The Art of Agent-Oriented Modeling

Leon Sterling and Kuldar Taveter

The MIT Press
Cambridge, Massachusetts
London, England

For information about special quantity discounts, please email special_sales@mitpress.mit.edu.

This book was set in Times New Roman on 3B2 by Asco Typesetters, Hong Kong.
Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Sterling, Leon.
The art of agent-oriented modeling / Leon S. Sterling and Kuldar Taveter.
p. cm. — (Intelligent robotics and autonomous agents series)
Includes bibliographical references and index.
ISBN 978-0-262-01311-6 (hardcover : alk. paper) 1. Intelligent agents (Computer software) 2. Computer software—Development. I. Taveter, Kuldar. II. Title.
QA76.76.I58S757 2009
006.3—dc22 2008044231

10 9 8 7 6 5 4 3 2 1

1 Introduction

We live today in a complicated world. Complexity comes in many guises, and ranges from small-scale to large-scale concerns. On the small scale, we interact with an ever-increasing array of devices, most of them new and incompletely understood, such as mobile phones and portable digital music players. On the large scale we live in complex institutions—governments, corporations, educational institutions, and religious groups. No one can understand all that goes on in such institutions.

A sensible response to dealing effectively with the complexity is to seek help. Help can come from other people. We may hire a consultant to help us deal with government. We may get a friend to help us install a home wireless network, or we may use software tools to automate tasks like updating clocks and software. It might even be sensible to combine both people and software. An underlying purpose of this book is to help us conceptualize a complicated environment, where many parts—both social and technical—interact. The key concepts we use are agents and systems.

The underlying question in this book is how to design systems that work effectively in the modern environment, where computing is pervasive, and where people interact with technology existing in a variety of networks and under a range of policies and constraints imposed by the institutions and social structures that we live in. We use the word "system" in the broadest sense. Systems encompass a combination of people and computers, hardware, and software. There are a range of devices, from phones to MP3 players, digital cameras, cars, and information booths.

We are particularly interested in systems that contain a significant software component that may be largely invisible. Why the interest? Such systems have been hard to build, and a lot of expensive mistakes have been made. We believe that better conceptualization of systems will lead to better software.

In this first chapter, we discuss our starting philosophy. There are particular challenges within the modern networked, computing environment, such as its changeability and consequent uncertainty. We discuss challenges of the computing environment in the first section. In the second section we address agents, and why we think they are a natural way to tame complexity. In the third section, we discuss multiagent

systems. The fourth section addresses modeling. In the fifth section, we discuss systems engineering, which we believe is a good background for conceptualizing the building of systems. Using multiagent systems to better understand systems with a significant software component is really the raison d'être of the book. The sixth section briefly describes a complementary view of multiagent systems that proceeds "bottom-up," where desired behavior emerges from the interaction of components rather than being designed in "top-down." The final section in the first chapter frames our discussion in the context of the history of programming languages and paradigms over the last fifty years.

An aspect of our philosophy is a strongly pragmatic streak. This book is intended to encourage people to model systems from an agent-oriented perspective. From our teaching experience, we know that describing abstract concepts is insufficient. Clear examples of concept use are extremely helpful, and methods are needed to use the concepts. We strongly believe that developers and students can learn to model by looking at good examples and adapting them. The development of good examples from our teaching and research experience was why we felt ready to write this book. We decided to write this book only when convinced that people could build practical things, and our modeling approach would help people envisage multiagent systems operating in complex environments.

1.1 Building Software in a Complex, Changing World

The task of building software has never been more challenging. There is unprecedented consumer demand and short product cycles. Change in the form of new technology is happening at an increasing rate. Software needs to be integrated with existing systems and institutions as seamlessly as possible, and often in a global network where local cultural factors may not be understood.

In this section, we identify several key characteristics of the modern computing environment for which software must be written. These characteristics suggest five attributes that software should have in order to be effective within the environment. These desirable software attributes are motivating factors for the multiagent perspective being advocated in this book.

Complexity is the first characteristic we highlight. Essentially, the modern world is complicated, and that complexity affects software. As an example, consider a billing system for mobile phone usage or consumption of utilities such as electricity or gas. At first thought, billing may seem reasonably straightforward. All you need is a monitoring mechanism, such as a timer for phone calls or a meter for electricity, and a table of rates. Then it will be a simple calculation to determine the bill. However, billing is not simple in practice. There have been expensive software failures in building billing systems. New taxes could be introduced that change the way billing must

be done, and complications when taxes should apply. The government may allow rebates for certain classes of citizens, such as the elderly, with complicated rules for eligibility. There may be restrictions on how such rebates are to be reported. Phone calls that cross calendar days or have differential rates cause complications. International calls have a myriad of other factors. The phone company may decide on special deals and promotions. In other words, the potentially simple billing system is complex, as it is complicated by a range of social, commercial, and technical issues—a common occurrence.

Another characteristic of many modern systems is that they are *distributed*, both computationally and geographically. Web applications are the norm, and they engender a whole range of issues. For example, if the Web application has an international customer base, does it make sense to have mirror sites for storing and downloading information? If the web traffic is high volume, does there need to be load balancing? Are multilingual versions of the interface necessary, or at least a change of terms in different places where the software is delivered? Such considerations change the nature of an application.

Most software applications are *time-sensitive*. Time is an issue both in response to consumer demand and for consumption of resources. For the former, we expect instantaneous responses to our queries. Indeed, too slow a response can cause a product to fail. For the latter, if too many computational resources are necessary to process information, an application may be infeasible. Architectures and designs need to be analyzed for speed and other qualities.

The surrounding environment is *uncertain* and *unpredictable*. Not all of the information that we receive is reliable. Some of the information is caused by genuine uncertainty, like weather predictions or future prices of stock. Some information is fraudulent, such as emails that are part of phishing attacks or scams. Although unpredictability may make life interesting in some circumstances, it is a challenge for software developers. There is no guarantee that the environment can be controlled, which has been the prevailing style for standard software methods. Software has to be developed to expect the unexpected.

