Emitter)force;

                // Ignore self-luminescence (emitter & detector
                      // embedded in same vehicle).                                    390

                      // (*) Is there a better approach?  One that
                      // doesn't need expose sensorhood/actuatorhood?
                if (force instanceof Actuator) {
                  Actuator actuator = (Actuator)force;
                  if (actuator.getRigidBody()==particle)
                    continue;
                      }

                // contribution at the detector depends on the                       400
                // intensity and angle of emitted radiation
                double intensity = emitter.getIntensity(detector);
                double angle = emitter.getAngle(detector);
                double filtered = sensor.filter(intensity, angle);
                sensor.sample += filtered;
                    }
              }
            }
          // END RADIATION MODEL
        }                                                                            410
      }

      // compute external forces
      for (int i=0; i<this.getNumForceLaws(); i++) {
        ForceLaw force = this.getForceLawAt(i);
        force.apply();
      }
    }
}                                                                                    420

/**
 * $Log: ParticleSystem.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

PhotoSensor.java

```
/*
 * Photo Sensor Class.
 * $Id: PhotoSensor.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                                   10
 */
```

```
package robotworld.sim;
import java.awt.*;

/**
 * Provides a model of a photosensor with a simplistic cone of sensitivity.
 * A gaussian falloff is used to prevent discontinuities which would result
 * in stability problems for the numerical simulation routines.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: PhotoSensor.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class PhotoSensor extends Sensor implements Detector
{
  /**
   * The cone of sensitivity.  This is the angle at which the sensitivity
   * has dropped to <code>1/Math.E</code> from maximal.
   */
  public double coneAngle = Math.PI/4.0;

  /**
   * The offset of this sensor from the rigid body it's attached to.
   */
  public double[] offset;

  /**
   * This sensor's orientation relative to the rigid body it's attached to.
   */
  public double orientation;

  public PhotoSensor() {
    this.medium = Sensor.INCIDENT_RADIATION;
    this.offset = new double[2];
  }

  public double[] getPosition() {
    RigidBody body = this.getRigidBody();
    return DoubleVector.add(body.getPosition(),
                            this.offset);
  }

  public double getOrientation() {
    return this.getRigidBody().getOrientation() + this.orientation;
  }

  public double filter(double intensity, double phi) {
    // gaussians are better: smoother derivatives
    return intensity * gaussian(phi, this.coneAngle);
  }

  /**
   * Returns the gaussian of x with standard deviation sigma.
   *
   * @param x the argument to be evaluated
   * @param sigma the standard deviation
   * @return Math.E ^( -|x| ^2 / sigma^2)
   */
  public static double gaussian(double x, double sigma) {
    if (sigma==0) return 0d;

    // return e ^( -|x|^2 / sigma^2)
    double t = Math.abs(x) / sigma;
    return Math.exp(-t*t);
  }

  // until we can interactively edit the offset vector,
```

```
  // this will have to do.
  public class Left extends PhotoSensor {
    protected double angleOfOffset;
    public Left() {
      angleOfOffset = Math.PI/4;
    }

    public double[] getPosition() {
      RigidBody body = this.getRigidBody();
      double r = body.getRadius();                                          90
      offset[0] =  r*Math.cos(angleOfOffset);
      offset[1] = -r*Math.sin(angleOfOffset);
      return DoubleVector.add(body.getPosition(),
                              this.offset);
    }
  }

  // until we can interactively edit the offset vector,
  // this will have to do.
  public class Right extends PhotoSensor {                                 100
    protected double angleOfOffset;
    public Right() {
      angleOfOffset = -Math.PI/4;
    }

    public double[] getPosition() {
      RigidBody body = this.getRigidBody();
      double r = body.getRadius();
      offset[0] =  r*Math.cos(angleOfOffset);
      offset[1] = -r*Math.sin(angleOfOffset);                               110
      return DoubleVector.add(body.getPosition(),
                              this.offset);
    }
  }
}

/**
 * $Log: PhotoSensor.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.                                                      120
 *
 */
```

RigidBody.java

```
/*
 * Rigid Body Abstract Class.
 * $Id: RigidBody.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                          10
 */

package robotworld.sim;
import robotworld.gui.*;
import java.awt.*;
import java.util.*;
```

```
/**
 * Provides the minimal set of functionality required of all rigid bodies
 * in the particle system.  This includes access to the mechanical
 * state variables, as well as a means of managing embedded sensors and
 * actuators, and utilities for collision detection.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: RigidBody.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public abstract class RigidBody extends Sprite
{
  /**
   * The sensors attached to this rigid body.
   */
  protected Vector sensors = new Vector();

  /**
   * The actuators attached to this rigid body.
   */
  protected Vector actuators = new Vector();

  /**
   * The particle system that is animating this rigid body.
   */
  protected ParticleSystem pSystem;

  /**
   * Gets the velocity of this object.
   *
   * @return the velocity of this object.
   */
  public abstract double[] getVelocity();

  /**
   * Gets the force being applied to this object.
   *
   * @return the force being applied to this object.
   */
  public abstract double[] getForce();

  /**
   * Gets the position vector.  The position <em>(x,y)</em> is defined
   * to be identical to screen coordinates, with (0,0) defined to be the
   * upper left-hand corner, <em>x</em> increasing to the right,
   * <em>y</em> increasing downwards, and units given in pixels.
   *
   * @param the position vector.
   */
  public abstract double[] getPosition();

  /**
   * Gets the acceleration vector.
   *
   * @return the acceleration vector.
   */
  public abstract double[] getAcceleration();

  /**
   * Gets the normal to the surface of this object.
   * Objects may return null if they do not have a preferred direction
   * for a surface normal.
   *
   * @return the normal to the surface of this object.
   */
```

```
public abstract double[] getNormal();

/**
 * Gets the radius of this object.
 *
 * @return the radius of this object.
 */
public abstract double getRadius();

/**
 * Gets the inverse mass of this object.  A zero indicates an
 * object of infinite mass, which cannot be translated by
 * ordinary means.
 *
 * @return the inverse mass of this object.
 */
public abstract double getInverseMass();

/**
 * Gets the inverse inertia of this object.  A zero indicates
 * an object with infinite inertia, which cannot be rotated by
 * ordinary means.
 *
 * @return the inverse inertia of this object.
 */
public abstract double getInverseInertia();

/**
 * Gets the torque being applied to this object.
 *
 * @return the torque being applied to this object.
 */
public abstract double getTorque();

/**
 * Gets the linear momentum of this object.
 *
 * @return the linear momentum of this object.
 */
public abstract double[] getLinearMomentum();

/**
 * Gets the angular momentum of this object.
 *
 * @return the angular momentum of this object.
 */
public abstract double getAngularMomentum();

/**
 * Gets the angular velocity of this object.
 *
 * @return the angular velocity of this object.
 */
public abstract double getAngularVelocity();

/**
 * Gets the angular acceleration of this object.
 *
 * @return the angular acceleration of this object.
 */
public abstract double getAngularAcceleration();

/**
 * Gets the orientation of this object.  The orientation
 * <em>theta</em> is defined to be zero when pointing to the right
 * side of the screen and increasing as the rigid body turns
 * counter-clockwise, with units given in radians.
 *
```

```
 * @return the orientation of this object.
 */
public abstract double getOrientation();

/**
 * Gets the inter-penetration distance between the given bodies.
 * Returns the normal distance of the closest point of A
 * from the surface of B, defined such that:
 * <ul>
 * <li>negative values indicate inter-penetration
 * <li>a value of zero indicate tangency
 * <li>positive values indicate disjoint geometry
 * </ul>
 * @param a a rigid body
 * @param b another rigid body
 * @return the inter-penetration distance between the two bodies.
 */
public static double getInterPenetrationDistance(
  RigidBody a, RigidBody b) {

  double[] diff = DoubleVector.subtract(a.getPosition(),
                                        b.getPosition());
  double distance = 0;

  if (a.getNormal()==null && b.getNormal()==null) {
    // point-point distance ||a-b||
    distance = DoubleVector.magnitude(diff);

  } else if (a.getNormal()==null) {
    // point-plane distance (a-b) . n
    distance = DoubleVector.dotProduct(diff, b.getNormal());

  } else if (b.getNormal()==null) {
    // plane-point distance (b-a) . n
    distance = -DoubleVector.dotProduct(diff, a.getNormal());
  }

  // minus thickness
  return distance - a.getRadius() - b.getRadius();
}

public void addChild(Sprite child) {
  super.addChild(child);
  if (child instanceof Sensor) {
    this.addSensor((Sensor)child);
  }
  if (child instanceof Actuator) {
    this.addActuator((Actuator)child);
  }
}

public void removeChild(Sprite child) {
  super.removeChild(child);
  if (child instanceof Sensor) {
    this.removeSensor((Sensor)child);
  }
  if (child instanceof Actuator) {
    this.removeActuator((Actuator)child);
  }
}

public void addSensor(Sensor s) {
  this.sensors.addElement(s);
  s.setRigidBody(this);
}

public void removeSensor(Sensor s) {
  this.sensors.removeElement(s);
```

```
    s.setRigidBody(null);
  }

  public int getNumSensors() {
    return this.sensors.size();
  }

  public Sensor getSensorAt(int k) {
    return (Sensor)this.sensors.elementAt(k);
  }                                                                    230

  public void addActuator(Actuator a) {
    this.actuators.addElement(a);
    a.setRigidBody(this);
    this.pSystem.addForceLaw(a);
  }

  public void removeActuator(Actuator a) {
    this.actuators.removeElement(a);
    a.setRigidBody(null);                                              240
    this.pSystem.removeForceLaw(a);
  }

  public int getNumActuators() {
    return this.actuators.size();
  }

  public Actuator getActuatorAt(int k) {
    return (Actuator)this.actuators.elementAt(k);
  }                                                                    250

  public void setParticleSystem(ParticleSystem p) {
    this.pSystem = p;
  }

  public ParticleSystem getParticleSystem() {
    return this.pSystem;
  }
}
                                                                       260
/**
 * $Log: RigidBody.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

RungeKutta.java