The final characteristic of the modern environment that we highlight is that it is *open*. There is new information and new realities. New software viruses are written that must be protected against. There are new policies promulgated by institutions and new legislation developed by governments. And it is not just new information that affects the environment. The external environment is *changing*. Bank interest rates change; mobile phone plans change. To be effective in the environment, behavior must change accordingly.

Having identified these challenging characteristics of the modern environment, let us suggest desirable attributes for software if it is to perform effectively and serve us well.

The first attribute desirable for software is *adaptivity*. As the world changes, we would like our software to reflect the change. As new facts enter the world, the software should not break. Brittleness was a problem with expert systems and has limited their applicability. An obvious area where adaptivity is essential is security. As a new virus or security threat is determined, it would be good if the virus checking/firewall/security system incorporated the information automatically. Indeed, software is beginning to run automatic security updates, and this trend needs to continue.

The second attribute is *intelligence*. Consumer ads already sometimes claim that their product is more intelligent than its competitors—for example, an air conditioning system or a mobile phone. Presumably they mean more features. We would hope that increased intelligence would lead to better integration. We certainly appreciate when a computer clock automatically adjusts for Daylight Saving Time, and when memory sticks and digital cameras work seamlessly across a range of computers and operating systems. Intelligence is a way of dealing with complexity and uncertainty, and being able to determine when knowledge may be false. One dimension of intelligence that we would expect is awareness. A system that was unaware of what was going on around it would not seem intelligent.

A third desirable attribute is *efficiency*. There is a need and expectation for instantaneous responses, which will be achieved only by efficient implementations in light of the complexity. Efficiency may well determine the possibility of solutions—for example, file sharing or downloading large video files.

A more abstract fourth attribute is *purposefulness*. In light of the complexity and changing nature of the environment, it will be difficult—if not impossible—for all requirements to be stated. It is better to work at a higher level and to explain purposes in terms of goals, and, in certain circumstances, to have the system determine the appropriate path of action. This approach can aid system design and clarity, which leads to the next attribute.

The final attribute is a little bit different and perhaps less obvious; namely, the software should be *understandable*, at least in its design and overall purpose. We need ways of thinking about and describing software that simplify complexity, at least with regard to describing behavior. The desire for understandability is influenced by the software engineering perspective undertaken within this book. There are many potential advantages of understandable software, including better instructions for how to use it.

These are a lot of demands, and we need to address them. Let us explore one way of thinking about them. Indeed, the rationale for developing the agent-oriented modeling techniques that form the essence of this book is that they better address the characteristics of the world around us and can meet the desirable software objectives.

1.2 What Is an Agent?

This book advocates adoption of the concept of agents in thinking about software in today's world. Agents are suitable for the current software development challenges outlined in the previous section. In our opinion, the applicability of agents is likely to increase over the coming years.

An agent has existed as a concept for thousands of years. The *Oxford American Dictionary* gives two meanings for the word "agent," both of which are relevant. Perhaps the more fundamental definition of the two is "a person or thing that takes an active role or produces a specified effect." The connotation is that agents are active entities that exist in the world and cause it to change. The phrase "agent of change" springs to mind, and indeed was mentioned in the dictionary entry. The concepts of roles and effects mentioned in the definition are key. They will be discussed in the next chapter and throughout the book.

The more common sense meaning is the other definition: "a person who acts on behalf of another, for example, managing business, financial, or contractual matters, or provides a service."

In human communities and societies, an agent is a person who carries out a task on behalf of someone else. For example, a travel agent can make enquiries and bookings for your holiday; a literary agent interacts with publishers to try and find a book publisher; and a real estate agent helps you buy, sell, or rent a house or factory. In a well-known biblical story from the Old Testament (Gen. 24:1–67), Abraham sends his servant Eliezer to act as a marriage agent to find a bride for his son Isaac.

Computer science researchers have used the word "agent" for more than twenty-five years with a range of different meanings. For the purpose of this chapter, we define an agent as "an entity that performs a specific activity in an environment of which it is aware and that can respond to changes." Depending on their background, readers are likely to bring initially differing emphases and understanding of the word "agent" to the book. We anticipate that the reader will gain an increased understanding of the word "agent" through engagement with this book, its associated exercises, and attempts to construct agent-oriented models.

One obvious consequence of our informal definition that is worth explicitly pointing out is that people are agents. People live in the world, are aware of changes in the world in many different factors and attributes, including weather, politics, and social organizations. People act in the world. For example, they might protect themselves against the weather by carrying an umbrella or putting on sunscreen or a snow suit—usually not all three simultaneously. They might vote in an election to influence politics; they might form networks of friends.

Let us look at some agents from the world of computing over the past decade. Some readers may be surprised at what we consider to be an agent. However, whether you agree or disagree with the classification should not affect your appreciation of the book.

The first successful robot in the consumer market has been Roomba. Its Web site proclaims that Roomba is "the world's top-selling home robot, with over two million sold." Roomba is an automated vacuum cleaner, designed to clean rooms. It senses the shape of your room and determines a cleaning pattern for traversing the room. It is flat and capable of vacuuming under beds and couches. Most models have a charging base to which they return when they are running out of power. Why we describe it as an agent is that it senses its environment, a house, and its own state, and performs a task—namely cleaning a floor. It responds to changes in the environment, such as moving furniture, people, and its own power level. In 2006, the company introduced the Scooba, which washes floors rather than vacuuming, but is otherwise similar. A video clip distributed on the Internet shows it sucking up Diet Coke and "eating" pretzel crumbs.

The next example we consider is a Tamagotchi, a toy developed in Japan and popularized in the late 1990s. In one sense, a Tamagotchi is just a simple interactive simulation. It has "needs": food, bathroom, sleep; its owner, usually a young child, must provide for those needs. If the needs are not met, the Tamagotchi "gets sick" and can even "die." Later models interact with other Tamagotchis, and recently, Tamagotchi Towns have been created on the Internet. We model Tamagotchis in detail in chapter 3.

The most popular robot in Japan is AIBO, a robot produced by Sony Entertainment. Sony marketed AIBO as an electronic pet. AIBO comes with preprogrammed behaviors, including walking, wagging its tail, flashing its eyes, and electronic barking (or making some sound). The behaviors are sophisticated, including a great ability to right itself if it falls over on an uneven surface, for example. Several of the behaviors are affected by its interaction with people, such as being patted on the head: hence the marketing as a pet.

As well as preprogrammed behaviors, Sony provided a programming environment for AIBO. It promoted the use of the programming environment through the addition of a special league to RoboCup, a robot soccer-playing competition held annually since 1997. The Sony dog league, started in 2000, has teams of three AIBO dogs playing soccer on a field larger than two table-tennis tables. The dogs have improved their play over time. While playing a game of soccer, an AIBO dog is certainly an agent. It must sense where the ball is, move to the ball, propel the ball forward toward the goal, and play out its assigned team role—for example, attacker or defender. A photo of an AIBO dog is shown in figure 1.1.

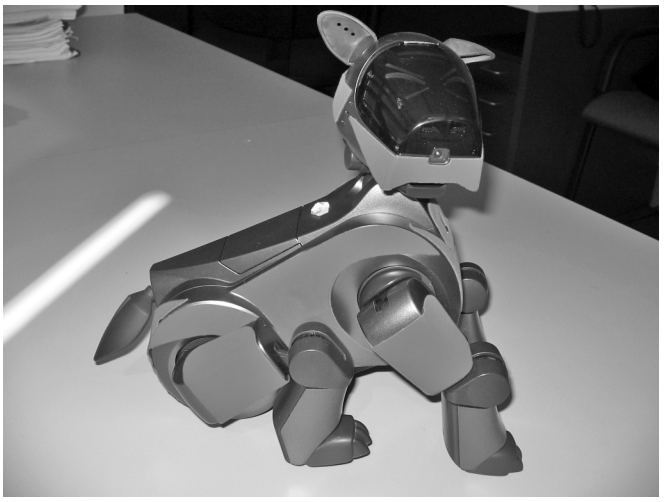

Figure 1.1
The AIBO by Sony

These three examples are tangible. The agents can be identified with a physical device, a vacuum cleaner, a toy, or a dog. Let us be a bit more abstract, as we consider four software examples.

In addition to the so-called gadget-based digital pets mentioned previously, such as Tamagotchis, there are other types of digital pets. They can be Web site–based, such as digital pets that can be obtained and played with in the Neopets, Webkinz, and Petz4fun Web sites. There are also game-based digital pets running on video game consoles, such as Nintendogs by Nintendo and HorseZ by Ubisoft, both for Nintendo DS game consoles.

The second example is a virus of the software kind. Essentially a computer virus is a computer program written to alter the way a computer operates, without the permission or knowledge of the user. We believe a virus should be able to self-replicate and be able to execute itself. Consequently, we regard a computer virus to be an agent because it needs to sense the status of the networked environment that it is in, and to act by affecting files and computers. Viruses can certainly be effective. The MyDoom virus is reported to have infected more than two hundred thousand computers in a single day. Viruses can be regarded as a malevolent agent, as opposed to a benevolent agent.

The next example is one always discussed in the graduate agent class at the University of Melbourne. Consider a program for handling email, such as Microsoft Outlook, Eudora, or the native Mac OS X mail program, Mail. Is such a program

an agent? In our opinion, it is. The mail handling program needs to be aware of the network environment, and whether particular hosts are receiving email. If hosts are not receiving mail, messages are queued up and sent later without further human intervention. The email program performs actions such as filtering spam. Further, the email program is an agent in the sense of acting as a person's representative. It has detailed knowledge of network protocols that both Leon and Kuldar would need to research. It knows what networks you are connected with—for example, Ethernet or a wireless network—and chooses which one to use appropriately. Students, however, are initially reluctant to regard Outlook or the other programs as agents, instead regarding them as merely programs.

To consider this issue further, how about a RIM BlackBerry? Is the physical device the agent, or is the software running the device the agent? In our opinion, there is no precise answer. What you are modeling determines the response.

Another standard example used in the agent class at the University of Melbourne is considering whether a Web crawler is an agent. Here, students usually regard the Web crawler as an agent. Certainly a Web crawler takes action in the environment, but whether it is aware of its environment is less clear. There are many more examples that could be explored, but we will encourage further discussion via the exercises at the end of the chapter.

To conclude this section, we note that the agent is not a precisely defined entity. There is an associated metaphor of an agent as a representative that suggests several qualities. We highlight three qualities now. One quality is being purposeful in both senses of agents mentioned earlier. Another important quality of an agent is controlled autonomy, or the ability to pursue its own goals seemingly independently. The third quality is the agent needs to be situated—that is, aware of the environment around it. It must be capable of perceiving changes and responding appropriately. All of the examples that we have discussed are situated in an environment they must respond to, and possess the qualities of purposefulness and autonomy, at least to some extent. We consider this topic in more detail in chapter 2.

1.3 From Individual Agents to Multiagent Systems

Individual agents can be interesting. Interactions between agents can also be interesting, as can interactions between an agent and the environment in which it is situated. Adopting the agent metaphor for developing software raises both the visibility and abstraction level of interactions between agents. To appreciate the value in being able to understand agents and their interactions, we need to consider a broader systems view.

Loosely, a *system* is a set of entities or components connected together to make a complex entity or perform a complex function. If several—or perhaps all—of the

connected entities are agents, we have a *multiagent system*. This book is about modeling systems as multiagent systems.

The term *sociotechnical system* is sometimes used to refer to systems that contain both a social aspect, which may be a subsystem, and a technical aspect. Although the term sociotechnical system has some attraction, we prefer the term multiagent system for two reasons. First, it emphasizes our interest in agent-oriented modeling. Second, it avoids any existing associations with different meanings of sociotechnical system. In using the term "multiagent system," we typically convey the sense that the whole is greater than the sum of the parts; also, that the agents interact, and that there will be a focus on interactions.