```
/*
 * Runge Kutta Class.
 * $Id: RungeKutta.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                      10
 */
```

```
package robotworld.sim;

/**
 * Provides a Runge-Kutta 4th order solver. This method of numerical
 * integration has <em>O(h^5)</em> error, where
 * <em>h</em> is the time step.  Compared to an Euler solver,
 * it's slightly slower, but much more accurate.  This kind of
 * solver is much more robust to the presence of large derivatives,
 * but can still become unstable if the time step is too large.
 * Using a smaller time step usually helps.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: RungeKutta.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class RungeKutta implements DiffeqSolver
{
  /**
   * Public no-argument constructor.
   */
  public RungeKutta() {
  }

  public void solve(DynamicalSystem system, double timeStep) {
    double[] f = null, x0 = null;
    double[] k1 = null, k2 = null, k3 = null, k4 = null;
    double t0 = 0;
    double h = 0;

    // k1 = h * f(x0, t0)
    h = timeStep;
    x0 = system.getStateVector();
    t0 = system.getTime();
    f = system.getDerivative();
    k1 = DoubleVector.multiply(h, f);

    // k2 = h * f(x0 + k1 / 2, t0 + h / 2)
    system.setStateVector(
      DoubleVector.add(x0, DoubleVector.multiply(1/2, k1)));
    system.setTime(t0+h/2);
    f = system.getDerivative();
    k2 = DoubleVector.multiply(h, f);

    // k3 = h * f(x0 + k2 / 2, t0 + h / 2)
    system.setStateVector(
      DoubleVector.add(x0, DoubleVector.multiply(1/2, k2)));
    system.setTime(t0+h/2);
    f = system.getDerivative();
    k3 = DoubleVector.multiply(h, f);

    // k4 = h * f(x0 + k3, t0 + h)
    system.setStateVector(DoubleVector.add(x0, k3));
    system.setTime(t0+h);
    f = system.getDerivative();
    k4 = DoubleVector.multiply(h, f);

    // x(t0+h) = x0 + k1 / 6 + k2 / 3 + k3 / 3 + k4 / 6
    system.setStateVector(formula(x0, k1, k2, k3, k4));
    system.setTime(t0+h);
  }

  // wish I didn't have to write it this way.  But using the
  // DoubleVector class would have forced me to write a
  // nested statement seven levels deep -- ugly as well as slow.
  // Why can't Java have lambda expressions built in???
  private static double[] formula(double[] x0, double[] k1, double[] k2,
```

```
                                double[] k3, double[] k4) {
    double[] x = new double[x0.length];
    for (int i=x0.length-1; i>=0; i--) {
      x[i] = x0[i] + (k1[i] + 2*(k2[i]+k3[i]) + k4[i])/6;
    }
    return x;
  }
}

/**                                                                    90
 * $Log: RungeKutta.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

Sensor.java

```
/*
 * Sensor Abstract Class.
 * $Id: Sensor.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                      10
 */

package robotworld.sim;
import robotworld.net.*;
import robotworld.ide.*;
import robotworld.gui.*;
import robotworld.ctl.*;
import java.awt.*;
import java.awt.geom.*;
import java.util.*;                                                    20

/**
 * Provides a means of polling a local connection, and broadcasting the
 * data to an unknown number of remote sensor ports.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: Sensor.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */                                                                    30

public abstract class Sensor extends MultiPortSprite
implements RemoteSource
{
  /**
   * The rigid body which this sensor is attached to.
   */
  protected RigidBody body;

  /**                                                                  40
   * The sensor's unique identifier.
   */
  protected Object uid;
```

```
/**
 * The latest sample of the actuator command signal.
 */
public double sample;

/**
 * The medium of sensation.  Typically <code>CONTACT_FORCE</code> or
 * <code>INCIDENT_RADIATION</code>.
 */
public Object medium;

/**
 * A medium indicating that this sensor measures contact force.
 */
public static final String CONTACT_FORCE = "Contact Force";

/**
 * A medium indicating that this sensor measures incident radiation.
 */
public static final String INCIDENT_RADIATION = "Incident Radiation";

/**
 * Public no-argument constructor.
 */
public Sensor() {
}

public boolean isSource(Sprite spr) {
  return false;
}

public boolean isDestination(Sprite spr) {
  return false;
}

/**
 * Sets the rigid body which this sensor is attached to.
 *
 * @param b the rigid body which this sensor is attached to.
 */
public void setRigidBody(RigidBody b) {
  this.body = b;
}

/**
 * Gets the rigid body which this actuator is attached to.
 *
 * @return the rigid body which this actuator is attached to.
 */
public RigidBody getRigidBody() {
  return this.body;
}

/**
 * Gets the unique identifier for this sensor.
 * If <code>uid</code> is non-null, simply returns it.
 * Otherwise, it starts with this sprite and climbs up
 * the sprite tree, concatenating sprite labels seperated
 * by a colon, until it reaches root.  The result is
 * a UID string of the form:<br><pre>
 *    my name : parent's name : ... : root sprite's name</pre>
 *
 * @return the unique identifier object for this sprite.
 */
public Object getUID() {
  if (this.uid == null) {
    String tempUID = this.getLabel();
```

```
        Sprite tempSprite = null;
        for (tempSprite = this.getParent();
             tempSprite != null;
             tempSprite = tempSprite.getParent()) {
          tempUID += ":"+tempSprite.getLabel();
        }
        this.uid = tempUID;
      }
      return this.uid;                                                    120
    }

    /**
     * Filters the sensor's response for the given intensity and angle.
     *
     * @param intensity the intensity of the sensed quantity
     * @param angle the angular deviation from the primary sensor axis
     * @return the sensor's response as a function of intensity and angle.
     */
    public abstract double filter(double intensity, double angle);        130

    /**
     * Gets the current sample of the sensor data signal.
     *
     * @return the current sample of the sensor data signal.
     */
    public double getSample() {
      return this.sample;
    }
                                                                          140
    /**
     * Sets the current sample of the sensor data signal.
     *
     * @param s the current sample of the sensor data signal.
     */
    public void setSample(double s) {
      this.sample = s;
    }

    /**                                                                   150
     * Polls for the latest data and writes it to the blackboard
     * using this sprite's UID as a subject, making sure to remove
     * the previous entry first.
     */
    public void act() {
      try {
        Blackboard bb = Workspace.getBlackboard();
        SignalEntry update = (SignalEntry)Workspace.produce(SignalEntry.class);
        update.setSubject(this.getUID());
        update.setClient(null);                                           160
        bb.take(update);
        update.setSample(new Double(this.getSample()));
        bb.write(update);
      } catch (Exception x) {
      }
    }
}

/**
 * $Log: Sensor.java,v $                                                  170
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

SeperatingPlane.java