A note on words: a strength (and weakness) of this book is its attempt to bridge the gap between the everyday world and a strictly formal world. It does so by appropriating terms from our language, such as agent, model, goal, and role, and imbuing them with a technical meaning. Giving precise technical definitions is a mathematical skill, which can be learned but does not come easily, and many people struggle with learning this skill. We aim for accuracy in our use of words, but we do so unobtrusively. We tend not to define terms formally, but will describe specific models carefully. How to progress from general, abstract requirements to precise computer systems is a challenge that will be partly addressed in chapter 7 on methodologies and demonstrated in chapters 8, 9, and 10 on applications.

What is an example of an interaction between agents? A prototypical example from the domestic environment, among others, is an agent encountering and greeting another agent. Microwaves and DVD players often display a welcome message when they are turned on. The message is only for the purpose of interaction. We model greeting in chapters 4 and 9 in the context of a smart home.

Several agents interacting produce interesting behavior that is not predictable. A popular game in the first years of the twenty-first century is "The Sims," which was originally released in 2000 and which comes in several variants. People play the game by controlling activities, appearances, and other attributes of a set of computer-animated characters. The game engine executes the activities, effectively allowing a (simple) simulation of a family. A key feature of the interest in the game is in seeing how the characters interact.

A wonderful example of the power of defining simple characters and activities and letting the interactions produce interesting behavior comes from the movie trilogy *Lord of the Rings*. The movie producers had the challenge of creating realistic large-scale battle scenes between the coalition of the heroes and their followers and the coalition of their enemies. Clearly, participants in a battle are agents, having to observe the actions of others attacking them and having to effect a mortal blow. The key actors of the movie are agents in the battle and need to be filmed. However, it would be expensive getting thousands of extras to enact a battle. Instead,

the key human protagonists were superimposed on a computer animation of a battle. The animation of the battle was generated by software agents run under simulation. Each software agent had simple motions defined. It was the interaction between them that made the scene interesting. Very realistic battle scenes were produced.

Simulations are a good source of examples of multiagent systems. The military in Australia and the United States, among other countries, have successfully developed multiagent simulations. We mention here two examples that illustrate possibilities for agent applications.

The Smart Whole Air Mission Model (SWARMM) was developed as a collaborative project between the Air Operations Division of DSTO and the Australian Artificial Intelligence Institute in the mid-1990s. It integrated a physics simulation of flying aircraft with pilot tactics and reasoning. SWARMM was written in the multiagent reasoning system dMARS. The types of tasks modeled in SWARMM were air defense, attack, escort, and sweep (the use of a fighter to clear a path for an incoming fighter plane or planes). SWARMM allowed the pilots' reasoning to be traced graphically during execution of the simulation. The simulation was developed in cooperation with F-18 pilots who liked the rapid feedback and high-level understandability of the simulation. The simulation had tens of agents who formed small teams, and several hundred tactics. The project was very successful. Taking the agent perspective helped develop an understandable system from which the pilots received visual feedback on their tactics, and the interaction between tactics could be seen in the software and understood.

In the late 1990s, the U.S. military conducted Synthetic Theater of War (STOW 97), an Advanced Concept Technology Demonstration jointly sponsored by the Defense Advanced Research Projects Agency (DARPA) and the United States Atlantic Command. STOW 97 was a training activity consisting of a continuous forty-eight hour synthetic exercise. Technically, it was a large, distributed simulation at the level of individual vehicles. Five U.S. sites participated (one each for the Army, Navy, Marines, Air, and Opposing Forces), plus one in the United Kingdom (UK). All told, there were approximately five hundred computers networked together across these sites, generating on the order of five thousand synthetic entities (tanks, airplanes, helicopters, individual soldiers, ships, missile batteries, buses, and so on). Agent-based software controlled helicopters and airplanes in part of the exercise. Eight company-level missions were run, ranging in size from five to sixteen helicopters (plus automated commanders for Army missions), each of which succeeded in performing its principal task of destroying enemy targets.

We conclude this section by considering multiagent systems in two domains: an intelligent home and e-commerce. Both of them are naturally modeled as multiagent systems, and demonstrate both social and technical characteristics. Social character-

istics apply to how humans interact with technology and how humans interact with each other by the mediation of the technology, and technical characteristics obviously portray the technology to be used.

Let us consider a smart home where appliances interoperate seamlessly for the benefit of the home occupants, a promising area of application for intelligent agents. According to Wikipedia, the intelligent home "is a technological achievement aimed at connecting modern communication technologies and making them available for everyday household tasks. With intelligent home systems, it becomes possible to call home from any telephone or desktop computer in the world to control home appliances and security systems. Intelligent home systems guide the user to perform any operation, to control lighting, heating, air conditioning, or to arm or disarm the security system, and to record or to listen to messages. Other themes envisioned in intelligent home systems are automation, connectivity, wireless networking, entertainment, energy and water conservation, and information access."

An intelligent home system is easily envisaged with separate agents controlling the separate subsystems such as heating, lighting, air conditioning, security, and entertainment, and the agents interacting to facilitate the comfort and convenience of the home owner. Here are some small examples. When the phone rings, the entertainment agent could be aware and turn down the volume of any loud music in the vicinity. An alarm clock set to wake up the home owner for an early morning flight could reset the alarm after it contacted the airport and discovered that the flight was delayed. A security system could track any person in the house. If the person in the house was not recognized by the system, an intruder alert could be initiated, whereby the home owner and the police were contacted, provided with a photo of the intruder, and any visitors or tradespeople scheduled to visit the house were warned to stay away. Some of these example scenarios will be elaborated in chapter 9.

The intelligent home needs knowledge about the technical devices, including their communication capabilities, parameters that can be set, and functions that they can achieve. The intelligent home also needs to be aware of legal and social restrictions. Examples are not playing music too loudly late at night, and not running automatic watering systems in gardens during times of severe water restrictions. The home also needs considerable general knowledge.

We turn to e-commerce. According to Wikipedia (October 11, 2006), e-commerce "consists primarily of the distributing, buying, selling, marketing, and servicing of products or services over electronic systems such as the Internet.... It can involve electronic funds transfer, supply chain management, e-marketing, online marketing, online transaction processing, electronic data interchange (EDI), automated inventory management systems, and automated data collection systems.... It typically uses electronic communications technology such as the Internet, extranets, email, e-books, databases, catalogues, and mobile phones."

E-commerce in its broadest sense is already a big business. Buyers and sellers, both institutions and individuals, can and should clearly be modeled as agents. They form organizations such as company hierarchies, virtual enterprises, and markets. Interaction protocols such as auctions are relevant for modeling and understanding the behaviors of such organizations and their constituent individual agents. A key activity to be understood is negotiation, which has been thoroughly studied in the agent community. E-commerce also implies agents' detailed knowledge about their environment, which consists of environment objects, such as Enterprise Resource Planning (ERP) and Enterprise Application Integration (EAI) systems, servers, Web services, and databases. In business processes are also relevant cultural values, products, and their pricing. All these examples show the complexity of multiagent systems. Business-to-business e-commerce is illustrated in chapter 8.

1.4 What Is Modeling?

The underlying motivation of this book is to help people write software that can work effectively in the modern software context, such as a sophisticated smart home or global e-commerce. To deal with the complexities in such environments, we need to model the systems, highlighting which features are important for the software and how they will be enacted and which features can be ignored.

This section addresses modeling, or the construction and description of models. Modeling is empowering in a practical sense. If you can model, you are a significant part of the way to building something useful.

Let us consider the question: "What is a model?" A definition taken from the Web is that a model is a "hypothetical description of a complex entity or process." A model is constructed to aid in building the system that we have in mind. To paraphrase Parnas's well-known characterization of specifications, a model should be as complex as it needs to be to reflect the issues the system is being built to address, but no more complex.

What are some examples of models? A common school project for primary school children is to build a model of the solar system. In such a model, there is at least some depiction of individual planets, and the sun. More detailed models may include moons of planets and asteroids. More advanced students may try and get some idea of distance of planets from the sun, by either placing the planets in an order, or with some scaled representation of distance. Yet more ambitious students may add a dynamic element to the model by having the planets move around their orbit. Building a good model of the solar system clearly stretches the abilities of primary school children—and usually their parents.

Many years ago, Leon had the experience of visiting a steel plant in northern Ohio to pitch a project to build an expert system that could troubleshoot flaws while mak-

ing steel. He was accompanied by a professor of mechanical engineering, who was offering to build a model of the steel plant. The model would be built at a consistent scale, and would involve pouring liquid from a model of the steel furnace and transporting the molten steel in carts on a track to the location where it would be shaped into sheets. Key discussion points were the layout of the plant, viscosity of the model liquid, and what aspects of the model would be useful for the engineers ultimately designing and building the plant, so that they could be confident the plant would work correctly from the moment it started operating.

Kuldar had the experience of visiting the Melbourne Museum and seeing a display of gold mining. The model of the gold mine at the museum was a useful way of visualizing how the gold mine would have operated in its heyday. Without the model, it would have been difficult to understand.

These examples are of tangible, concrete models. The world of software is more abstract, and accordingly, more abstract models are needed to help envisage and build software systems. We note that software professionals or computing students, the likely readers of this book, possibly spend less time thinking about models than other engineering disciplines or construction areas. Perhaps they don't think in terms of models at all.

One field in which modeling has been used is the development of object-oriented systems. It is usual to build a UML description, which is in fact a model. UML is an acronym for Unified Modeling Language, which reminds us of its modeling nature. UML models can be checked and reviewed to see whether the system is understood and correct before code is generated and the system implemented. Modeling notations for systems built in a procedural style have also been developed, but are perhaps not as widely used.

Models abstract information. For object-oriented programming, interfaces between classes are given, and the actual data passing mechanisms are finalized when the code is implemented. The model limits what we focus on at various stages of the software development life cycle.

To summarize this section, we advocate building appropriate models in order to understand how to design and implement a complex system. It is essential to have intuitively understandable models. The models must have sufficient detail to be useful, but not so much detail as to overwhelm.

1.5 Systems Engineering

How does one build a multiagent system? The research community has diverse opinions on what to emphasize. The conference title of an early set of workshops devoted to all things "agenty," ATAL, reflects that diversity of opinion. ATAL is an acronym for Agent Theory, Architecture, and Languages. The theoreticians claim that

once the theory is established, the practice will be straightforward to implement, and so emphasis should be on theory. The architects claim that if you have the right architecture, all the rest will follow. The language developers claim that given the right programming language, it is straightforward for agent developers to build multiagent systems.

This book makes a different claim. A multiagent system is a system with a significant software component. We must build on what has been learned about developing software over the last forty years. The perspective that needs to be taken for building multiagent systems is a software engineering perspective, which we loosely identify with a systems engineering perspective. So we choose not to focus on theory, architecture, or language, though of course we don't ignore them. Indeed, chapters 5 and 7 discuss some of the architectures, languages, and tools that have emerged from agent research and development.

In this section, we give some motivation for software and systems engineering, a field often slighted and misunderstood by computer scientists and AI researchers. We mention the software engineering life cycle, which motivates the models that we discuss in chapter 3 and illustrate in our examples. We also relate the software engineering life cycle to the systems engineering life cycle by using the analogy of constructing a building. Chapter 4 discusses quality, an important issue to consider when taking a software engineering perspective.

To gain a perspective of software engineering, we offer the following analogy. Consider the task of building a small shed for storage in the backyard of a house, a common hobby for men, especially in previous decades. Many men and women could be successful with this task, particularly if they have a practical bent. However, just because someone built such a storage shed would not immediately qualify him or her to build a thirty-floor office building. There is extra knowledge needed about building materials, structures, and regulations—to mention just a few issues. Now consider the task of writing a computer program to process data. Many men and women could be successful with this task, particularly if they have a technical bent. However you wouldn't automatically trust that person to program an air traffic control system. The missing discipline and knowledge is loosely covered in the area of software engineering.