```
/*
 * Seperating Plane Class.
 * $Id: SeperatingPlane.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld.sim;
import java.awt.*;

/**
 * Provides a rigid body type which defines a seperating plane for
 * the purposes of collision detection.  The plane is defined by
 * a normal and a point on the plane.  The normal points in the
 * direction where collision candidates will be found.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: SeperatingPlane.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class SeperatingPlane extends RigidBody {
  public double[] point, normal, temp;
  public double[] acceleration, velocity;
  public double[] force, linearMomentum;
  public double orientation, angularVelocity, angularAcceleration;
  public double torque, angularMomentum;

  /**
   * Public no-argument constructor.
   */
  public SeperatingPlane() {
    point = new double[2];
    normal = new double[2];
    temp = new double[2];
    velocity = new double[2];
    acceleration = new double[2];
    force = new double[2];
    linearMomentum = new double[2];
  }

  /**
   * Convenience constructor which defines a plane using a
   * normal and a point on the plane.
   *
   * @param x the x coordinate of a point on the plane
   * @param y the y coordinate of a point on the plane
   * @param nx the x coordinate of a normal to the plane
   * @param ny the y coordinate of a normal to the plane
   */
  public SeperatingPlane(double x, double y, double nx, double ny) {
    this();
    this.point[0] = x;
    this.point[1] = y;
    this.normal[0] = nx;
    this.normal[1] = ny;
  }
```

```
  public double[] getVelocity() {
    return this.velocity;
  }

  public double[] getAcceleration() {
    return this.acceleration;                                          70
  }

  public double[] getForce() {
    return this.force;
  }

  public double[] getPosition() {
    return this.point;
  }
                                                                       80
  public double getRadius() {
    return 0;
  }

  public double getInverseMass() {
    return 0;
  }

  public double[] getNormal() {
    return this.normal;                                                90
  }

  public double getInverseInertia() {
    return 0;
  }

  public double getTorque() {
    return this.torque;
  }
                                                                       100
  public double[] getLinearMomentum() {
    return this.linearMomentum;
  }

  public double getAngularMomentum() {
    return this.angularMomentum;
  }

  public double getAngularVelocity() {
    return this.angularVelocity;                                       110
  }

  public double getAngularAcceleration() {
    return this.angularAcceleration;
  }

  public double getOrientation() {
    return this.orientation;
  }
                                                                       120
  public void paint(Graphics g) {
  }
}

/**
 * $Log: SeperatingPlane.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */                                                                    130
```

TouchSensor.java

```
/*
 * Touch Sensor Class.
 * $Id: TouchSensor.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.sim;
import java.awt.*;

/**
 * Provides a unidirectional touch sensor much like a bump skirt.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *                                                                             20
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: TouchSensor.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class TouchSensor extends Sensor
{
  /**
   * Public no-argument constructor.
   */
  public TouchSensor() {                                                        30
    this.medium = Sensor.CONTACT_FORCE;
  }

  // the bump skirt is a unidirectional touch sensor
  public double filter(double intensity, double angle) {
    return intensity;
  }
}

/**                                                                             40
 * $Log: TouchSensor.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

Vehicle2D.java

```
/*
 * Vehicle 2D Class.
 * $Id: Vehicle2D.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
```

```
 * Please do not redistribute without permission.                          10
 */

package robotworld.sim;
import robotworld.gui.*;
import java.awt.*;
import java.awt.geom.*;
import java.util.*;

/**
 * Provides a model of two dimensional disk with the appropriate           20
 * inertia.  Drawn as a circle with a radial line indicating the
 * current heading.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: Vehicle2D.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class Vehicle2D extends Particle                                    30
{
  public Vehicle2D() {
    this.radius = 9.0;

    // inertia for a uniform density 2d disk: MR^2 / 2
    this.inverseInertia = 2.0 * this.inverseMass /
                          (this.radius*this.radius);
  }

  public void paint(Graphics g) {                                          40
    double r = this.radius;
    double hx = this.heading[X];
    double hy = this.heading[Y];
    double x = this.position[X];
    double y = this.position[Y];
    double cr = Math.max(this.clickRadius, r);
    this.bounds.setBounds((int)(x-cr),
                          (int)(y-cr),
                          (int)(cr*2),
                          (int)(cr*2));                                    50
    g.setColor(Color.blue);
    g.drawOval(bounds.x, bounds.y, bounds.width, bounds.height);
    g.drawLine((int)x,
               (int)y,
               (int)(x+r*hx),
               (int)(y+r*hy));
  }
}

/**                                                                        60
 * $Log: Vehicle2D.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

ViscousDrag.java

```
/*
 * Viscous Drag Class.
```

```
 * $Id: ViscousDrag.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld.sim;
import robotworld.gui.*;
import java.awt.*;

/**
 * Provides a model of viscous drag that resists the direction of motion.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.                   20
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: ViscousDrag.java,v 1.1.1.1 1999/06/17 02:56:50 craigh Exp $
 */

public class ViscousDrag extends Sprite implements ForceLaw
{
  protected ParticleSystem pSystem;
  protected double[] force;
                                                                            30
  /**
   * The coefficient of drag, allowed to vary between 0 and 1.
   * A value of 0 indicates no drag.  A value of 1 indicates
   * critical damping.
   */
  protected double coefficientOfDrag;

  public ViscousDrag() {
    this.coefficientOfDrag = 0.25;
    this.force = new double[2];                                             40
  }

  public void setParticleSystem(ParticleSystem p) {
    this.pSystem = p;
  }

  /**
   * Sets the coefficient of drag.
   *
   * @param coefficientOfDrag the coefficient of drag                       50
   */
  public void setCoefficientOfDrag(double coefficientOfDrag) {
    this.coefficientOfDrag = coefficientOfDrag;
  }

  /**
   * Gets the coefficient of drag.
   *
   * @return the coefficient of drag
   */                                                                       60
  public double getCoefficientOfDrag() {
    return this.coefficientOfDrag;
  }

  public void apply() {
    int n = this.pSystem.getNumParticles();
    for (int i=0; i<n; i++) {
      Particle p = this.pSystem.getParticleAt(i);
      DoubleVector.multiply(this.force,
                    -this.coefficientOfDrag, p.getLinearMomentum());        70
```

```
        DoubleVector.add(p.getForce(),
                         this.force, p.getForce());
        p.torque -= this.coefficientOfDrag * p.getAngularMomentum();
      }
    }

    public void paint(Graphics g) {
    }
}
```

80

```
/**
 * $Log: ViscousDrag.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:50  craigh
 * Original Version.
 *
 */
```

C.8 The robotworld package

BehaviorCanvas.java