A definition of software engineering developed for Engineers Australia is "a discipline applied by teams to produce high-quality, large-scale, cost-effective software that satisfies the users' needs and can be maintained over time."

Significant words and phrases in the definition include *discipline*, which implies an underlying body of knowledge; *users*, which implies the need for requirements; *teams*, which implies the need for communications and interfaces; *over time*, which implies that the system should be able to be changed without becoming brittle; *high-quality*, which suggests performance criteria, not only functional capabilities; and *large-scale*,

which means different architectural consideration about performance and other qualities. Understanding costs and trade-offs in design will be important. Also important will be recognizing the needs of stakeholders, not only users.

Although all aspects of software engineering are not explicitly addressed, we have been influenced by taking a software engineering view. Models have been proposed that we believe can be understood by a variety of stakeholders at varying levels of abstraction. We take a systems view because multiagent system designers and developers should have a broad awareness of how the software they are designing and building interacts with other hardware, software, and agents more generally.

We presume that the multiagent system will follow a systems development life cycle. There will be a stage of gathering requirements. Once the requirements have been elicited, they are analyzed. The analysis goes hand in hand with design, where trade-offs are expected to be needed to allow the building of a system that meets users' requirements, both functional and nonfunctional. The system must be implemented, tested, and maintained. Explicit languages, methodologies, and tools for the latter stages are presented in chapters 5 and 7. But the models of chapter 3 and 4 have been developed in the belief that good engineering practices can be followed.

As discussed in section 1.3, this book takes a systems engineering approach by conceiving of the final product as a system. *Systems engineering* has been defined as the process of specifying, designing, implementing, validating, deploying, and maintaining sociotechnical systems. A useful analogy is the process of constructing a building. When someone plans the building of a house, the first thing that needs to be done is a sketch. The sketch roughly specifies the location, size, shape, and purpose of the building and the layout and purposes of its rooms. It proceeds from conversations between the customer and architect.

Next, the architect turns the sketch into the architect's drawings, which include floor plans, cutaways, and pictures of the house-to-be. The purpose of the drawings is to enable the owner to relate to them and either agree or disagree with its different parts and aspects. We can call this process *requirements engineering*, because its main purpose is to understand and specify requirements for the building. Moreover, as the architect normally creates the drawings by using a computer-aided design (CAD) system of some sort, it could also be possible to simulate some aspects of the building, such as how doors and windows are opened, or what kind of interior and functionalities the building should include. Both the sketch and the architect's drawings model the final product—the building—from the *owner's perspective*.

As the next step, the architect's drawings are turned into the architect's plans. The plans constitute the *designer's perspective* of the final product. They consist of detailed descriptions of the building-to-be from different aspects, including site work, plumbing, electrical systems, communication systems, masonry, wood structure, and

so forth. The architect's plans specify the materials to be used for construction work and serve as a basis for negotiation with a general contractor.

Finally, the contractor transforms the architect's plans into the contractor's plans, which represent the *builder's perspective*. The contractor's plans essentially provide a "how to build it" description. They define the order of building activities and consider the technology available to the contractor. There can also be the so-called shop plans, which are out-of-context specifications of the parts, or functional areas that are outsourced to subcontractors.

A systems engineering process is in many ways similar to the process of constructing a building. First, we sketch the system as situated in its environment. The models employed for this purpose may, for example, include *use cases*—a means of specifying required usages of a system. This is the system modeled from the *owner's perspective*. The owner's perspective may also comprise scenarios that can be simulated. The *designer's perspective* consists of various models that describe from different aspects how the system should be designed. A standard widely accepted by the software industry for this purpose is UML. The *builder's perspective* is based on the designer's perspective, but considers specific languages, technologies, and tools to be used and defines the order of systems engineering activities. Changing perspectives is not always straightforward. For example, a designer has to consider the languages, technologies, and tools of the problem domain.

The order in which we represent an agent-oriented modeling process in this book has been influenced by the systems engineering perspective. Different kinds of models to be presented in Chapter 3 are to allow inclusion of different perspectives of the system.

1.6 Emergent Behavior

Models proceed from the requirements of an owner to the design expertise of a designer, and are then handed over to a developer to be implemented and deployed. Implicit in this previous sentence, and indeed in our discussion in this chapter, has been that building a multiagent system proceeds from the top down. If the three stakeholders—the owner, the designer, and developer—are different people, it is more natural for the order to proceed top-down. It is possible, though not advisable, for development of a system to proceed from the bottom up. The developer can implement a system, which can retrospectively be designed, and the underlying motivation and purpose be determined through use. This makes some sense if the three perspectives are that of the same person, and the bottom-up approach essentially becomes rapid prototyping. Otherwise, the owner is completely dependent on the will of the developer.

To contrast the top-down and bottom-up approaches, let us reconsider the example of building a house. The top-down approach starts with the goal of building a house with particular objectives. These days, for example, one objective might be to be environmentally friendly. The objectives are explained to an architect, who comes up with a design. The design is then fully specified and agreed upon and given to a builder to construct. The bottom-up approach has builders starting to build. This is presumably what happens in the insect world. Termites build quite complex nests for the termite colony to live in, assembled from the bottom up. To the best of our knowledge, there is no "head termite" with an overall plan. Rather, the nest emerges from the simple behavior of the termites. This is sometimes called *emergent behavior* or *self-organizing behavior*.

Emergent behavior is not just a feature of the insect world. City life or group culture is often emergent. Creative cities work by collocating a group of people and hoping that synergies happen. Several successful cities have developed in this way throughout history.

We are aware of two classes of agent-oriented applications in which an exploratory, bottom-up approach is natural, and in which there has been extensive research. One is the field of mobile robots. Two of the agents discussed in section 1.2, the Roomba vacuum cleaner and the AIBO robotic dog, are essentially mobile robots. Vacuum cleaning can be approached both from the top down and from the bottom up. Top-down, there is a clear overall goal of removing dust, especially from carpets. At the design stage, deciding which agent cleans which surface needs to be determined. Then appropriate devices are deployed. Bottom-up, a vacuuming device is built. How well it works is factored in and other steps may be needed either to improve the overall quality goal, or to augment the cleaning with other devices. Roomba clearly fits in the bottom-up stage. Similarly, AIBO is a robotic pet designed with particular capabilities that it is hoped are entertaining to the owners that buy them. Although Sony, the manufacturer, clearly had design specifications, once they are in an environment, they interact in their own way.

The second class of examples in which emergent behavior is interesting is that of simulation and modeling. To give an example, suppose that one was holding a large sport event with tens of thousands of attendees. One would like to know whether the event was suitably hosted. One way to do this is to build a model and run it to see what behaviors by attendees may emerge.

Many researchers have investigated "ant algorithms," which are essentially bottom-up explorations by simple agents to achieve an objective. Such algorithms are interesting and worthy of further research. We do not explore them, or the larger issue of emergent behavior, any further in this book. For additional information, the reader is referred to De Wolf and Holvoet 2005.

1.7 A Quick History of Programming Paradigms

We used the previous sections to set the development of agent concepts in context. We mentioned in section 1.2 that agents connote intelligent assistance. The desire to create intelligence in a computer has been a constant theme for computing, dating back to Alan Turing, the father of computer science. Sixty years of research on computing, fifty years of research in AI, and forty years of research in software engineering are all relevant to how agents are conceptualized and deployed. We give a brief history here.

Early computing research focused on programming, not modeling. The earliest programming languages reflected a procedural view of the computer world. Programmers wrote instructions in the languages for the computer, which was envisaged as a machine for executing a sequence of instructions. Analysts, perhaps loosely equivalent to modelers, used flow charts to represent the instructions that needed to be followed at a more abstract level.

Early computer languages, such as Fortran and COBOL, typify the procedural view. A business application written in COBOL could be viewed as a collection of financial agents cooperating to produce a business report. The analyst, however, had to specify the complete control flow for the coder, and there was no advantage in the agent perspective. It is worth noting that the SWARMM system mentioned earlier in this chapter superseded a system written in Fortran. It was hard to maintain the Fortran system and very difficult to engage with the end users—the pilots. The agent-oriented system worked much better.

Object orientation is an alternate to the procedural view of computing. It views a program as a collection of objects sending messages to each other rather than as a sequential list of instructions. Though object-oriented languages such as Simula appeared as early as the 1960s, the popularity of object-oriented computing grew through Smalltalk and especially C++ only in the 1980s. By the 1990s, object orientation had become the preferred paradigm for developing applications for a distributed, complex world.

It is plausible to view agent-oriented computing as an extension of object-oriented computing. Amusingly, agents have been described as ''objects on steroids.'' Such a description, however, is not helpful for developers, or politically correct in a climate of cracking down on drugs in sport. In our experience teaching programming, trying to understand one paradigm in terms of another often limits one's understanding. Indeed, some of the models to be described in chapter 3 were prompted by a need to differentiate them from object-oriented models that students were using without gaining an appreciation of how to think from an agent perspective. So although we do appropriate some object-orientation notation in our models, we downplay the

connection between the agent-oriented and object-oriented ways of thinking in this book.

The meteoric rise of the Internet's popularity and the need to deploy applications on a variety of platforms led to the emergence and popularity of Java. To some extent, Java is an object-oriented applications development language, in contrast to C++, which is more of an object-oriented systems development language. Java's simplification of network interactions has made it popular for developing agent applications, as has its platform independence. It is no coincidence that most of the agent programming languages described in chapter 5 are built on Java.

Having described Java as a language suitable for developing applications, we briefly mention scripting languages. These languages make it quick to develop and deploy applications. Some have been developed for agents such as the now-defunct Telescript and Aglets from IBM. Using a scripting language tends to discourage thinking in terms of systems and models. Although we are not averse to people using scripting languages, such languages are not relevant to the book and are not discussed further.

We turn now to another programming paradigm: declarative programming. The declarative paradigm has been influential in agent research. It is connected with the body of work in formalizing agent behavior using various logics. Early researchers in AI advocated the physical symbol system hypothesis, that a machine manipulating physical symbols had the necessary and sufficient means to be intelligent and exhibit intelligent behavior. Lisp was an early programming language that was adopted in applications that manipulated symbols.

Predicate logic was natural to use when taking a symbolic view. A prototypical AI project conducted from 1966 to 1972 at the Artificial Intelligence Center at what was then the Stanford Research Institute in the United States is illustrative. The project centred on a mobile robot system nicknamed Shakey. Shakey moved around in a physical environment consisting of makeshift "rooms" and blocks that could be pushed from room to room. Shakey can definitely be considered an agent.

Part of the project involved Shakey formulating and executing plans to move blocks from room to room. Shakey's world was described in terms of logic formulae such as `in(room1,shakey)`, denoting that Shakey was in room 1. The planning problem for Shakey was to move blocks from room to room. Knowing which room the robot was in was a challenge for the vision system of the time, but that need not concern us here. The planning problem was formulated in terms of STRIPS (Stanford Research Institute Problem Solver) operators. Operators were expressed in terms of preconditions, which needed to be true if the operator were applicable, and postconditions, which became true after the operator was executed. For example, an operator's instruction `move(block1, room1, room2)` might be invoked to

move block1 from room 1 to room 2. The precondition would be that Shakey was in room 1, and the postcondition that Shakey was in room 2 and no longer in room 1. Of course, the operators were expressed more generically. Planning was stringing together a sequence of operators that could achieve a desired state. STRIPS, the planning approach, is intuitive, and has been influential in AI.