```
/*
 * Behavior Canvas Class.
 * $Id: BehaviorCanvas.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld;
import robotworld.gui.*;
import java.awt.*;

/**
 * Overrides the SpriteCanvas user interface to add connectability support.
 * Acts as a standard SpriteCanvas until the user sets the current bean
 * to a class which is Connectable.  Then it enters Special Mode.              20
 * When the BehaviorCanvas is in Special Mode, normal sprite creation
 * is disabled.  The standard selection interface is overridden and replaced
 * with the following:
 * <ul>
 * <li>Clicking on a source, and then a destination, creates a connectable
 *     between the two.
 * <li>Clicking on a connectable selects (or deselects) it.
 * </ul>
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.                   30
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: BehaviorCanvas.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 */
```

```
public class BehaviorCanvas extends SpriteCanvas {

  protected void selectSprite(Sprite spr) {

    // when in special mode                                                    40
    if (this.specialMode) {

      // if nothing is selected, and the user selects a source,
      // create the connector and add the source to it
      if (this.lastSprite == null && (spr instanceof Source)) {
        this.lastSprite = (Sprite)this.createSprite();
        this.lastSprite.addChild(spr);

        // otherwise, if something has been selected, and then the
      // user selects a destination, add it to the connector and deselect      50
      } else if (this.lastSprite != null && (spr instanceof Destination)){
        this.lastSprite.setSelected(false);
        this.lastSprite.addChild(spr);
        this.lastSprite = null;

      // otherwise, if nothing is selected and the user selects a
      // connectable, allow it to be toggled
      } else if (this.lastSprite == null && (spr instanceof Connectable)) {
        spr.setSelected(!spr.getSelected());
        this.lastSprite = spr;                                               60
      }

    // otherwise act like any other sprite canvas
    } else {
      super.selectSprite(spr);
    }
  }

  protected void createSprite(Point p) {
    // Normal sprite creation is disabled when in special mode.               70
    if (!this.specialMode) {
      super.createSprite(p);
    }
  }

  public void setCurrentBean(Class c) {
    if (c != null) {

      // enter special mode whenever c is a Connectable
      // leave special mode whenever c is not                                80
      this.specialMode = Connectable.class.isAssignableFrom(c);

      // and clear the last selection so the selectSprite logic works
      if (this.lastSprite != null) {
        this.lastSprite.setSelected(false);
        this.lastSprite = null;
      }
    }

    // but make sure to do the usual stuff too                               90
    super.setCurrentBean(c);
  }
}

/**
 * $Log: BehaviorCanvas.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:49  craigh
 * Original Version.
 *
 */                                                                          100
```

BehaviorWindow.java

```
/*
 * Behavior Window Class.
 * $Id: BehaviorWindow.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                                10
 */

package robotworld;
import robotworld.gui.*;

/**
 * Provides a sprite window with behavior-flavored beans.
 * In other words, this class provides functionality for the behavior layer.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.                     20
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: BehaviorWindow.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 */

public class BehaviorWindow extends SpriteWindow
{

  /**
   * Public no-argument constructor.                                             30
   * Creates a behavior-flavored bean bag and editor canvas,
   * and calls <code>init</code>.
   */
  public BehaviorWindow() {
    this.spriteCanvas = new BehaviorCanvas();
    this.init();
  }
}

/**                                                                             40
 * $Log: BehaviorWindow.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:49  craigh
 * Original Version.
 *
 */
```

Client.java

```
/*
 * Client Class.
 * $Id: Client.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
```

```
 * Please do not redistribute without permission.                          10
 */

package robotworld;
import robotworld.gui.*;
import robotworld.ide.*;
import java.awt.*;
import javax.swing.JList;

/**
 * Provides a point of entry for the RobotWorld client.                    20
 * Don't be fooled.  This is named Client only for convenience
 * of the command-line guerrillas, who will type:<br><pre>
 *   java RobotWorld.Client [args]</pre>
 * <p>
 * This is just a window.  The real animacy of the Client,
 * the kind that reads cool stuff from the blackboard,
 * is located in ClientCanvas.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *                                                                         30
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: Client.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 */

public class Client extends SpriteWindow
{
  /**
   * Public no-argument constructor.
   * Creates a client-flavored bean bag and client canvas,
   * and calls <code>init</code>.                                          40
   */
  public Client() {
    this.spriteCanvas = new ClientCanvas();
    this.init();
  }

  /**
   * Main point of entry for RobotWorld client.
   *
   * @param args a list of property files to load                          50
   */
  public static void main(String[] args) {
    // set up RobotWorld properties
    Workspace.loadProperties(args);

    // pop up window
    Client win = new Client();

    // things will appear in the window
    // whenever Server gets around to it                                   60
  }
}

/**
 * $Log: Client.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:49  craigh
 * Original Version.
 *
 */
```

ClientCanvas.java

```
/*
 * Client Canvas Class.
 * $Id: ClientCanvas.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld;
import robotworld.net.*;
import robotworld.ide.*;
import robotworld.gui.*;

/**
 * Provides a sprite canvas that continually reads the root sprite
 * (and all of its children) from the blackboard.  Any client-side
 * changes to the sprite tree are discarded.  Not particularly elegant
 * or efficient.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: ClientCanvas.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 */

public class ClientCanvas extends SpriteCanvas
{
  /**
   * The unit of action.  Does everything a sprite canvas would do,
   * but first reads the fresh root sprite off the blackboard.
   */
  public void act()
  {
    try {
      WorkspaceEntry template = (WorkspaceEntry)Workspace.produce(
        WorkspaceEntry.class);
      template.setServer(WorkspaceEntry.EVERYBODY);
      template.setDomain(WorkspaceEntry.ARCHITECTURE_EDITOR);
      template.setVerb(null); // I dunno, do we care?
      template.setSubject(null);
      template.setClient(null);
      WorkspaceEntry entry = (WorkspaceEntry)Workspace.getBlackboard().read(
        template);
      if (entry != null) {
        this.setRootSprite((Sprite)entry.getSubject());
      }
    } catch (Exception e) {
      this.workspace.launchExceptionDialog(e);
    }
    super.act();
  }
}

/**
 * $Log: ClientCanvas.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:49  craigh
 * Original Version.
 *
 */
```

HabitatWindow.java

```
/*
 * Habitat Window Class.
 * $Id: HabitatWindow.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
 */

package robotworld;
import robotworld.gui.*;

/**
 * Provides a sprite window with habitat-flavored beans.
 * In other words, this class provides functionality for the habitat layer.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.                   20
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: HabitatWindow.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 */

public class HabitatWindow extends SpriteWindow
{
  /**
   * Public no-argument constructor.
   * Creates a habitat-flavored bean bag and sprite canvas,                    30
   * and calls <code>init</code>.
   */
  public HabitatWindow() {
    this.spriteCanvas = new SpriteCanvas();
    this.init();
  }
}

/**
 * $Log: HabitatWindow.java,v $                                               40
 * Revision 1.1.1.1  1999/06/17 02:56:49  craigh
 * Original Version.
 *
 */
```

MorphologyCanvas.java

```
/*
 * Morphology Canvas Class.
 * $Id: MorphologyCanvas.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                              10
```

```
 */

package robotworld;
import robotworld.ctl.*;
import robotworld.gui.*;
import java.awt.*;