Shakey's system is declarative, in the sense that the system developer merely expressed (declared) a set of operators. The underlying planning was left to the system. Declarative programming is ideal in theory. In practice, how the operators are expressed can have considerable influence on the effectiveness of the planner. The analogous declarative view of systems is to express a series of axioms as statements of logic, and leave it to an underlying theorem prover to draw the conclusions or reason appropriately. How axioms are expressed affects how well a system may perform.

Around the time of the Shakey project, experiments with theorem proving led to the development of the programming language Prolog. Prolog is a good language for many AI applications. In particular, Prolog is useful for prototyping logic reasoning systems. Several of the agent programming languages described in chapter 5 build on Prolog, or are at least influenced by it.

Formulating statements in logic and relying on an underlying theorem prover to prove whether they are correct or to generate a plan is orthogonal to the concerns of this book. Declarative programming is a good paradigm. Being precise in one's knowledge is to be commended. Also, it would be helpful, as many researchers advocate, to delegate the reasoning to some interpreter. However, considering the execution as proving a theorem masks the systems approach of how agents interact and whose responsibilities and what constraints there are, which is how we model systems. So despite the large amount of research in logic, we do not take that focus here.

There was a strong reaction to the declarative view of intelligence in the 1980s. The contrasting view was a reactive approach to achieve intelligence. The idea is to build more intelligent activities on top of core functionalities. An analogy can be made with human learning. Infants learn to walk and talk in their first years. Greater intelligence comes later, built on top of our skills in walking and talking. Our robust intelligence depends to a great degree on the robustness of the underlying mechanism. This book is not, however, a book about intelligence, but rather about models, and how they can be developed top-down as part of a systems engineering life cycle.

To summarize the chapter, we advocate conceiving of the world in which software must operate as a multiagent system operating in an environment subject to rules and policies. There will be models to reflect important aspects of multiagent systems to

aid understanding. The conceptual space that we look at will be discussed in more detail in chapter 2. The models themselves will be presented in chapter 3, and applied in later chapters.

1.8 Background

The background section at the end of each chapter is where we will supply more detailed information about the references cited in each chapter, a more informal way of handling references than footnotes. It also serves as a place for suggesting further reading. Needless to say, the size and style of the background section will vary from chapter to chapter.

This first chapter sets the scene for agents. The agent paradigm has become much more prevalent over the past decade. For example, the agent metaphor is used as the unifying image for AI in the leading AI textbook by Russell and Norvig (2002).

In the late 1990s, there was a spate of survey articles about intelligent agents, which make useful additional readings. Two of the better known examples are Wooldridge and Jennings 1995 and Nwana 1995. The one most useful in our experience for graduate students is Wooldridge 1999.

A common approach to introducing agents is to spend time discussing definitions. The amusing definition of "objects on steroids" mentioned in this chapter has been introduced by Parunak (2000, p. 13). For those readers seeking a more extensive discussion of definitions, we recommend the article by Franklin and Graesser (1997). In this book, we have not agonized over the definition of an agent. In classes, votes are taken on what people regard as agents. There is always diversity, and usually some change of opinion by the end of the class. Indeed, a popular examination question used in the University of Melbourne agent class is why does it not matter that there is no exact agreed-upon definition of an agent.

The digital pet Web sites for Neopets, Webkinz, and Petz4fun mentioned in the chapter are http://www.neopets.com, http://www.webkinz.com, and http://www.petz4fun.com, respectively.

The analogy between constructing a building and developing a software system is by John Zachman (1987).

UML is defined in OMG 2007.

A place to read more about the SWARMM system is Heinze et al. 2002. More information about the STOW-97 simulation can be found in Laird et al. 1998.

Our understanding of the role of agents in the film trilogy *Lord of the Rings* came from footage on the extended DVDs of the movies. In fact, the film producer Peter Jackson explaining agents is the best endorsement of agents that we have seen.

The leading proponent of the reactive approach to intelligence is Rodney Brooks. One of the most well-known papers by him is Brooks 1991.

Another influential critic of classical AI in the 1980s was Terry Winograd. He published a book with Fernando Flores entitled *Understanding Computers and Cognition* (1986) that argued that there were limits to computer intelligence and that computers should be used to facilitate human communication. Although we don't explore the connection, philosophically, our holistic view of multiagent systems and thinking about roles, responsibilities, and constraints are in keeping with this Winograd and Flores philosophy. Their philosophy also influenced William Clancey, who developed a theory of activity that we draw on in chapters 2 and 3.

Having given the references, we'd like to comment on the following. Many of the references were acquired by searching the Web using Google. For example, the definitions of models were extracted from the answers generated from typing the question "What is a model?" into Google. The exercise was interesting and illustrates well the diversity of the use of a common English word across a range of fields. We encourage readers to try such searches for themselves.

Several of the sources were adapted or quoted directly from Wikipedia articles. Wikipedia's status as a reference of record is admittedly controversial. However, it typically reflects popular opinion, supports the impression that we are imparting of complex distributed sociotechnical systems, and is appropriate for our purposes in this chapter. It would be fun to discuss why aspects of Wikipedia are controversial relating to authorship and the ability to change records, but that is beyond the scope of this book.

The Wikipedia references on e-commerce and intelligent homes are respectively Wikipedia 2006a and 2006b. The Wikipedia reference on computer viruses is Wikipedia 2006c.

Exercises for Chapter 1

1. Discuss which of the following you would consider an agent, and explain why or why not. If you are unfamiliar with the example, try looking it up on the Internet.

- The Furby toy
- The Australian device for autonomously cleaning swimming pools, known as a Kreepy Krauly
- An unmanned aerial vehicle (UAV)
- A search engine such as Google

2. Identify the various agents involved in the following:

- A book publishing company

- A university
- A church or your local religious organization
- Australia, Estonia, or a country of your choice

3. Search the Internet for an application of agents in an area of personal interest.

4. The robotic soccer competition was augmented by having a competition, called RoboCup Rescue, for teams of robots working in a rescue site. This was partly triggered by the devastation caused by the Kobe earthquake in 1995. Discuss what needs to be modeled to make such a competition useful. What features should be included in a model?

5. Consider what would be useful to model in the transportation system of your city.