/**
 * Overrides the SpriteCanvas user interface to add directability
 * (remote connectability) support. It is very similar to an EditorCanvas,
 * with the following substitutions:
 * <ul>
 * <li>RemoteSource for Source
 * <li>RemoteDestination for Destination
 * <li>Directable for Connectable
 * </ul>
 * <p>
 * Acts as a standard SpriteCanvas until the user sets the current bean
 * to a class which is Directable (remotely connectable).  Then it
 * enters Special Mode.  When the MorphologyCanvas is in Special Mode,
 * normal sprite creation is disabled.  The standard selection interface
 * is overridden and replaced with the following:
 * <ul>
 * <li>Clicking on a remote source, and then a remote destination,
 *     creates a directable between the two.
 * <li>Clicking on a directable selects (or deselects) it.
 * </ul>
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: MorphologyCanvas.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 */

public class MorphologyCanvas extends SpriteCanvas {

  /**
   * Public no-argument constructor.
   */
  public MorphologyCanvas() {
  }

  protected void selectSprite(Sprite spr) {
    // when in special mode
    if (this.specialMode) {

      // if nothing is selected, and the user selects a remote source,
      // create the remote connector and add the remote source to it
      if (this.lastSprite == null && (spr instanceof RemoteSource)) {
        this.lastSprite = (Sprite)this.createSprite();
        this.lastSprite.addChild(spr);

        // otherwise, if something has been selected, and then the
      // user selects a remote destination, add it to the
      // remote connector and deselect
      } else if (this.lastSprite != null &&
                 (spr instanceof RemoteDestination)){
        this.lastSprite.setSelected(false);
        this.lastSprite.addChild(spr);
        this.lastSprite = null;

      // otherwise, if nothing is selected and the user selects a
      // directable, allow it to be toggled
      } else if (this.lastSprite == null && (spr instanceof Directable)) {
        spr.setSelected(!spr.getSelected());
        this.lastSprite = spr;
      }
```

```
    // otherwise act like any other sprite canvas
    } else {
      super.selectSprite(spr);
    }
  }

  protected void createSprite(Point p) {
    // Normal sprite creation is disabled when in special mode.
    if (!this.specialMode) {
      super.createSprite(p);
    }
  }

  public void setCurrentBean(Class c) {
    if (c != null) {

      // enter special mode whenever c is a Directable
      // leave special mode whenever c is not
      this.specialMode = Directable.class.isAssignableFrom(c);

      // and clear the last selection so the selectSprite logic works
      if (this.lastSprite != null) {
        this.lastSprite.setSelected(false);
        this.lastSprite = null;
      }
    }

    // but make sure to do the usual stuff too
    super.setCurrentBean(c);
  }
}

/**
 * $Log: MorphologyCanvas.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:49  craigh
 * Original Version.
 *
 */
```

MorphologyWindow.java

```
/*
 * Morphology Window Class.
 * $Id: MorphologyWindow.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld;
import robotworld.gui.*;

/**
 * Provides a sprite window with morphology-flavored beans.
 * In other words, this class provides functionality for the morphology layer.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
```

```
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: MorphologyWindow.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 */

public class MorphologyWindow extends SpriteWindow
{
  /**
   * Public no-argument constructor.
   * Creates a morphology-flavored bean bag and morphology canvas,
   * and calls <code>init</code>.
   */
  public MorphologyWindow() {
    this.spriteCanvas = new MorphologyCanvas();
    this.init();
  }
}

/**
 * $Log: MorphologyWindow.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:49  craigh
 * Original Version.
 *
 */
```

Server.java

```
/*
 * Server Class.
 * $Id: Server.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.
 */

package robotworld;
import robotworld.gui.*;
import robotworld.ide.*;
import java.awt.*;
import javax.swing.JList;

/**
 * Provides a point of entry for the RobotWorld server.
 * Don't be fooled.  This is named Server only for convenience
 * of the command-line guerrillas, who will type:<br><pre>
 *   java RobotWorld.Server [args]</pre>
 * <p>
 * This is just a window.  The real animacy of the server,
 * the kind that writes cool stuff to the blackboard,
 * is located in ServerCanvas.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: Server.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 */
```

```
public class Server extends SpriteWindow
{
   /**
    * Public no-argument constructor.
    * Creates a server-flavored bean bag and server canvas,
    * and calls <code>init</code>.                                    40
    */
  public Server() {
    this.spriteCanvas = new ServerCanvas();
    this.init();
  }

  /**
   * Main point of entry for RobotWorld server.
   *
   * @param a list of property files to load.                        50
   */
  public static void main(String[] args) {
    // set up RobotWorld properties
    Workspace.loadProperties(args);

    // pop up window
    Server win = new Server();

    // create the grandmother sprite
    win.configureToEdit(new LabelSprite());                         60
  }
}

/**
 * $Log: Server.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:49  craigh
 * Original Version.
 *
 */
```

ServerCanvas.java

```
/*
 * Server Canvas Class.
 * $Id: ServerCanvas.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 *
 * Developed for "Rethinking CS101", a project of Lynn Andrea Stein's AP Group.
 * For more information, see <a href="http://www.ai.mit.edu/projects/cs101/">the
 * CS101 homepage</a> or email <las@ai.mit.edu>.
 *
 * Copyright (C) 1999 Massachusetts Institute of Technology.
 * Please do not redistribute without permission.                     10
 */

package robotworld;
import robotworld.net.*;
import robotworld.ide.*;
import robotworld.gui.*;

/**
 * Provides a sprite canvas that continually writes the root sprite
 * (and all of its children) to the blackboard.  Stale copies are     20
 * removed before each write.  Not particularly elegant or efficient.
 * <p>
 * Copyright (C) 1999 Massachusetts Institute of Technology.
```

```
 *
 * @author  Craig Henderson, craigh@alum.mit.edu
 * @version $Id: ServerCanvas.java,v 1.1.1.1 1999/06/17 02:56:49 craigh Exp $
 */

public class ServerCanvas extends SpriteCanvas
{
  /**
   * The unit of action.  Does everything a sprite canvas would do,
   * but also takes the stale root sprite off the blackboard (if any)
   * and writes the fresh root sprite back.
   */
  public void act()
  {
    super.act();
    if (this.getRootSprite() == null) return;

    // out with the old, in with the new
    try {
      WorkspaceEntry template = (WorkspaceEntry)Workspace.produce(
        WorkspaceEntry.class);
      template.setServer(WorkspaceEntry.EVERYBODY);
      template.setDomain(WorkspaceEntry.ARCHITECTURE_EDITOR);
      template.setVerb(null); // I dunno, do we care?
      template.setSubject(null);
      template.setClient(null);
      WorkspaceEntry toss = (WorkspaceEntry)Workspace.getBlackboard().take(
        template);

      WorkspaceEntry entry = (WorkspaceEntry)Workspace.produce(
        WorkspaceEntry.class);
      entry.setServer(WorkspaceEntry.EVERYBODY);
      entry.setDomain(WorkspaceEntry.ARCHITECTURE_EDITOR);
      entry.setVerb(null); // I dunno, do we care?
      entry.setSubject(this.getRootSprite());
      entry.setClient(null);
      Workspace.getBlackboard().write(entry);
    } catch (Exception e) {
      this.workspace.launchExceptionDialog(e);
    }
  }
}

/**
 * $Log: ServerCanvas.java,v $
 * Revision 1.1.1.1  1999/06/17 02:56:49  craigh
 * Original Version.
 *
 */
```

C.9 Configuration and startup files

makefile

```
all : net ide gui ctl sim
        javac -d ./classes robotworld/*.java
        javac -d ./classes robotworld/net/simple/*.java
```

```
        javac -d ./classes robotworld/net/jini/*.java
        rmic -d ./classes robotworld.net.jini.JiniListener

doc :
        javadoc -author -d ./javadoc @robotworld.packages

net :                                                                    10
        javac -d ./classes robotworld/net/*.java

ide :
        javac -d ./classes robotworld/ide/*.java

gui :
        javac -d ./classes robotworld/gui/*.java

ctl :
        javac -d ./classes robotworld/ctl/*.java                         20

sim :
        javac -d ./classes robotworld/sim/*.java
```

client.bat

```
start rmiregistry
```

server.bat

```
start rmiregistry
start java -jar /jini1_0/lib/tools.jar -dir /jini1_0/lib -trees
start java -jar -Djava.security.policy=/jini1_0/example/lookup/policy.all \
     -Dcom.sun.jini.use.registry=yes \
     -Dcom.sun.jini.rmiregistryport=1099 \
     -Djava.rmi.server.codebase=http://your.server.here:8080/outrigger-dl.jar \
     -Dcom.sun.jini.outrigger.spaceName=JavaSpace \
     /jini1_0/lib/transient-outrigger.jar
```

jdk.properties

```
# Hook into JDK for compiling Java code only.
# Editing will be taken care of by a Java GUI.
# Assumes the JDK are installed, and
# that PATH has been updated to include them.
# Note that {0} is a format specifier for the filename(s).
#
robotworld.ide.IDEManager = robotworld.ide.ExternalIDEManager
```

```
#
# Uncomment the following line if you use Windows and think my editor stinks
# robotworld.ide.ExternalIDEManager.EDIT      = notepad {0}
robotworld.ide.ExternalIDEManager.COMPILE   = javac -cp classes {0}
```

layers.properties

```
#
# MAPPING SPRITES TO LAYERS
# These properties establish the transitions between layers of RobotWorld.
# The mapping:
#    Foo = Bar
# Translates to:
#    When you double-click on sprite Foo, instantiate the layer Bar
#    to edit the Foo.
#
robotworld.gui.Sprite = robotworld.ide.CodeEditorWindow
robotworld.sim.RigidBody = robotworld.MorphologyWindow
robotworld.sim.ParticleSystem = robotworld.HabitatWindow
robotworld.ctl.CompositeController = robotworld.BehaviorWindow
#
# MAPPING LAYERS TO BEAN BAGS
# These properties establish the beans (sprites) available for
# instantiation on a particular layer.
# The mapping:
#   Bar = Foo1, Foo2, ... FooN
# Translates to:
#   Sprites Foo1, Foo2, ... FooN should appear in a bean bag within
#   layer Bar, allowing users to choose which sprite to instantiate next.
#
robotworld.MorphologyWindow = \
        robotworld.sim.PhotoSensor, \
        robotworld.sim.PhotoSensor.Left, \
        robotworld.sim.PhotoSensor.Right, \
        robotworld.sim.TouchSensor, \
        robotworld.sim.Beacon, \
        robotworld.sim.DifferentialDrive, \
        robotworld.ctl.CompositeController, \
        robotworld.ctl.RemoteConnection
robotworld.HabitatWindow = \
        robotworld.sim.Particle, \
        robotworld.sim.Vehicle2D
robotworld.BehaviorWindow = \
        robotworld.ctl.Controller, \
        robotworld.ctl.CompositeController, \
        robotworld.ctl.SignalSource, \
        robotworld.ctl.SignalDestination, \
        robotworld.ctl.Connection, \
        robotworld.ctl.SensorPort, \
        robotworld.ctl.ActuatorPort
robotworld.Server = \
        robotworld.sim.ParticleSystem
robotworld.Client =
robotworld.ide.UniformWorkspace.About = file:about.html
robotworld.ide.UniformWorkspace.Java_Reference = \
        http://java.sun.com/products/jdk/1.2/docs/api/index.html
robotworld.ide.UniformWorkspace.Textbook = http://www-cs101.ai.mit.edu/ipij/index.html
robotworld.ide.UniformWorkspace.Users_Manual = file:manual.html
```

multiuser.properties

```
#
# Multi-user implemenation of RobotWorld.
#
java.security.policy          = /jini1_0/example/lookup/policy.all
java.rmi.server.codebase      = http://your.server.here:8080/outrigger-dl.jar
robotworld.net.Blackboard     = robotworld.net.jini.JiniBlackboard
robotworld.net.WorkspaceEntry = robotworld.net.jini.JiniWorkspaceEntry
robotworld.ctl.RedirectEntry  = robotworld.net.jini.JiniRedirectEntry
robotworld.ctl.SignalEntry    = robotworld.net.jini.JiniSignalEntry
#                                                                              10
# This property must exactly match
# the com.sun.jini.outrigger.spaceName property given in server.bat
#
robotworld.net.jini.JiniBlackboard.NAME = JavaSpace
```

ntemacs.properties

```
# Hook into NT Emacs for editing and compiling Java code.
# Assumes gnuserv / gnuclient and the JDK are installed, and
# that PATH has been updated to include them.
# Note that {0} is a format specifier for the filename(s).
robotworld.ide.IDEManager                 = robotworld.ide.ExternalIDEManager
robotworld.ide.ExternalIDEManager.EDIT    = gnuclientw -F {0}
robotworld.ide.ExternalIDEManager.COMPILE = gnudoit (find-file \\\"{0}\\\") \
                                (compile \\\"javac -cp classes {0}\\\")
#
# Emacs seems to prefer forward slashes exclusively, regardless of platform.   10
robotworld.ide.ExternalIDEManager.FILE_SEPERATOR = /
```

singleuser.properties

```
# Single user implemenation of RobotWorld.
robotworld.net.Blackboard     = robotworld.net.simple.SimpleBlackboard
robotworld.net.WorkspaceEntry = robotworld.net.simple.SimpleWorkspaceEntry
robotworld.ctl.RedirectEntry  = robotworld.net.simple.SimpleRedirectEntry
robotworld.ctl.SignalEntry    = robotworld.net.simple.SimpleSignalEntry
```

.emacs

```
;; Use bash for a shell
(setq shell-file-name "sh")
(setq explicit-shell-file-name shell-file-name)
```

```
(setenv "PID" nil)
(defun my-shell-setup ()
  "For bash (cygwin 18) under Emacs 20"
  (setq comint-scroll-show-maximum-output 'this)
  (setq comint-completion-addsuffix t)
  (setq comint-process-echoes t)
  (setq comint-eol-on-send t)
  (setq w32-quote-process-args ?\")
  (make-variable-buffer-local 'comint-completion-addsuffix))
(setq shell-mode-hook 'my-shell-setup)
(setq process-coding-system-alist (cons '("bash" . raw-text-unix)
                                        process-coding-system-alist))

;; Enable syntax coloring
(cond ((fboundp 'global-font-lock-mode)
       ;; Turn on font-lock in all modes that support it
       (global-font-lock-mode t)
       ;; maximum colors
       (setq font-lock-maximum-decoration t)))

;; Highlight parenthesis matches
(show-paren-mode 1)

;; Make sure Java files are always compiled relative to the codebase
(add-hook 'java-mode-hook
  (function
    (lambda ()
      (setq default-directory command-line-default-directory))))

;; Gnuserv-gnuclient setup
(if (load "gnuserv" 'noerror 'nomessage)
    (gnuserv-start))
```

Bibliography

[1] Ralph H. Abraham and Christopher D. Shaw. *Dynamics—The Geometry of Behavior.* Addison-Wesley/Ariel Press, Reading, MA, 2nd edition, 1992.

[2] Phil E. Agre and David Chapman. "What Are Plans For?" In Pattie Maes, editor, *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, pages 17–34. The MIT Press, Cambridge, MA, 1990.

[3] Philip Edward Agre. *The Dynamic Structure of Everyday Life.* PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1989.

[4] William Ross Ashby. *Design for a Brain; the Origin of Adaptive Behavior.* John Wiley and Sons, New York, NY, 2nd edition, 1960.

[5] Norman I. Badler, Brian A. Barsky, and David Zeltzer, editors. *Making Them Move: Mechanics, Control, and Animation of Articulated Figures.* Morgan Kaufmann, San Mateo, CA, 1991.

[6] Yaneer Bar-Yam. *Dynamics of Complex Systems.* Addison-Wesley, Reading, MA, 1997.

[7] Randall D. Beer. *Intelligence as Adaptive Behavior.* Academic Press, San Diego, CA, 1990.

[8] Randall D. Beer. "A Dynamical Systems Perspective on Agent-Environment Interaction." *Artificial Intelligence Journal*, 72(1/2):173–215, 1995.

[9] Mordechai Ben-Ari. "Constructivism in Computer Science Education." In Daniel Joyce, editor, *Proceedings from the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE Bulletin, pages 257–261, March 1998.

[10] Eric W. Bonabeau and Guy Theraulaz. "Why Do We Need Artificial Life?" In Christopher G. Langton, editor, *Artificial Life: An Overview*, pages 303–325. The MIT Press, Cambridge, MA, 1995.

[11] Valentino Braitenburg. *Vehicles: Experiments in Synthetic Psychology.* The MIT Press/Bradford Books, Cambridge, MA, 1984.

[12] Rodney A. Brooks. "A Robot That Walks: Emergent Behavior from a Carefully Evolved Network." *Neural Computation*, 1(2):253–262, 1989.

[13] Rodney A. Brooks. "The Behavior Language; User's Guide." A. I. Memo 1227, MIT Artificial Intelligence Laboratory, Cambridge, MA, April 1990.

[14] Rodney A. Brooks and Cynthia Breazeal. *Embodied Intelligence.* The MIT Press, Cambridge, MA. In preparation.

[15] Dave Cliff. "Neural Networks for Visual Tracking in an Artificial Fly." In Francisco J. Varela and Paul Borgine, editors, *Toward a Practice of Autonomous Systems*, Proceedings from the First European Conference on Artificial Life (ECAL '91), pages 78–87, Cambridge, MA, 1992. The MIT Press.

[16] Dave Cliff. "Computational Neuroethology: A Provisional Manifesto." In Jean-Arcady Meyer and Stewart W. Wilson, editors, *From Animals to Animats*, Proceedings from the First International Conference on the Simulation Adaptive Behavior, pages 71–80, Cambridge, MA, 1993. The MIT Press.

[17] Dave Cliff, Phil Husbands, and Inman Harvey. "Explorations in Evolutionary Robotics." *Adaptive Behavior*, 2(1):71–108, 1993.

[18] Peter Coad and Edward Yourdon. *Object-Oriented Analysis.* Yourdon Press, 1991.

[19] Peter Coad and Edward Yourdon. *Object-Oriented Design.* Yourdon Press, 1991.

[20] Richard Dawkins. *The Selfish Gene.* Oxford University Press, New York, NY, 1989.

[21] Herbert L. Dersem and James Vanderhyde. "Java Class Visualization for Teaching Object-Oriented Concepts." In Daniel Joyce, editor, *Proceedings from the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education,* SIGCSE Bulletin, pages 53–57, March 1998.

[22] Michael G. Dyer. "Toward Synthesizing Artificial Neural Networks that Exhibit Cooperative Intelligent Behavior: Some Open Issues in Artificial Life." In Christopher G. Langton, editor, *Artificial Life: An Overview,* pages 111–134. The MIT Press, Cambridge, MA, 1995.

[23] David Flanagan. *Java in a Nutshell: A Quick Desktop Reference.* O'Reilly, Sebastopol, CA, 2nd edition, 1997.

[24] James Foley, Andries van Dam, Steven Feiner, and John Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, MA, 2nd edition, 1990.

[25] Stephanie Forrest, editor. *Emergent Computation: Self-Organizing, Collective, and Cooperative Phenomena in Natural and Artificial Computing Networks.* Special Issues of Physica D. The MIT Press, Cambridge, MA, 1st mit press edition, 1991.

[26] Amy Fowler. "A Swing Architecture Overview: The Inside Story on JFC Component Design." See `http://java.sun.com/products/jfc/tsc/archive/what_is_arch/swing-arch/swing-arch.html`.

[27] Amy Fowler. "Mixing Heavy and Light Components." See `http://java.sun.com/products/jfc/tsc/archive/tech_topics_arch/mixing/mixing.html`.

[28] Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language.* Addison-Wesley, Reading, MA, 1997.

[29] Gene F. Franklin, J. David Powell, and Abbas Emami-Naeini. *Feedback Control of Dynamic Systems.* Addison-Wesley, Reading, Massachusetts, 1986.

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[31] David Gelernter. *Mirrorworlds, or the day software puts the universe in a shoebox: how it will happen and what it will mean.* Oxford University Press, New York, NY, 1991.

[32] James L. Gould. *Ethology: the Mechanisms and Evolution of Behavior.* W.W. Norton and Company, New York, NY, 1982.

[33] Mark Green. "Using Dynamics in Computer Animation: Control and Solution Issues." In Norman I. Badler, Brian A. Barsky, and David Zeltzer, editors, *Making Them Move: Mechanics, Control, and Animation of Articulated Figures,* pages 281–314. Morgan Kaufmann, San Mateo, CA, 1991.

[34] Philip Greenspun. *Philip and Alex's Guide to Web Publishing.* Academic Press / Morgan Kauffman, San Mateo, CA, 1999. See also `http://photo.net/wtr/thebook/index.html`.

[35] John Gribbin. *In Search of Schrödinger's Cat: Quantum Physics and Reality.* Bantam Books, 1984.

[36] Hermann Haken. *Principles of Brain Functioning: A Synergetic Approach to Brain Activity, Behavior and Cognition.* Springer-Verlag, New York, NY, 1996.

[37] Lynne Hall and Adrian Gordon. "A Virtual Learning Environment for Entity-Relationship Modeling." In Daniel Joyce, editor, *Proceedings from the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education,* SIGCSE Bulletin, pages 345–349, March 1998.

[38] John Hertz, Anders Krogh, and Richard G. Palmer. *Introduction to the Theory of Neural Computation*, volume 1 of *Studies in the Sciences of Complexity*. Addison-Wesley, Reading, MA, 1991.

[39] David W. Hogg, Fred G. Martin, and Mitchel Resnick. "Braitenburg Creatures." Epistemology and Learning Memo 13, MIT Media Laboratory, Cambridge, MA, 1991.

[40] Joseph Jones and Anita Flynn. *Mobile Robots: Inspiration to Implementation.* A K Peters, Wellesley, MA, 1st edition, 1993. Avoid the 2nd edition—too many typos.

[41] Leslie Pack Kaelbling. *Learning in Embedded Systems.* The MIT Press, Cambridge, MA, 1993.

[42] Leslie Pack Kaelbling and S. J. Rosenschein. "Action and Planning in Embedded Agents." In Pattie Maes, editor, *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, pages 35–48. The MIT Press, Cambridge, MA, 1990.

[43] Stuart A. Kauffman and Sonke Johnsen. "Co-Evolution to the Edge of Chaos: Coupled Fitness Landscapes, Poised States, and Co-Evolutionary Avalanches." In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, number 10 in Studies in the Sciences of Complexity, pages 325–369, Reading, MA, 1991. Santa Fe Institute, Addison-Wesley.

[44] J. A. Scott Kelso. *Dynamic Patterns: The Self-Organization of Brain and Behavior.* The MIT Press, Cambridge, MA, 1995.

[45] Roger King. "My Cat is Object-Oriented." In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 23–30. ACM Press and Addison-Wesley, New York, NY, 1989.

[46] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press, Cambridge, MA, 1992.

[47] Deepak Kumar and Lisa Meeden. "A Robot Laboratory for Teaching Artificial Intelligence." In Daniel Joyce, editor, *Proceedings from the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE Bulletin, pages 341–344, March 1998.

[48] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley, Reading, MA, 1997.

[49] Henry Lieberman. "Object-Oriented Programming in Act 1." In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. ACM Press and Addison-Wesley, New York, NY, 1989.

[50] Pattie Maes. "Modeling Adaptive Autonomous Agents." In Christopher G. Langton, editor, *Artificial Life: An Overview*, pages 135–162. The MIT Press, Cambridge, MA, 1995.

[51] Fred G. Martin. "A Toolkit for Learning: Technology of the MIT LEGO Robot Design Competition." Prepared for the Workshop on Mechatronics Education, Stanford University. See `http://lcs.www.media.mit.edu/people/fredm/papers/stanford/index.html`, 1994.

[52] Fred G. Martin. *Circuits to Control: Learning Engineering by Designing LEGO Robots.* PhD thesis, Massachusetts Institute of Technology, Program in Media Arts and Sciences, 1994.

[53] Fred G. Martin. "The Handy Board Technical Reference." See `http://el.www.media.mit.edu/groups/el/projects/handy-board/index.html`, 1997.

[54] H. R. Maturana and F. J. Varela. *The Tree of Knowledge: The Biological Roots to Understanding.* Shambhala Publications, Boston, MA, 1987.

[55] Scott McCloud. *Understanding Comics: The Invisible Art.* HarperCollins, New York, NY, 1993.

[56] David McFarland and Thomas Bösser. *Intelligent Behavior in Animals and Robots.* The MIT Press, Cambridge, MA, 1993.

[57] Francesco Mondada, Edoardo Franzi, and Paolo Ienne. "Mobile Robot Miniaturization: a Tool for Investigation in Control Algorithms." In *Experimental Robotics III*, Proceedings of the Third International Symposium on Experimental Robotics, Kyoto, Japan, October 28-30, 1993, pages 501–513, London, 1994. Springer Verlag.

[58] Mathew J. Palakal, Frederick W. Myers, and Carla L. Boyd. "An Interactive Learning Environment for Breadth-First Computing Science Curriculum." In Daniel Joyce, editor, *Proceedings from the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE Bulletin, pages 1–5, March 1998.

[59] Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas.* Basic Books, New York, NY, 1980.

[60] Richard E. Pattis, Jim Roberts, and Mark Stehlik. *Karel the Robot: a Gentle Introduction to the Art of Programming.* John Wiley and Sons, New York, NY, 2nd edition, 1995.

[61] Jean Piaget. *Genetic Epistemology.* Columbia University Press, New York, NY, 1970.

[62] Marc H. Raibert. *Legged Robots That Balance.* The MIT Press, Cambridge, MA, 1986.

[63] George Reese. *Database Programming with JDBC and Java.* O'Reilly, Sebastopol, CA, 1997.

[64] M. Resnick, R. Berg, M. Eisenberg, S. Turkle, and F. Martin. "Beyond Black Boxes: Bringing Transparency and Aesthetics Back to Scientific Instruments." Proposal to the National Science Foundation (project

funded 1997–1999). See `http://el.www.media.mit.edu/groups/el/papers/mres/black-box/proposal.html`, 1996.

[65] Mitchel Resnick. *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds.* The MIT Press, Cambridge, MA, 1994.

[66] Craig W. Reynolds. "Flocks, Herds, and Schools: A Distributed Behavioral Model." In *Computer Graphics*, volume 21 of *Annual Conference Series*, pages 25–34. SIGGRAPH '87, 1987.

[67] Craig W. Reynolds. "Competition, Coevolution and the Game of Tag." In Rodney A. Brooks and Pattie Maes, editors, *Artificial Life IV*, Proceedings from the Fourth International Workshop on the Synthesis and Simulation of Living Systems, Cambridge, MA, 1994. The MIT Press.

[68] John Rosenburg and Michael Kölling. "Testing Object-Oriented Programs: Making it Simple." In Daniel Joyce, editor, *Proceedings from the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education*, pages 77–81, March 1997.

[69] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[70] James Randal Sargent. The Programmable LEGO Brick: Ubiquitous Computing for Kids. Master's thesis, Massachusetts Institute of Technology, Program in Media Arts and Sciences, 1995.

[71] Sally Shlaer and Stephen J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data.* Yourdon Press, 1989.

[72] Sally Shlaer and Stephen J. Mellor. *Object Lifecycles: Modeling the World in States.* Yourdon Press, 1991.

[73] Karl Sims. "Evolving 3D Morphology and Behavior by Competition." In Rodney A. Brooks and Pattie Maes, editors, *Artificial Life IV*, Proceedings from

the Fourth International Workshop on the Synthesis and Simulation of Living Systems, pages 28–39, Cambridge, MA, 1994. The MIT Press.

[74] Karl Sims. "Evolving Virtual Creatures." In *Computer Graphics*, Annual Conference Series, pages 15–22. SIGGRAPH '94, July 1994.

[75] Luc Steels. "The Artificial Life Roots of Artificial Intelligence." In Christopher G. Langton, editor, *Artificial Life: An Overview*, pages 75–110. The MIT Press, Cambridge, MA, 1995.

[76] Lynn Andrea Stein. *Interactive Programming in Java.* Morgan Kaufmann, San Mateo, CA. In preparation. See also `http://www-cs101.ai.mit.edu/ipij/index.html`.

[77] Lynn Andrea Stein. "Interactive Programming: Revolutionizing Introductory Computer Science." *Computing Surveys*, 28A(4), 1996.

[78] Lynn Andrea Stein. "Challenging the Computational Metaphor: Implications for How We Think." *Cybernetics and Systems*, 30(6), September 1999.

[79] Steven H. Strogatz. *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry, and Engineering.* Addison-Wesley, Reading, MA, 1994.

[80] Sun Microsystems. *JavaSpaces Specification*, 1999. Available online as PostScript or PDF at `http://www.sun.com/jini/specs/index.html`.

[81] Charles Taylor and David Jefferson. "Artificial Life as a Tool for Biological Inquiry." In Christopher G. Langton, editor, *Artificial Life: An Overview*, pages 1–13. The MIT Press, Cambridge, MA, 1995.

[82] D. Terzopoulos, X. Tu, and R. Greszczuk. "Artificial Fishes with Autonomous Locomotion, Perception, Behavior, and Learning in a Simulated Physical World." In Rodney A. Brooks and Pattie Maes, editors, *Artificial Life IV*, Proceedings

from the Fourth International Workshop on the Synthesis and Simulation of Living Systems, pages 17–27, Cambridge, MA, 1994. The MIT Press.

[83] Chris Tomlinson and Mark Scheevel. "Concurrent Object-Oriented Programming Languages." In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 79–124. ACM Press and Addison-Wesley, New York, NY, 1989.

[84] Michael David Travers. *Programming with Agents: New Metaphors for Thinking About Computation.* PhD thesis, Massachusetts Institute of Technology, Program in Media Arts and Sciences, 1996.

[85] Sherry Turkle. "Seeing Through Computers: Education in a Culture of Simulation." *The American Prospect*, (31):76–82, March–April 1997.

[86] Sherry Turkle and Seymour Papert. "Epistemological Pluralism: Styles and Voices within the Computer Culture." *Signs: Journal of Women in Culture and Society*, 16(1):128–157, 1990.

[87] Timothy van Gelder. "What Might Cognition Be, If Not Computation?" *Journal of Philosophy*, 92(7):345–381, July 1995.

[88] Norbert Wiener. *Cybernetics: or, Control and Communication in the Animal and the Machine.* The MIT Press, Cambridge, MA, 2nd edition, 1965.

[89] Andrew Witkin, David Baraff, and Michael Kass. "An Introduction to Physically Based Modelling." Lecture notes from the SIGGRAPH '95 course. See `http://www.cs.cmu.edu/afs/cs.cmu.edu/user/baraff/www/pbm/pbm.html`, 1997.